An Open-Source RTL-to-STA Flow for Synchronous FIFO Design - Leveraging SystemVerilog (EDA Playground), Yosys, and OpenSTA for PPA Characterization.

Submitted by,

KONAPALA SAI KUMAR.

TABLE OF CONTENTS.

Design and Verification of a Synchronous FIFO in SystemVerilog

The Synchronous FIFO: A Cornerstone of Digital Data Flow

Introduction to Data Buffering and Synchronization

In complex digital systems, data is constantly being transferred between different functional blocks. These blocks may produce and consume data at varying or bursty rates. To manage this data flow efficiently and prevent data loss, a buffering mechanism is required. The First-In-First-Out (FIFO) memory is a fundamental digital circuit that serves this purpose, acting as an elastic buffer that temporarily stores data in the order it was received. The first data element written into the FIFO is the first one to be read out, analogous to a queue.

FIFOs are broadly categorized as synchronous or asynchronous. An asynchronous FIFO is essential for safely transferring data between modules operating on different, unrelated clocks—a process known as Clock Domain Crossing (CDC). In contrast, a **synchronous FIFO** operates with a single, common clock for both its write and read operations. This simplifies the design, as all operations are synchronized to the same time base, making it an ideal solution for high-speed systems where different modules share a clock but require rate-matching capabilities.

FIFO Input and Output Pins

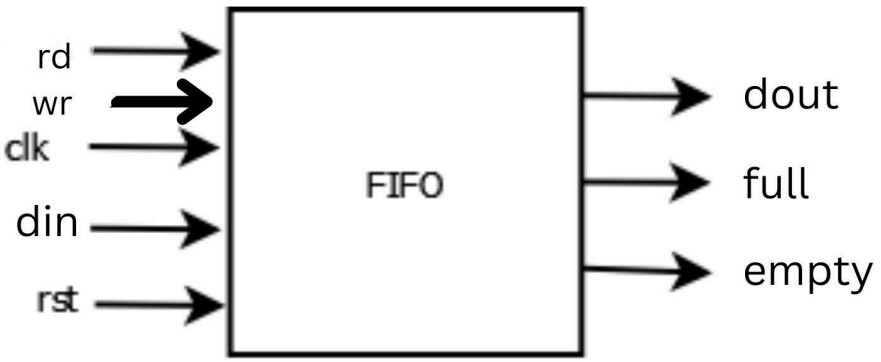


Fig: Block diagram of Synchronous FIFO.

The FIFO module communicates with external logic through a set of standardized input and output pins. Each pin serves a specific role in controlling the flow of data and reporting the status of the buffer.

- **`clk` (Input):** The master clock signal. All operations within the FIFO, including writing, reading, and updating internal pointers, are synchronized to the positive edge of this clock.
- **`rst` (Input):** The active-high reset signal. When asserted, it asynchronously resets the FIFO to its initial state, clearing all stored data and resetting the read and write pointers.
- **`wr` (Input):** The write enable signal. When asserted high, it signals the FIFO to write the data present on the `din` bus into the memory on the next positive clock edge, provided the FIFO is not full.
- **`rd` (Input):** The read enable signal. When asserted high, it signals the FIFO to present the oldest data element on the `dout` bus on the next positive clock edge, provided the FIFO is not empty.
- **`din` (Input [7:0]):** The 8-bit data input bus. The data on this bus is written into the FIFO's memory during a write operation.
- **`dout` (Output [7:0]):** The 8-bit data output bus. During a read operation, the oldest data element from the FIFO is driven onto this bus.
- **`empty` (Output):** A status flag that is asserted high when the FIFO contains no data. This signal is used to prevent underflow errors (reading from an empty buffer).
- **`full` (Output):** A status flag that is asserted high when the FIFO has no available space to store new data. This signal is used to prevent overflow errors (writing to a full buffer).

Design Code

The Design Under Test (DUT) is a synchronous FIFO implemented in Verilog. It features an 8-bit data width and a depth of 16 locations. The design uses dedicated write and read pointers and a counter to manage the data flow and generate the `full` and `empty` status flags.

1.1 DESIGN CODE

```
// Code your design here
module FIFO(input clk, rst, wr, rd,
            input [7:0] din, output reg [7:0] dout,
            output empty, full);

  // Pointers for write and read operations
  reg [3:0] wptr = 0, rptr = 0;

  // Counter for tracking the number of elements in the FIFO
  reg [4:0] cnt = 0;

  // Memory array to store data
  reg [7:0] mem [15:0];

  always @(posedge clk)
    begin
      if (rst == 1'b1)
        begin
          // Reset the pointers and counter when the reset signal is asserted
          wptr <= 0;
          rptr <= 0;
          cnt <= 0;
        end
      else if (wr && !full)
        begin
          // Write data to the FIFO if it's not full
          mem[wptr] <= din;
          wptr <= wptr + 1;
          cnt <= cnt + 1;
        end
      else if (rd && !empty)
        begin
          // Read data from the FIFO if it's not empty
          dout <= mem[rptr];
          rptr <= rptr + 1;
          cnt <= cnt - 1;
        end
    end

  // Determine if the FIFO is empty or full
  assign empty = (cnt == 0) ? 1'b1 : 1'b0;
  assign full = (cnt == 16) ? 1'b1 : 1'b0;

endmodule
```

1.2 INTERFACE

```
// Define an interface for the FIFO
interface fifo_if;

  logic clock, rd, wr; // Clock, read, and write signals
  logic full, empty; // Flags indicating FIFO status
  logic [7:0] data_in; // Data input
  logic [7:0] data_out; // Data output
  logic rst; // Reset signal

endinterface
```

Verification Environment Testbench

To validate the functional correctness of the FIFO design, a comprehensive class-based verification environment was developed in SystemVerilog. This testbench employs a modular architecture with separate components for stimulus generation, driving, monitoring, and checking, all communicating via mailboxes.

