Examples where state-of-the art models like GPT-5 fails to spit out fully-functional RTL code from well-defined specifications.

Example 1: Problem Statement: 2-Read 1-Write (2W1R) Register File

Implement a programmable bitwidth, 32-word 2-read 1-write (2R1W) register file (RF). The RF has 32 entries, each of which being a DATA\_WIDTH-bit binary word. The input word din is written to one of the entries of the RF by using the write address port wad1 and asserting signal wen1. Entries are read from the RF by selecting the addresses and asserting the ren1 and/or ren2 signals. The RF must support up to three operations per clock cycle, that is, two reads and one write; no operations (NOP), one, and two operations must be also supported. The default value of both dout1 and dout2 is zero. If, at the rising edge of the clock, one of the read-enable ports is deasserted, then it is expected that its respective data-output port produces the default value (zero). If one tries to read from an address that has never been written to, then dout1 and/or dout2 produce zero. RF's output port collision = 1 when at least two out of three input addresses are equal, and collision = 0 otherwise. That is, the RF must flag when one tries to write to and read from the same address (wad1 = rad1 or wad1 = rad2), or attempts to read from the same address using both read ports (rad1 or wad1 = rad2).

**Input and Output Signals**

* din - Input data port
* wad1 - Write input address 1
* rad1 - Read input address 1
* rad2 - Read input address 2
* wen1 - Write-enable signal 1
* ren1 - Read-enable input signal 1
* ren2 - Read-enable input signal 2
* clk - Clock signal
* resetn - Synchronous, active-low, reset signal
* dout1 - Output data port 1
* dout2 - Output data port 2
* collision - Indicates collisions among read & write addresses

**Output signals during reset**

* dout1 - 0
* dout2 - 0
* collision - 0
* And That's the code GPT gave me:
* verilog
* module model #(parameter
* DATA\_WIDTH = 16
* ) (
* input [DATA\_WIDTH-1:0] din,
* input [4:0] wad1,
* input [4:0] rad1, rad2,
* input wen1, ren1, ren2,
* input clk,
* input resetn,
* output logic [DATA\_WIDTH-1:0] dout1, dout2,
* output logic collision
* );
* // 32-word register file memory
* logic [DATA\_WIDTH-1:0] rf [0:31];
* integer i;
* // Combinational collision detection
* always\_comb begin
* if ((wad1 == rad1) || (wad1 == rad2) || (rad1 == rad2))
* collision = 1'b1;
* else
* collision = 1'b0;
* end
* // Sequential read/write and reset behavior
* always\_ff @(posedge clk or negedge resetn) begin
* if (!resetn) begin
* dout1 <= '0;
* dout2 <= '0;
* collision <= 1'b0;
* for (i = 0; i < 32; i = i + 1)
* rf[i] <= '0;
* end else begin
* // Write
* if (wen1)
* rf[wad1] <= din;
* // Read 1
* if (ren1)
* dout1 <= rf[rad1];
* else
* dout1 <= '0;
* // Read 2
* if (ren2)
* dout2 <= rf[rad2];
* else
* dout2 <= '0;
* end
* end
* Endmodule

But it failed tests,

A screenshot of a computer

AI-generated content may be incorrect.

Fig1 Correct Waveform

A green and white background with lines

AI-generated content may be incorrect.

Fig2 Wrong Waveform

My conclusions

ChatGPT failed to generate correct RTL code due to fundamental architectural limitations inherent in Large Language Models and specific hardware design misunderstandings.

1. Incorrect Collision Detection Logic

The Bug:​

```verilog

if ((wad1 == rad1) || (wad1 == rad2) || (rad1 == rad2))

collision = 1'b1;

```

The condition (rad1 == rad2) incorrectly flags a collision when two read operations target the same address. Reading the same register twice is perfectly valid in hardware and causes no structural hazard. True collisions only occur for write-read conflicts: when we're writing to an address while simultaneously reading from it. This reveals ChatGPT doesn't understand the physical hardware constraint: collisions matter only when there's contention for write/read on the same memory cell​

