**Journal – How I did it !!!!  
Part 1 - Ideation  
How to design PCie component using verilog step by step**

Designing a PCI Express (PCIe) component from scratch using Verilog is a complex task due to the high-speed, layered, and intricate nature of the PCIe protocol. The most common and practical approach is to leverage the hardened PCIe IP cores provided by FPGA vendors like Xilinx, Intel, and Lattice Semiconductor, and then design your custom logic (the "user logic") to interface with it.

This guide outlines the step-by-step process of designing user logic for a PCIe component that integrates with a vendor-provided IP core, which is the industry-standard method.

**1. Understand the PCIe protocol and architecture**

1. Before writing any code, you must understand the PCIe stack. Your Verilog user logic will primarily interface with the Transaction Layer, using a high-level bus like AXI, not the low-level physical layer.
2. Physical Layer: Manages the low-level electrical signaling, serialization/deserialization, and 128b/130b encoding.
3. Data Link Layer: Handles error checking, link management, and flow control with ACK/NACK protocols.
4. Transaction Layer: Creates and processes Transaction Layer Packets (TLPs), which include Memory Read/Write, I/O, and Configuration transactions. This is the layer your custom logic will interact with.

**2. Define your application and microarchitecture**

1. Determine what your PCIe component needs to do and how it will be structured.
2. Functionality: Will it perform Direct Memory Access (DMA) to and from system memory? Will it act as a simple register-based interface? A DMA engine is a common, complex use case.
3. Interfaces: Sketch out your user logic modules. A common structure includes:
   1. Configuration Logic: Handles requests to the PCIe Configuration Space.
   2. DMA Controller: Manages memory read and write transfers if DMA is needed.
   3. Application Logic: The core functionality that processes data.

**3. Generate the PCIe IP core**

1. Use the IP generation tool provided by your FPGA vendor to create a PCIe core. This process will create all the necessary low-level Verilog files and constraints.
2. Launch the Tool: Open your FPGA design suite (e.g., Xilinx Vivado, Intel Quartus, Lattice Diamond).
3. Select the Core: Find the "PCIe IP Core" or "Integrated Block for PCIe" and add it to your project.
4. Configure Parameters: Set up the core's parameters based on your requirements:
5. Lane Width: x1, x4, x8, or x16.
6. Link Speed: Gen1, Gen2, or Gen3.
7. Interface Type: AXI4 is the most common and robust interface for connecting your user logic to the PCIe core.
8. BARs (Base Address Registers): Configure the memory-mapped regions that the host will use to communicate with your device.

**4. Design the user logic using Verilog**

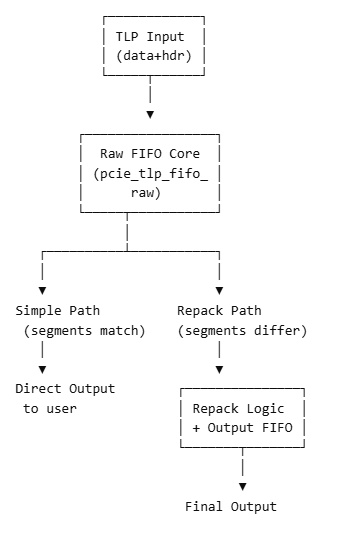
1. Write the Verilog code for your custom modules to connect to the AXI interface of the generated PCIe IP core.
2. User Logic Top Module: Instantiate your custom modules and the vendor's PCIe IP core. Connect the AXI signals between the two.
3. AXI Masters and Slaves: Depending on your application, you will design your user logic to act as either an AXI master (initiating memory transfers) or an AXI slave (responding to transfers).
4. Example: AXI-Lite Slave for register access
5. Design a simple module with an axil\_awaddr, axil\_wdata, and axil\_rdata interface.
6. Use a state machine to decode the incoming AXI address and read/write the internal registers. This is the foundation for a control and status register (CSR) block.
7. Example: AXI Master for DMA
8. Design a DMA engine that takes read/write requests from your application logic.
9. This logic will then generate AXI transactions (address, data, and control signals) to be sent through the PCIe core to the host's memory.

**5. Create a testbench for simulation**

1. A comprehensive testbench is crucial for verifying the correctness of your design before hardware synthesis.
2. Instantiate your design: The testbench will contain your top-level user logic module.
3. PCIe Bus Functional Model (BFM): The most effective way to test PCIe logic is with a PCIe BFM. You do not need to write this from scratch; open-source or vendor-provided BFMs can simulate a root complex and generate PCIe transactions.
4. Simulate Host Actions: Use the BFM to simulate the host performing the following actions:
5. Enumeration: Simulate the bus enumeration process.
6. Configuration Space Access: Read and write to your configured BARs.
7. Memory Access: Initiate memory read/write requests to test your data path.
8. Verification: Check that your user logic correctly processes the incoming transactions and produces the expected output.