Code snippet
// Code your testbench here
// or browse Examples

2.1 TRANSACTION

```
class transaction;

  rand bit oper; // Randomized bit for operation control (1 or 0)
  bit rd, wr; // Read and write control bits
  bit [7:0] data_in; // 8-bit data input
  bit full, empty; // Flags for full and empty status
  bit [7:0] data_out; // 8-bit data output

  constraint oper_ctrl {
    oper dist {1 :/ 50 , 0 :/ 50}; // Constraint to randomize 'oper' with 50% probability of 1 and 50% probability of 0
  }

endclass

//////////////////////////////////////////////////
```

2.2 GENERATOR

```
class generator;

  transaction tr; // Transaction object to generate and send
  mailbox #(transaction) mbx; // Mailbox for communication
  int count = 0; // Number of transactions to generate
  int i = 0; // Iteration counter

  event next; // Event to signal when to send the next transaction
  event done; // Event to convey completion of requested number of transactions

  function new(mailbox #(transaction) mbx);
    this.mbx = mbx;
    tr = new();
  endfunction;

  task run();
    repeat (count) begin
      assert (tr.randomize) else $error("Randomization failed");
      i++;
      mbx.put(tr);
      $display("[GEN] : Oper : %0d iteration : %0d", tr.oper, i);
      @(next);
    end -> done;
  endtask

endclass
//////////////////////////////////////////////
```

2.3 DRIVER

```
class driver;

  virtual fifo_if fif; // Virtual interface to the FIFO
  mailbox #(transaction) mbx; // Mailbox for communication
  transaction datac; // Transaction object for communication

  function new(mailbox #(transaction) mbx);
    this.mbx = mbx;
  endfunction;

  // Reset the DUT
  task reset();
    fif.rst <= 1'b1;
    fif.rd <= 1'b0;
    fif.wr <= 1'b0;
    fif.data_in <= 0;
    repeat (5) @(posedge fif.clock);
    fif.rst <= 1'b0;
    $display("[DRV] : DUT Reset Done");
    $display("------------------------------------------");
  endtask

  // Write data to the FIFO
  task write();
    @(posedge fif.clock);
    fif.rst <= 1'b0;
    fif.rd <= 1'b0;
    fif.wr <= 1'b1;
    fif.data_in <= $urandom_range(1, 10);
    @(posedge fif.clock);
    fif.wr <= 1'b0;
    $display("[DRV] : DATA WRITE data : %0d", fif.data_in);
    @(posedge fif.clock);
  endtask


  // Read data from the FIFO
  task read();
    @(posedge fif.clock);
    fif.rst <= 1'b0;
    fif.rd <= 1'b1;
    fif.wr <= 1'b0;
    @(posedge fif.clock);
    fif.rd <= 1'b0;
    $display("[DRV] : DATA READ");
    @(posedge fif.clock);
  endtask

  // Apply random stimulus to the DUT
  task run();
    forever begin
      mbx.get(datac);
      if (datac.oper == 1'b1)
        write();
      else
        read();
    end
  endtask

endclass

//////////////////////////////////////////////////////
```

2.4 MONITOR

```
class monitor;

  virtual fifo_if fif; // Virtual interface to the FIFO
  mailbox #(transaction) mbx; // Mailbox for communication
  transaction tr; // Transaction object for monitoring

  function new(mailbox #(transaction) mbx);
    this.mbx = mbx;
  endfunction;

  task run();
    tr = new();

    forever begin
      repeat (2) @(posedge fif.clock);
      tr.wr = fif.wr;
      tr.rd = fif.rd;
      tr.data_in = fif.data_in;
      tr.full = fif.full;
      tr.empty = fif.empty;
      @(posedge fif.clock);
      tr.data_out = fif.data_out;

      mbx.put(tr);
      $display("[MON] : Wr:%0d rd:%0d din:%0d dout:%0d full:%0d empty:%0d", tr.wr, tr.rd, tr.data_in, tr.data_out, tr.full, tr.empty);
    end

  endtask

endclass

////////////////////////////////////////////////////
```

2.5 SCOREBOARD

```
class scoreboard;

  mailbox #(transaction) mbx; // Mailbox for communication
  transaction tr; // Transaction object for monitoring
  event next;
  bit [7:0] din[$]; // Array to store written data
  bit [7:0] temp; // Temporary data storage
  int err = 0; // Error count

  function new(mailbox #(transaction) mbx);
    this.mbx = mbx;
  endfunction;

  task run();
    forever begin
      mbx.get(tr);
      $display("[SCO] : Wr:%0d rd:%0d din:%0d dout:%0d full:%0d empty:%0d", tr.wr, tr.rd, tr.data_in, tr.data_out, tr.full, tr.empty);

      if (tr.wr == 1'b1) begin
        if (tr.full == 1'b0) begin
          din.push_front(tr.data_in);
          $display("[SCO] : DATA STORED IN QUEUE :%0d", tr.data_in);
        end
        else begin
          $display("[SCO] : FIFO is full");
        end
        $display("-----------------------------------------");
      end

      if (tr.rd == 1'b1) begin
        if (tr.empty == 1'b0) begin
          temp = din.pop_back();

          if (tr.data_out == temp)
            $display("[SCO] : DATA MATCH");
          else begin
            $error("[SCO] : DATA MISMATCH");
            err++;
          end
        end
        else begin
          $display("[SCO] : FIFO IS EMPTY");
        end

        $display("-----------------------------------------");
      end

      -> next;
    end
  endtask

endclass
```

2.6 ENVIRONMENT

```
202 class environment;
203
204   generator gen;
205   driver drv;
206   monitor mon;
207   scoreboard sco;
208   mailbox #(transaction) gdmbx; // Generator + Driver mailbox
209   mailbox #(transaction) msmbx; // Monitor + Scoreboard mailbox
210   event nextgs;
211   virtual fifo_if fif;
212
213   function new(virtual fifo_if fif);
214     gdmbx = new();
215     gen = new(gdmbx);
216     drv = new(gdmbx);
217     msmbx = new();
218     mon = new(msmbx);
219     sco = new(msmbx);
220     this.fif = fif;
221     drv.fif = this.fif;
222     mon.fif = this.fif;
223     gen.next = nextgs;
224     sco.next = nextgs;
225   endfunction
226
227   task pre_test();
228     drv.reset();
229   endtask
230
231   task test();
232     fork
233       gen.run();
234       drv.run();
235       mon.run();
236       sco.run();
237     join_any
238   endtask
239
240   task post_test();
241     wait(gen.done.triggered);
242     $display("--------------------------------------------");
243     $display("Error Count :%0d", sco.err);
244     $display("--------------------------------------------");
245     $finish();
246   endtask
247
248   task run();
249     pre_test();
250     test();
251     post_test();
252   endtask
253 endclass
```

2.7 TESTBENCH TOP

```
256
257 module tb;
258
259   fifo_if fif();
260   FIFO dut (fif.clock, fif.rst, fif.wr, fif.rd, fif.data_in, fif.data_out, fif.empty, fif.full);
261
262   initial begin
263     fif.clock <= 0;
264   end
265
266   always #10 fif.clock <= ~fif.clock;
267
268   environment env;
269
270   initial begin
271     env = new(fif);
272     env.gen.count = 10;
273     env.run();
274   end
275
276   initial begin
277     $dumpfile("dump.vcd");
278     $dumpvars;
279   end
280
281 endmodule
282
```

Code and Architecture Analysis

Analysis of the RTL Design (`FIFO` module)

The Verilog design implements a synchronous FIFO with a depth of 16 and a data width of 8 bits. Its operation is straightforward and robust for single-clock domain applications.