2. Combinational vs Sequential Logic Confusion

The Bug: Collision signal is assigned in both always\_comb and always\_ff blocks​

```verilog

always\_comb begin

if ((wad1 == rad1) || (wad1 == rad2) || (rad1 == rad2))

collision = 1'b1;

end

always\_ff @(posedge clk or negedge resetn) begin

if (!resetn) begin

collision <= 1'b0; // Also assigned here!

end

end

```

Combinational logic should not have reset logic​. Collision detection must be immediate (combinational), not registered (delayed by one clock cycle). This creates a fundamental timing error where collision is detected one cycle too late. ChatGPT conflated two distinct hardware design patterns without understanding their semantic differences​

3. Read Port Behavior Misunderstanding

The Bug:

```verilog

if (ren1)

dout1 <= rf[rad1];

else

dout1 <= '0;

```

Register files typically provide combinational reads or maintain previous values when read enable is low. Forcing outputs to zero when ren is low violates standard register file behavior​. This indicates ChatGPT learned syntactic patterns but not the functional semantics of hardware components

Example 2:

Problem Statement: PS/2 packet parser implementation

A screenshot of a computer screen

AI-generated content may be incorrect.

The code GPT gave me:

```verilog

module top\_module (

input clk,

input [7:0] in,

input reset, // Synchronous reset

output reg [23:0] out\_bytes,

output done

);

// Internal signals

reg [1:0] state, next\_state;

reg [23:0] next\_out\_bytes;

reg done\_reg;

reg [7:0] byte1, byte2, byte3;

// State encoding

localparam BYTE1 = 2'd0,

BYTE2 = 2'd1,

BYTE3 = 2'd2,

DONE = 2'd3;

// FSM sequential logic

always @(posedge clk) begin

if (reset) begin

state <= BYTE1;

byte1 <= 8'h00;

byte2 <= 8'h00;

byte3 <= 8'h00;

end else begin

state <= next\_state;

if (done)

{byte1, byte2, byte3} <= {in, byte1, byte2};

end

end

// FSM combinational logic

always @(\*) begin

next\_state = state;

done\_reg = 1'b0;

case (state)

BYTE1: if (in[3]) next\_state = BYTE2;

BYTE2: if (in[3]) next\_state = BYTE3;

BYTE3: if (in[3]) next\_state = DONE;

DONE: begin

done\_reg = 1'b1;

next\_state = BYTE1;

end

endcase

end

assign done = done\_reg;

assign out\_bytes = {byte1, byte2, byte3};

endmodule

Once again! Intended results didn't match:

A screenshot of a computer

AI-generated content may be incorrect.

* My Conclusions on why chatGPT failed to produce correct RTL code

The Bug:​

Verilog

case (state)

BYTE1: if (in[3]) next\_state = BYTE2;

BYTE2: if (in[3]) next\_state = BYTE3;

BYTE3: if (in[3]) next\_state = DONE;

No handling for in[3] == 0 cases - FSM doesn't specify what happens when the condition is false. In hardware, undefined state transitions lead to latches or unpredictable behavior​. The FSM should either stay in the current state or reset when receiving invalid data. ChatGPT generated incomplete state machines because it was predicting likely next tokens, not reasoning about all possible state transitions​

1. Race Condition in Done Signal

verilog

always\_ff @(posedge clk) begin

if (done)

{byte1, byte2, byte3} <= {in, byte1, byte2};

end

always @(\*) begin

case (state)

DONE: begin

done\_reg = 1'b1;

done is combinational (always @(\*)) but controls sequential logic in the same cycle. This creates a combinational loop and timing hazard​. The byte shifting happens in the DONE state after done is asserted, but it should happen during state transitions (BYTE1→BYTE2→BYTE3)

1. Next-Token Prediction vs. Formal Verification

LLMs operate on statistical pattern matching rather than formal reasoning:​ What ChatGPT Does: Predicts the most probable next token based on training data patterns. What RTL Requires: Formal guarantees about timing, state completeness, and hardware semantics​

1. No Understanding of Timing Constraints

* Hardware design requires satisfying timing constraints:​
* Setup and hold times.
* Clock-to-output delays.
* Propagation delays through combinational logic.
* Multi-cycle paths.

ChatGPT cannot:

1. Perform static timing analysis​
2. Ensure timing closure​
3. Understand physical hardware constraints​
4. Reason about race conditions and hazards