**6. Synthesize and implement the design**

1. After successful simulation, move to the hardware implementation phase.
2. Add Files to Project: Include all your Verilog user logic files and the generated IP core files in your FPGA project.
3. Add Constraints: Apply the necessary constraints, including clock periods and pin assignments. The IP core generation should provide many of these automatically, but you may need to add constraints for your custom logic.
4. Synthesize and Place and Route: Run the synthesis, place, and route tools in your FPGA suite to convert your Verilog code into a hardware bitstream.
5. Timing Closure: Address any timing violations reported by the tools to ensure your design operates reliably at the required clock frequencies.

**Part 2: Step 1 – Understanding Architecture and designing basic Modules  
A) PCIe FIFO   
  
Concept: Why this module exists**

In PCIe, *Transaction Layer Packets (TLPs)* flow between blocks. But different parts of the design may operate at different rates or segment widths.  
So, we need a FIFO:

* Store incoming TLPs (with data, headers, sequence numbers, etc.)
* Output them when downstream logic is ready
* Handle segmentation: input TLPs may come in multiple segments, and output TLPs may require re-packing.

This module is a wrapper FIFO that:

* Uses pcie\_tlp\_fifo\_raw as the actual storage.
* Adds segment packing/unpacking logic when IN\_TLP\_SEG\_COUNT ≠ OUT\_TLP\_SEG\_COUNT.
* Provides status signals like half\_full and watermark.

**Structure (big picture)**

* Parameters: configure depth, widths, number of segments.
* I/O: TLP input (AXI-like interface), TLP output, status.
* Internal wires: connect to the underlying raw FIFO.
* Generate block:
  + If segment counts match → simple direct FIFO.
  + Else → complicated re-packing + an additional small output FIFO.

**Breaking down code step by step**

**Header & parameters**

`resetall

`timescale 1ns / 1ps

`default\_nettype none

* Reset compiler directives to default.
* Set simulation timescale (1ns steps, 1ps precision).
* Disallow implicit wire declaration.

module pcie\_tlp\_fifo #(

parameter DEPTH = 2048,

parameter TLP\_DATA\_WIDTH = 256,

parameter TLP\_STRB\_WIDTH = TLP\_DATA\_WIDTH/32,

parameter TLP\_HDR\_WIDTH = 128,

parameter SEQ\_NUM\_WIDTH = 6,

parameter IN\_TLP\_SEG\_COUNT = 1,

parameter OUT\_TLP\_SEG\_COUNT = IN\_TLP\_SEG\_COUNT,

parameter WATERMARK = DEPTH/2

)

* Configurable FIFO. For example: 256-bit data width, 128-bit header, up to 2048 entries.

**I/O definitions**

input wire clk, rst;

* Clock & reset.

TLP Input bus

in\_tlp\_data, in\_tlp\_strb, in\_tlp\_hdr, in\_tlp\_seq, ...

* These carry PCIe TLP data, byte-enable strobes, headers, sequence numbers, etc.
* in\_tlp\_valid/in\_tlp\_ready → handshake.
* in\_tlp\_sop/in\_tlp\_eop → start and end of packet markers.

TLP Output bus

out\_tlp\_data, out\_tlp\_strb, out\_tlp\_hdr, ...

* Same as input, but for reading out from the FIFO.
* Controlled by out\_tlp\_ready.

Status

output wire half\_full, watermark;

* Indicators for monitoring FIFO usage.

**Internal Parameters**

parameter INT\_TLP\_SEG\_COUNT = IN\_TLP\_SEG\_COUNT > OUT\_TLP\_SEG\_COUNT ? IN\_TLP\_SEG\_COUNT : OUT\_TLP\_SEG\_COUNT;

* Normalize segment count → ensures FIFO can handle whichever side has more segments.

Other parameters compute widths for strb/data per segment.

**Sanity checks**

initial begin

if (TLP\_HDR\_WIDTH != 128) ...

if (TLP\_STRB\_WIDTH\*32 != TLP\_DATA\_WIDTH) ...

end

* Makes sure configuration is PCIe-compliant.
* Header must be 128 bits, data must be aligned to 32-bit boundaries.

**FIFO instantiation**

pcie\_tlp\_fifo\_raw #(...params...)

pcie\_tlp\_fifo\_raw\_inst (...connections...);

* The *raw storage FIFO*.
* It stores everything (data, headers, sop/eop, etc.).
* Exposes read/write interfaces.

**Generate block**

generate

if (INT\_TLP\_SEG\_COUNT == 1) begin

// simpler case: pass-through

end else begin

// complex repacking logic

end

end generate

* If no segment mismatch → just connect raw FIFO to outputs.
* If mismatch → add extra logic.