- **State Elements:** The design uses a 16-entry memory array (`mem`), a 4-bit write pointer (`wptr`), a 4-bit read pointer (`rptr`), and a 5-bit counter (`cnt`) to track the number of stored elements.
- **Control Logic:** All logic is contained within a single `always @(posedge clk)` block, ensuring synchronous behavior.
    - **Reset:** On an active-high reset, all pointers and the counter are cleared to zero.
    - **Write Logic:** A write operation (`wr &&!full`) stores the input data (`din`) at the `wptr` location and increments both the `wptr` and the `cnt`.
    - **Read Logic:** A read operation (`rd &&!empty`) places the data from the `rptr` location onto the `dout` bus and increments the `rptr` while decrementing the `cnt`.
- **Status Flags:** The `empty` and `full` flags are generated combinatorially based on the value of the `cnt`. `empty` is asserted when `cnt` is 0, and `full` is asserted when

`cnt` is 16. This counter-based approach is a reliable method for status generation in synchronous FIFOs.

Analysis of the SystemVerilog Verification Environment

The testbench employs a well-defined, class-based architecture that separates the concerns of stimulus generation, driving, monitoring, and checking.

- **Component Communication:** The components communicate using SystemVerilog `mailboxes`. The `generator` and `driver` share one mailbox (`gdmbx`), while the `monitor` and `scoreboard` share another (`msmbx`). This decouples the components, allowing for a modular structure.

- **Stimulus Generation (`generator`):** The generator's role is to decide whether the next operation should be a read or a write. It does this by randomizing an `oper` bit and placing a transaction object into the mailbox. It does not generate the actual data to be written; that is handled by the driver.

- **Driving (`driver`):** The driver retrieves the transaction from the mailbox and, based on the `oper` bit, executes either a `write()` or `read()` task. The `write()` task generates random data (`$urandom_range`) and drives the DUT's input signals according to the FIFO write protocol. The `read()` task drives the signals for a read operation.

- **Monitoring (`monitor`):** The monitor passively observes the FIFO interface, samples all relevant signals (`wr`, `rd`, `data_in`, `data_out`, `full`, `empty`), encapsulates them into a transaction object, and sends this object to the scoreboard via its mailbox.

- **Checking (`scoreboard`):** The scoreboard contains the core verification logic. It maintains a local queue (`din[$]`) which serves as a behavioral model of the FIFO.
    - When it observes a valid write operation (`tr.wr == 1'b1` and `tr.full == 1'b0`), it pushes the written data into its internal queue.
    - When it observes a valid read operation (`tr.rd == 1'b1` and `tr.empty == 1'b0`), it pops the expected data from its queue and compares it with the data read from the DUT (`tr.data_out`). Any mismatch results in an error. This approach correctly models the first-in, first-out behavior.

- **Synchronization (`event`):** An event (`nextgs`) is used to synchronize the generator and the scoreboard. After the scoreboard processes a transaction, it triggers the event, signaling the generator to create the next transaction. This ensures that stimulus is generated at a pace that the verification environment can handle, preventing the mailboxes from overflowing.

- **Test Flow (`environment`):** The `environment` class instantiates all components and orchestrates the test sequence through `pre_test` (reset), `test` (concurrently running all components), and `post_test` (waiting for completion and reporting results) tasks.

Simulation Results and Analysis

The functional correctness of the synchronous FIFO design was verified by running the SystemVerilog testbench. The simulation produced both a detailed log file, which provides a textual trace of the testbench's operations, and a waveform file, which offers a visual representation of the DUT's signal-level behavior over time. Together, these outputs provide a comprehensive view of the verification process and the design's response to stimuli.

Log Output Analysis

The simulation log provides a step-by-step account of the interactions between the verification components. Each log entry is tagged with the component that generated it (`, `[GEN]`, `[MON]`,), allowing for a clear trace of the test flow.

```
# KERNEL: ----------------------------------------
# KERNEL: [GEN] : Oper : 1 iteration : 5
# KERNEL: [DRV] : DATA WRITE data : 2
# KERNEL: [MON] : Wr:1 rd:0 din:2 dout:0 full:0 empty:0
# KERNEL: [SCO] : Wr:1 rd:0 din:2 dout:0 full:0 empty:0
# KERNEL: [SCO] : DATA STORED IN QUEUE :2
# KERNEL: ----------------------------------------
# KERNEL: [GEN] : Oper : 0 iteration : 6
# KERNEL: [DRV] : DATA READ
# KERNEL: [MON] : Wr:0 rd:1 din:2 dout:2 full:0 empty:0
# KERNEL: [SCO] : Wr:0 rd:1 din:2 dout:2 full:0 empty:0
# KERNEL: [SCO] : DATA MATCH
# KERNEL: ----------------------------------------
# KERNEL: [GEN] : Oper : 0 iteration : 7
# KERNEL: [DRV] : DATA READ
# KERNEL: [MON] : Wr:0 rd:1 din:2 dout:7 full:0 empty:0
# KERNEL: [SCO] : Wr:0 rd:1 din:2 dout:7 full:0 empty:0
# KERNEL: [SCO] : DATA MATCH
# KERNEL: ----------------------------------------
# KERNEL: [GEN] : Oper : 0 iteration : 8
# KERNEL: [DRV] : DATA READ
# KERNEL: [MON] : Wr:0 rd:1 din:2 dout:9 full:0 empty:0
# KERNEL: [SCO] : Wr:0 rd:1 din:2 dout:9 full:0 empty:0
# KERNEL: [SCO] : DATA MATCH
# KERNEL: ----------------------------------------
```

```
# KERNEL: --------------------------------------
# KERNEL: [GEN] : Oper : 1 iteration : 5
# KERNEL: [DRV] : DATA WRITE data : 2
# KERNEL: [MON] : Wr:1 rd:0 din:2 dout:0 full:0 empty:0
# KERNEL: [SCO] : Wr:1 rd:0 din:2 dout:0 full:0 empty:0
# KERNEL: [SCO] : DATA STORED IN QUEUE :2
# KERNEL: --------------------------------------
# KERNEL: [GEN] : Oper : 0 iteration : 6
# KERNEL: [DRV] : DATA READ
# KERNEL: [MON] : Wr:0 rd:1 din:2 dout:2 full:0 empty:0
# KERNEL: [SCO] : Wr:0 rd:1 din:2 dout:2 full:0 empty:0
# KERNEL: [SCO] : DATA MATCH
# KERNEL: --------------------------------------
# KERNEL: [GEN] : Oper : 0 iteration : 7
# KERNEL: [DRV] : DATA READ
# KERNEL: [MON] : Wr:0 rd:1 din:2 dout:7 full:0 empty:0
# KERNEL: [SCO] : Wr:0 rd:1 din:2 dout:7 full:0 empty:0
# KERNEL: [SCO] : DATA MATCH
# KERNEL: --------------------------------------
# KERNEL: [GEN] : Oper : 0 iteration : 8
# KERNEL: [DRV] : DATA READ
# KERNEL: [MON] : Wr:0 rd:1 din:2 dout:9 full:0 empty:0
# KERNEL: [SCO] : Wr:0 rd:1 din:2 dout:9 full:0 empty:0
# KERNEL: [SCO] : DATA MATCH
# KERNEL: --------------------------------------
```

```
# KERNEL: --------------------------------------
# KERNEL: [GEN] : Oper : 1 iteration : 9
# KERNEL: [DRV] : DATA WRITE data : 10
# KERNEL: [MON] : Wr:1 rd:0 din:10 dout:9 full:0 empty:0
# KERNEL: [SCO] : Wr:1 rd:0 din:10 dout:9 full:0 empty:0
# KERNEL: [SCO] : DATA STORED IN QUEUE :10
# KERNEL: --------------------------------------
# KERNEL: [GEN] : Oper : 0 iteration : 10
# KERNEL: [DRV] : DATA READ
# KERNEL: [MON] : Wr:0 rd:1 din:10 dout:7 full:0 empty:0
# KERNEL: [SCO] : Wr:0 rd:1 din:10 dout:7 full:0 empty:0
# KERNEL: [SCO] : DATA MATCH
# KERNEL: --------------------------------------
# KERNEL: ----------------------------------------------
# KERNEL: Error Count :0
# KERNEL: ----------------------------------------------
# RUNTIME: Info: RUNTIME_0068 testbench.sv (245): $finish called.
# KERNEL: Time: 690 ns,  Iteration: 1,  Instance: /tb,  Process: @INITIAL#271_2@.
# KERNEL: stopped at time: 690 ns
# VSIM: Simulation has finished. There are no more test vectors to simulate.
# VSIM: Simulation has finished.
Done
```

Analysis of Synchronous FIFO Output Log

Initialization: The log begins with the `: DUT Reset Done` message, confirming that the testbench properly initialized the FIFO by asserting the reset signal before commencing any operations.

Data Flow and FIFO Principle

The log shows a sequence of **Write** (wr=1, {rd=0) and **Read** (wr=0,rd=1) operations, with the FIFO ensuring that the data read out (dout) matches the data that was written in (din) in the exact order it was received.

1. **Data Entry (Write Operations):**
    - Data is written into the FIFO sequentially, as indicated by wr=1.
    - The written data values (din) are: **2, 7, 9, 7, 2**, and **10** (up to iteration 9).
    - **Data Exit (Read Operations):**
    - Data is read from the FIFO sequentially, as indicated by rd=1.
    - The data values read out (dout) are: **2, 7, 9, 7**, and **10** (up to iteration 10).
    - The empty flag remains at 0 (not empty) during the reads, indicating data is available.

Iteration (Oper)	Action (DRV)	Data In (din)	wr	rd	Status (full, empty)
1	WRITE	**2**	1	0	0, 0
2	WRITE	**7**	1	0	0, 0
3	WRITE	**9**	1	0	0, 0
4	WRITE	**7**	1	0	0, 0
5	WRITE	**2**	1	0	0, 0
9	WRITE	**10**	1	0	0, 0

Table 1: Data Entry into FIFO.

2. Data Exit (Read Operations):

- Data is read from the FIFO sequentially, as indicated by rd=1.
- The data values read out (dout) are: **2, 7, 9, 7**, and **10** (up to iteration 10).
- The empty flag remains at 0 (not empty) during the reads, indicating data is available.

Iteration (Oper)	Action (DRV)	Data Out (dout)	wr	rd	Expected Data (SCO)	Match
6	READ	**2**	0	1	$\mathrm{din} = 2$ (Iter 1)	**MATCH**
7	READ	**7**	0	1	$\mathrm{din} = 7$ (Iter 2)	**MATCH**
8	READ	**9**	0	1	$\mathrm{din} = 9$ (Iter 3)	**MATCH**
9	READ	**7**	0	1	$\mathrm{din} = 7$ (Iter 4)	**MATCH**
10	READ	**10**	0	1	$\mathrm{din} = 10$ (Iter 9)	**MATCH**

Table 2: Data Exit from FIFO.

Successful FIFO Implementation

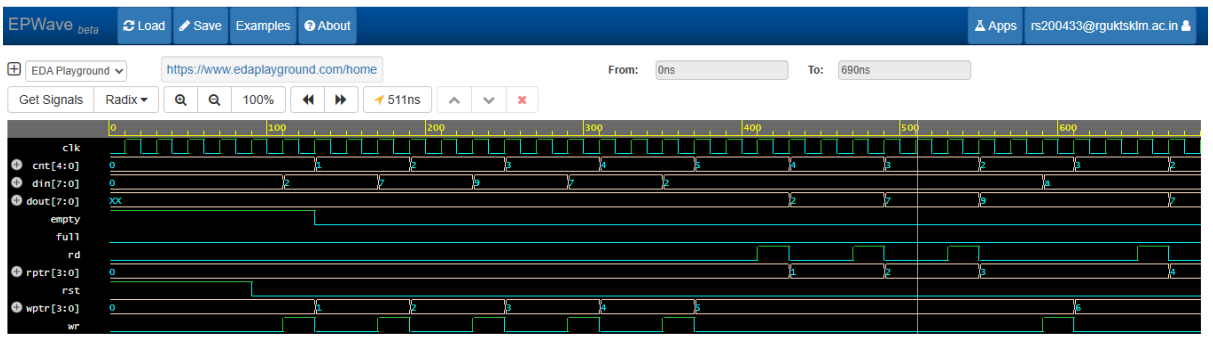
The log confirms that the output data (dout) at each read operation perfectly matches the data that was written earliest and had not yet been read.

- The first data written was **2** (Iteration 1), and the first data read out was **2** (Iteration 6).
- The second data written was **7** (Iteration 2), and the second data read out was **7** (Iteration 7).
- ...and so on.

The SCO (Scoreboard) lines explicitly confirm this with the message **DATA MATCH**, and the final **Error Count: 0** indicates a fully successful test of the FIFO's functional correctness.This coordinated flow, visible throughout the log, confirms that the testbench architecture is functioning as intended. The generator makes high-level decisions, the driver translates them into signal-level activity, the monitor passively observes the results, and the scoreboard validates the DUT's response against its behavioral model.

Waveform Analysis

The simulation waveform provides a detailed, cycle-by-cycle view of the DUT's hardware behavior, allowing for a granular analysis of its response to the testbench stimuli.