**Concept Requirement :**

**PCIe / TLP essentials (what matters for the FIFO)**

These are the spec details / conventions you need for implementing a PCIe TLP FIFO:

* **TLP (Transaction Layer Packet)** = header + optional payload + CRC (CRC often removed before this FIFO). The FIFO here deals with header + data payload + control/meta (SOP/EOP/valid/ready).
* **DW granularity (32-bit)**: PCIe payloads are organized in 32-bit words (DWords). Byte strobe (strb) is typically payload\_width/32 bits per DWord lane. The code you started with enforces TLP\_STRB\_WIDTH \* 32 == TLP\_DATA\_WIDTH.
* **TLP header length**:
  + Typical TLP header sizes: **3 DW (12 bytes)** for 32-bit addressing and **4 DW (16 bytes)** for 64-bit addressing / 64-bit requester IDs etc.
  + Many implementations use **128 bits (16 bytes)** for header storage to keep alignment and to hold all metadata (hence TLP\_HDR\_WIDTH = 128 in your code). That’s a design choice — the PCIe spec allows 3 or 4 DW, but using 4 DW (128 bits) simplifies aligning to 128b lanes.
* **Start/End markers**: SOP (start of packet) and EOP (end of packet) flags let you pack/unpack variable-length payloads across segments.
* **Sequence numbers (SEQ)**: used by some blocks (ordering, replay). Your FIFO stores a SEQ\_NUM\_WIDTH per TLP segment.
* **Fields often stored with each segment**: header, sequence number, BAR ID, function number, error bits, valid/sop/eop, data, and data strobe.
* **Backpressure handshake**: valid + ready style handshake. The producer asserts valid; the consumer asserts ready; transfer happens when both valid && ready.
* **Segmentation**: Input may present multiple *segments* per cycle or segment count differs between in/out sides. The wrapper must repack segments so downstream sees expected segment count(s).  
    
  **line-by-line / sentence-by-sentence explanation**:
* `timescale 1ns / 1ps — sets the resolution/time unit for simulation (1 ns time unit, 1 ps precision).
* `default\_nettype none — prevents implicit wire declaration when you forget wire or reg; helps catch typos.
* module tlp\_fifo\_simple #(...) (...); — define a module named tlp\_fifo\_simple with parameters and ports.
* parameter DEPTH = 16 — number of FIFO entries; small for teaching.
* parameter ADDR\_WIDTH = 4 — address width to index DEPTH entries. For DEPTH=16, width=4.
* parameter TLP\_DATA\_WIDTH = 128 — width of payload data bus in bits.
* parameter TLP\_HDR\_WIDTH = 128 — width of header field. We pick 128 for alignment.
* input wire clk, rst — clock and synchronous reset inputs.
* Input side ports (in\_data, in\_hdr, in\_valid, in\_sop, in\_eop, in\_ready) — producer sends data when in\_valid asserted; in\_ready indicates FIFO can accept.
* Output side ports (out\_data, out\_hdr, out\_valid, out\_ready) — FIFO presents data when out\_valid high; consumer accepts when out\_ready high.
* reg [ADDR\_WIDTH:0] wr\_ptr = 0; — write pointer is ADDR\_WIDTH+1 bits. An extra MSB simplifies full/empty detection (a common circular buffer trick).
* reg [ADDR\_WIDTH:0] rd\_ptr = 0; — read pointer.
* reg [TLP\_DATA\_WIDTH-1:0] mem\_data [0:DEPTH-1]; — array of DEPTH registers holding payloads.
* reg [TLP\_HDR\_WIDTH-1:0] mem\_hdr [0:DEPTH-1]; — array holding headers.
* reg mem\_valid[0:DEPTH-1]; — optional per-entry valid flags for clarity (helps show entry state).
* wire full = (wr\_ptr MSB != rd\_ptr MSB) && (lower bits equal) — full detection: when pointers wrap and lower index matches but MSB different, FIFO is full.
* wire empty = (wr\_ptr == rd\_ptr) — FIFO empty when both pointers equal.
* assign in\_ready = !full; — FIFO can accept if not full.
* assign out\_valid = !empty; — FIFO has data if not empty.
* assign out\_data = mem\_data[rd\_ptr[ADDR\_WIDTH-1:0]]; — present head-of-queue data on output bus (combinational read).
* assign out\_hdr = mem\_hdr[rd\_ptr[ADDR\_WIDTH-1:0]]; — present head-of-queue header on output bus.
* always @(posedge clk) write block — synchronous write/initialization on reset.
* On reset: wr\_ptr <= 0; and clear mem\_valid[].
* Else: if in\_valid && in\_ready then write data/header into memory at write index, set valid bit, and increment wr\_ptr.
* always @(posedge clk) read block — synchronous read.
* On reset: rd\_ptr <= 0;
* Else: if out\_valid && out\_ready then consumer accepted the data; clear entry valid and increment rd\_ptr.