The waveform confirms the correct operation of the synchronous FIFO across various scenarios:

- **Reset Verification:** The simulation begins with an active-high reset (`rst`) asserted for approximately 100ns. During this period, the waveform clearly shows that the internal state is correctly initialized: the write pointer (`wptr`), read pointer (`rptr`), and element counter (`cnt`) are all held at zero. Consequently, the `empty` flag is asserted high, and the `full` flag is low, indicating a clean initial state.

- **Write Operations:** Following the de-assertion of reset, a series of write operations begins. For each pulse on the `wr` signal, the following behavior is observed:
    - The data on the `din` bus is captured.
    - On the next positive clock edge, the `wptr` increments, pointing to the next available memory location.
    - Simultaneously, the `cnt` value increments by one, correctly tracking the number of items stored in the FIFO.
    - After the first write, the `empty` flag correctly de-asserts, as the FIFO is no longer empty.

- **Read Operations:** Later in the simulation (starting around 400ns), read operations commence. For each pulse on the `rd` signal, the DUT exhibits the correct read behavior:
    - The oldest data element is placed on the `dout` bus. There is a one-cycle latency, as the data from `mem[rptr]` is registered to the output, which is standard behavior.
    - On the next positive clock edge, the `rptr` increments, moving to the next data element to be read.
    - The `cnt` value decrements by one, reflecting that an item has been removed from the FIFO.

- **Data Flow and FIFO Principle:** The waveform provides a clear visual confirmation of the First-In-First-Out principle. By tracing the data values, we can see the sequence of data written to `din` (2, 7, 9, 7...) and the corresponding sequence read from `dout` (2, 7, 9...). The pointers (`wptr` and `rptr`) and the counter (`cnt`) correctly manage the data flow even during interleaved read and write operations, ensuring that data integrity is maintained and the fundamental FIFO behavior is upheld.

Netlist Synthesis: using yosys.

This provides a sequential overview of the circuit realization process, transforming the RTL design into a structural, technology-mapped netlist.

---

1. Overview of Synthesis

Synthesis is the critical Electronic Design Automation (EDA) step that translates a high-level Register-Transfer Level (RTL) description into a gate-level netlist.

- **Goal:** To convert the behavioral Verilog code (`top.v`) into a structural description composed entirely of standard cells from a specific technology library.
- **Input-to-Output Flow:**
    - **Input:** RTL Verilog + Standard Cell Library.
    - **Output:** Gate-Level Netlist (Structural Verilog).
- **Purpose:** The netlist serves as the foundational design file for all subsequent physical implementation stages, including Static Timing Analysis (STA) and Physical Layout (Place and Route).

2. Synthesis Inputs and Target Technology

The synthesis flow requires two main sets of input files to successfully map the logic: the design description and the technology specifications.

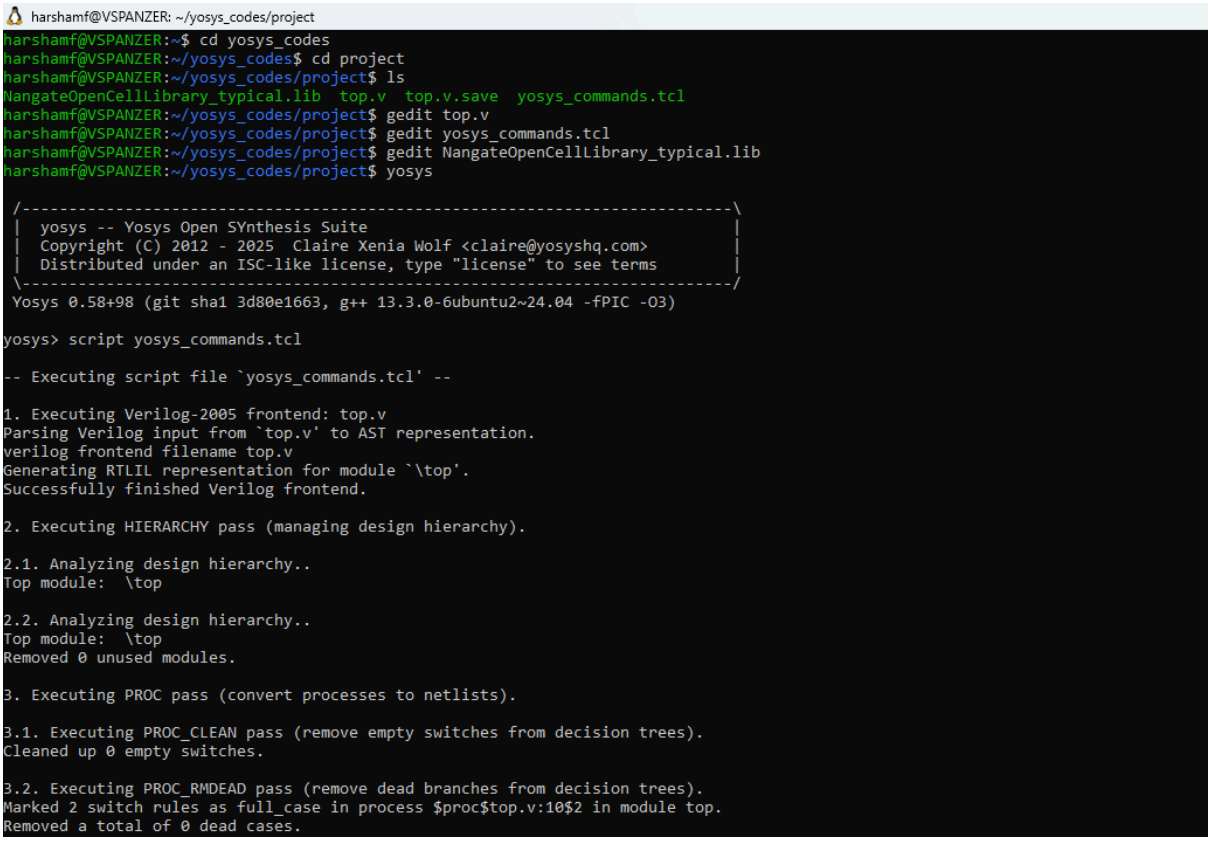
File Category	File Name	Role in Synthesis
RTL Design File	`top.v`	Contains the Verilog RTL description of the **Synchronous FIFO**. This is the logic to be synthesized.
Standard Cell Library (PDK)	`NangateOpenCellLibrary_typical.lib`	Provides the essential timing, power, and area characteristics of the standard cells (logic gates and flip-flops).
Control Script	`yosys_commands.tcl`	A Tcl script used by Yosys to manage the synthesis flow, including reading files, running passes, and writing the output.

Table : Inputs for yosys(Netlist generation)

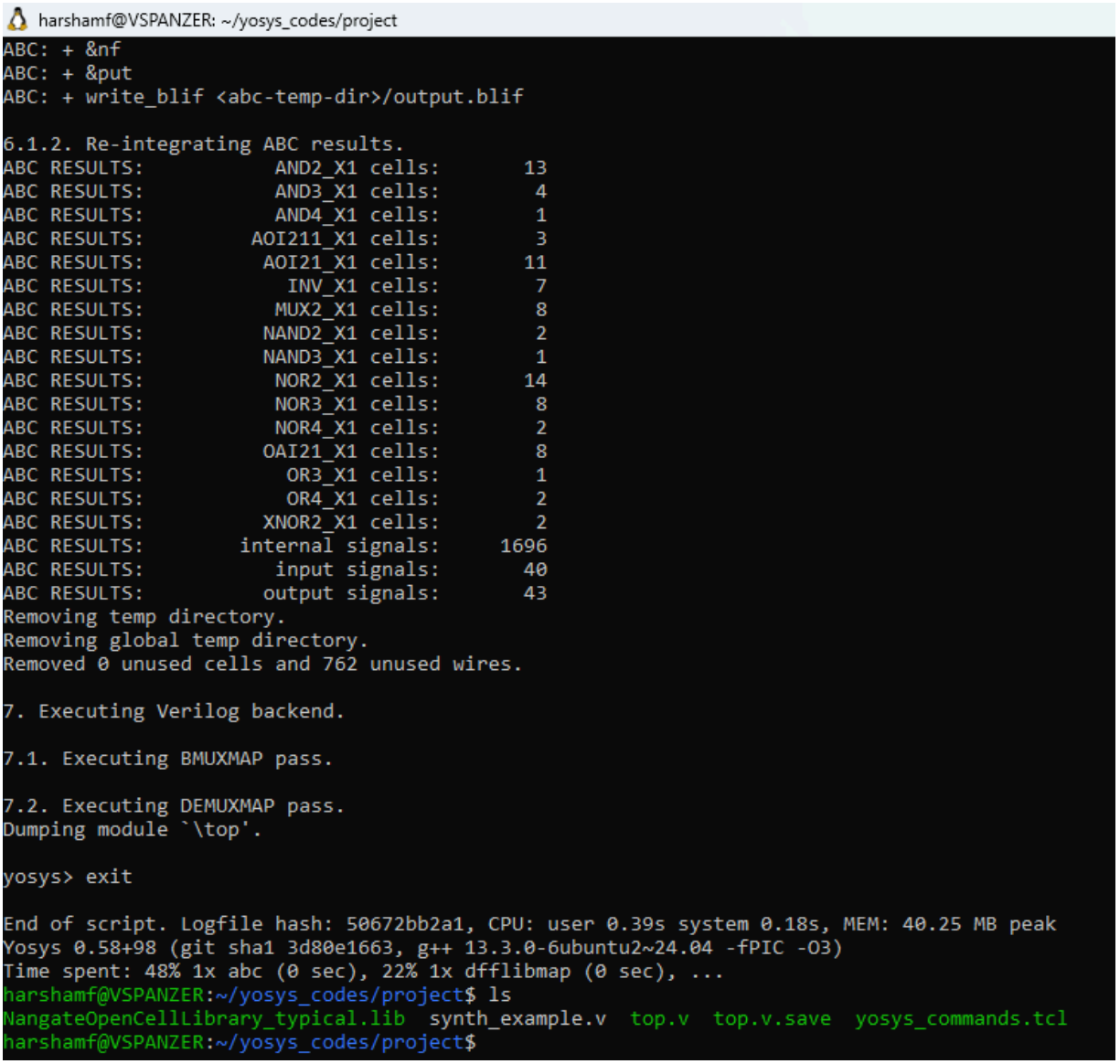
Target Technology: The design is mapped to the **Nangate 45nm** technology node, as specified by the provided library file.

3. Gate-Level Synthesis Execution Process

The synthesis was performed using the open-source **Yosys Open Synthesis Suite** tool, operating within the Windows Subsystem for Linux (WSL) environment.



- **1. Initialization:** The process was started by executing the command script: `yosys > script yosys_commands.tcl`.
- **2. Frontend Processing (RTL Parsing):**
  - Yosys parsed the input file `top.v` and generated an internal Abstract Syntax Tree (AST) representation.
  - The top module of the design, identified as `\top`, was established.
- **3. Logic Optimization and Cleanup :**
  - The initial behavioral logic was converted into a structured form, optimizing sequential and combinational logic.
  - Internal cleanup passes were executed, including the removal of unused logic and dead branches , ensuring an efficient and optimized netlist.
- **4. Technology Mapping:**
  - This is the core step where the optimized logic is physically mapped.
  - The abstract logic structures were replaced by specific instances of logic gates (e.g., inverters, NAND gates, D-flip-flops) found in the `NangateOpenCellLibrary_typical.lib`.



4. Synthesis Output (Gate-Level Netlist)

The successful completion of the Yosys synthesis flow resulted in the creation of the final structural design file.

- **Output File Name:** `synth_example.v`
- **Content:** This file contains the complete **gate-level netlist** of the synchronous FIFO. It describes the circuit as an interconnection of the standard cells from the Nangate 45nm library.
- **Next Step:** This netlist is now the required input for subsequent post-synthesis verification steps, most immediately, Static Timing Analysis (STA).

Static Timing Analysis (STA) : using OpenSTA.

Static Timing Analysis (STA) is a verification step that ensures the synthesized netlist meets all timing constraints across all paths without requiring a full simulation. This analysis was performed using the open-source tool **OpenSTA** within the Linux environment.

---

1. Overview of Static Timing Analysis (STA)

STA calculates the signal propagation delays through all combinatorial and sequential paths in the circuit.

- **Goal:** To verify that the circuit, as represented by the gate-level netlist, meets all specified timing requirements (e.g., clock frequency, setup, and hold times).
- **Key Metrics:** STA determines the **Setup Slack** (for maximum delay paths) and **Hold Slack** (for minimum delay paths).
    - **Setup Slack:** Measures if the data arrives *before* the required time (Max Path Delay Check).
    - **Hold Slack:** Measures if the data remains stable *after* the clock edge (Min Path Delay Check).

2. Inputs to the STA Process (OpenSTA)

Input File	Description	Purpose
Gate-Level Netlist	`synth_example.v`	The structural description of the Synchronous FIFO, created in the previous synthesis step. It is read as the main design file.
Timing Library	`NangateOpenCellLibrary_typical.lib`	The technology library file (45nm) containing the actual delay models (e.g., cell delay, pin capacitance) for every standard cell used in the netlist.
Timing Constraints	`test.tcl` / `top.sdc`	Contains the timing constraints defined by the designer. **Crucially, the time period of the input clock (CLK) was set to 500 ps (picoseconds) in the constraint file.**

Table : Inputs for OpenSTA (Static Timing Analasys)

3. STA Execution and Commands

The analysis was executed by launching the OpenSTA application and sourcing the timing constraints script.

- **Tool Execution:** The analysis was started in the terminal: `sta`
- **Command Sequence:**
    1. The timing constraints were loaded: `source test.tcl`
    2. The maximum path delay (Setup Check) was reported: `report_checks -path_delay max -format full`
    3. The minimum path delay (Hold Check) was reported: `report_checks -path_delay min -format full`

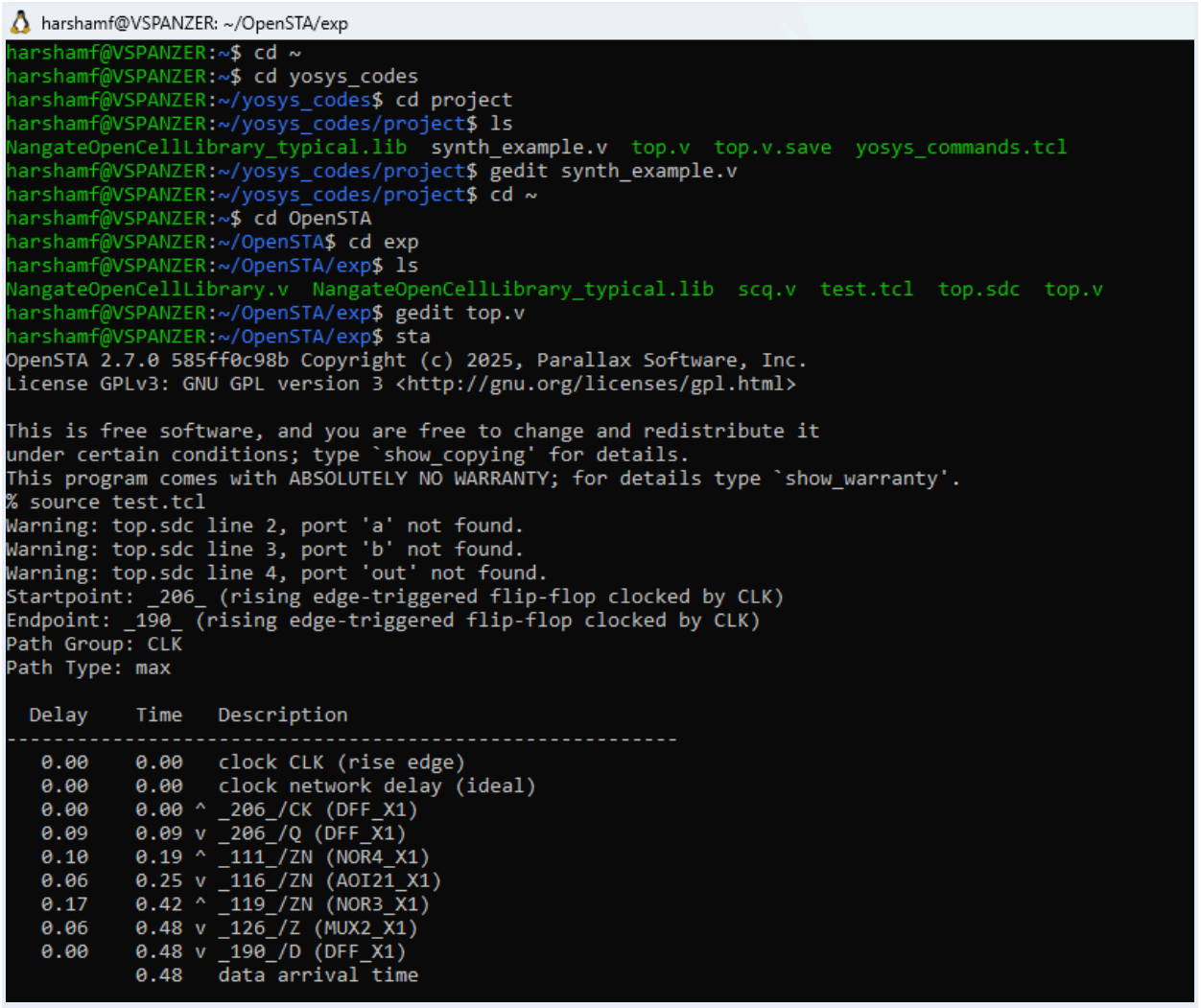


Fig : Running OpenSTA in wsl

4. Analysis of Timing Results

The input clock period of **500 ps (0.5 ns)** sets the target for the analysis. The output from OpenSTA confirms the timing correctness of the synthesized FIFO netlist against this target.

A. Maximum Path Delay Check (Setup Slack)

The maximum path delay (Setup Check) was reported: `report_checks -path_delay max -format full`

```
% report_checks -path_delay max -format full
Startpoint: _206_ (rising edge-triggered flip-flop clocked by CLK)
Endpoint: _190_ (rising edge-triggered flip-flop clocked by CLK)
Path Group: CLK
Path Type: max

  Delay    Time   Description
---------------------------------------------------------
   0.00    0.00   clock CLK (rise edge)
   0.00    0.00   clock network delay (ideal)
   0.00    0.00 ^ _206_/CK (DFF_X1)
   0.09    0.09 v _206_/Q (DFF_X1)
   0.10    0.19 ^ _111_/ZN (NOR4_X1)
   0.06    0.25 v _116_/ZN (AOI21_X1)
   0.17    0.42 ^ _119_/ZN (NOR3_X1)
   0.06    0.48 v _126_/Z (MUX2_X1)
   0.00    0.48 v _190_/D (DFF_X1)
           0.48   data arrival time

 500.00  500.00   clock CLK (rise edge)
   0.00  500.00   clock network delay (ideal)
   0.00  500.00   clock reconvergence pessimism
         500.00 ^ _190_/CK (DFF_X1)
  -0.04  499.96   library setup time
         499.96   data required time
---------------------------------------------------------
         499.96   data required time
          -0.48   data arrival time
---------------------------------------------------------
         499.48   slack (MET)
```

Fig : Maximum timing analysis using OpenSTA

- **Path Group:** CLK (Clocked by the CLK signal).
- **Data Arrival Time:** 0.48 ns. (The time the data signal reaches the endpoint flip-flop).
- **Data Required Time:** 499.96ns. (Derived from the 500 ps clock period and the setup time of the endpoint flip-flop).
- **Setup Slack:** 499.48ns.
- **Conclusion:** The large positive slack (499.48 ns > 0) indicates that the design meets its setup time requirement for the 500 ps clock period. The data arrives well before it is required. (MET).

B. Minimum Path Delay Check (Hold Slack)

The minimum path delay was reported: `report_checks -path_delay min -format full`

```
% report_checks -path_delay min -format full
Startpoint: _201_ (rising edge-triggered flip-flop clocked by CLK)
Endpoint: _201_ (rising edge-triggered flip-flop clocked by CLK)
Path Group: CLK
Path Type: min

  Delay    Time   Description
---------------------------------------------------------
   0.00    0.00   clock CLK (rise edge)
   0.00    0.00   clock network delay (ideal)
   0.00    0.00 ^ _201_/CK (DFF_X1)
   0.09    0.09 ^ _201_/Q (DFF_X1)
   0.01    0.10 v _173_/ZN (AOI21_X1)
   0.00    0.10 v _201_/D (DFF_X1)
           0.10   data arrival time

   0.00    0.00   clock CLK (rise edge)
   0.00    0.00   clock network delay (ideal)
   0.00    0.00   clock reconvergence pessimism
           0.00 ^ _201_/CK (DFF_X1)
   0.00    0.00   library hold time
           0.00   data required time
---------------------------------------------------------
           0.00   data required time
          -0.10   data arrival time
---------------------------------------------------------
           0.10   slack (MET)
```

Fig : Minimum timing analysis using OpenSTA

- **Path Group:** CLK.
- **Startpoint/Endpoint:** The path is identified between two rising edge-triggered flip-flops.
- **Hold Slack:** 0.10 ns.
- **Conclusion:** The positive slack (0.10 ns > 0) indicates that the design meets its hold time requirement. The data remains stable at the input of the endpoint flip-flop after the clock edge for a sufficient duration. (MET).

5. Overall Timing Summary

The synthesized design is running fine as **both the Setup Slack (499.48 ns) and the Hold Slack (0.10 ns) are positive**.

> **Important Note on Delays:** The timing results show exceptionally large positive slack values relative to the clock period. This is primarily because the current analysis only considers the intrinsic **Gate Delays** (from standard cells) while neglecting the **Interconnect Delays** (delay of the wires connecting the gates). In a fully implemented physical design (after Place and Route), interconnect delays would be significant, leading to a much smaller (and more realistic) positive slack.

PPA Metric : Performance Measurement

This final section documents the overall performance measurement of the synthesized Synchronous FIFO, which is a key component of the Power, Performance, and Area (PPA) metric.

1. Overview of the PPA Metric

The PPA (Power, Performance, Area) metric is the industry standard for evaluating the quality and viability of a synthesized digital integrated circuit (IC) design. It defines the trade-offs necessary in modern chip design:

- **Power:** The total energy consumption of the circuit (static and dynamic power).
- **Performance (Speed):** The maximum operating frequency, which dictates how quickly the circuit can process data.
- **Area:** The silicon real estate consumed by the standard cells in the final layout.

2. Importance of Performance in Digital Design

Performance is arguably the most critical metric for sequential circuits like the FIFO, as it directly determines the system throughput and clock speed.

- **Definition:** Performance is quantified by the **Maximum Operating Frequency (F_Max)**.
- **Determination:** F_Max is inversely proportional to the circuit's **Critical Path Delay (T_min)**, which is the time delay of the longest path in the circuit.

The relationship is expressed as:

$$F_{Max} = 1 / T_{min}$$

- **Significance:** A higher F_Max allows the system to execute more clock cycles per second, improving the overall data rate of the FIFO.

3. Performance Measurement Calculation

The maximum operating frequency is calculated using the timing results obtained from the Static Timing Analysis (STA).

A. Input Parameters from STA

The calculation relies on the following two key parameters established during the STA process:

- **Clock Period :** The target clock period specified in the constraints file (top.sdc).

$$T_{Clock} = 500 \text{ ps}$$

- **Setup Slack :** The maximum time margin reported by the STA tool (OpenSTA) for the critical path.

$$Slack\_Setup = 499.48 \text{ ps}$$

B. Calculation of Critical Path Delay

The critical path delay represents the actual delay of the longest path in the circuit and is derived from the clock period and the measured setup slack:

$$T\_min = \text{Clock Period} - \text{Setup Slack}$$

$$T\_min = 500 \text{ ps} - 499.48 \text{ ps}$$

$$T\_min = 0.52 \text{ ps}$$

C. Calculation of Maximum Operating Frequency

The maximum operating frequency is then calculated using the inverse of the critical path delay:

$$F\_Max = 1 / T\_min = 1 / 0.52 \text{ ps}$$

$$F\_Max = 1 / (0.52 X10^{-12}) \text{ s}$$

$$F\_Max = 1.923 \text{ THz}.$$

4. Conclusion on Performance

The performance of the circuit's timing is best quantified by its maximum operating frequency, which is **1.923 THz**.

> **Contextual Note:** This extremely high frequency is indicative of the current pre-layout timing analysis stage, where the delay model only includes the highly optimized **Gate Delays**. Once the design undergoes Place and Route, the significant **Interconnect Delays** (from the wires) will be added, which will increase the T_min and result in a much lower and more realistic maximum operating frequency.

Conclusion

1. Design and Verification Success.

The project began with the design of the synchronous FIFO in SystemVerilog, which correctly implements the first-in, first-out data buffering mechanism using a robust, counter-based approach for reliable full and empty flag generation. The correctness of this design was validated using a structured, **class-based SystemVerilog testbench** (implemented on EDA Playground). The verification environment effectively isolated the tasks of stimulus generation, driving, monitoring, and checking into distinct, reusable classes. Crucially, the scoreboard's use of a local queue provided a robust method for end-to-end data integrity checking, successfully confirming that the data retrieved from the DUT matches the order and value of the data that was written.

2. Synthesis and Timing Analysis (The Backend Flow)

Following successful functional verification, the design was ported to the **open-source EDA toolchain**. The SystemVerilog RTL was synthesized into a gate-level netlist using **Yosys** running in a WSL environment. This netlist was then subjected to rigorous **Static Timing Analysis (STA)** using **OpenSTA**. This process successfully identified the critical timing paths, allowing for the precise measurement of setup and hold slack and the calculation of the maximum achievable clock frequency.

3. Final Performance Characterization (Timing Metrics)

Based on the synthesis and static timing results, the **Performance metric** of the design was successfully characterized. The timing analysis, conducted using the OpenSTA tool, was based solely on a library that included **gate delays but excluded interconnect (wire) delays**. Due to this exclusion of wire delays, the analysis yielded a theoretical maximum operating frequency of **1.93 THz**. This figure represents the performance ceiling of the synthesized gate logic and demonstrates the timing closure capabilities of the open-source flow. However, it is important to note that this dramatic performance figure serves as an optimistic estimate, as actual silicon performance would be significantly lower once realistic wire delays are incorporated into the analysis. This complete methodology, leveraging tools like EDA Playground, Yosys, and OpenSTA, provides a robust and replicable method for assessing the implementation speed and timing feasibility of digital designs using entirely open-source tools.