

Logs of chats used:

can you find if there is a yosys command to reverse the tech mapped netlist post synthesis into a synthesised netlist before abc or tech mapping?

No, there is generally **no Yosys command to "reverse" a tech-mapped netlist back to a pre-ABC or pre-tech mapping state.**

Here's why and what Yosys does:

- **Synthesis is a Lossy Process:** When you synthesize HDL code (Verilog, VHDL) into a netlist, and especially when you perform technology mapping, information is inherently lost. High-level constructs (like if statements, for loops, arithmetic operations) are transformed into lower-level gates (AND, OR, NOT, flip-flops, etc.) or specific technology library cells (LUTs, specific standard cells). This transformation is not easily reversible.
- **Technology Mapping is Specific:** Technology mapping specifically replaces generic gates with gates available in a target technology library (e.g., specific NAND, NOR, Inverter cells, or lookup tables (LUTs) for FPGAs). These mapped cells have specific characteristics (area, delay) that were part of the mapping process. Reversing this would require "un-mapping" to generic gates, which isn't a defined operation in the same way.
- **ABC (AIG-based Logic Synthesis System):** ABC is often used by Yosys for advanced logic optimization, typically before final technology mapping. ABC operates on And-Inverter Graphs (AIGs) and performs powerful logic restructuring. Once the design has gone through ABC, the original structure from your RTL is heavily optimized and transformed, making it practically impossible to reconstruct the exact pre-ABC netlist.

What Yosys does:

Yosys works by transforming the design through a series of "passes." Each pass performs a specific operation, moving the design closer to a final gate-level netlist. The general flow is:

1. **Frontend (e.g., read_verilog):** Parses your HDL code into Yosys's internal Register Transfer Level Intermediate Language (RTLIL). This is the closest you get to your original design description in a structured, machine-readable format.
2. **Elaboration (proc, flatten, etc.):** Resolves hierarchy, converts behavioral constructs into basic logic, and flattens the design if desired.

3. **Optimization (opt, techmap, abc):** This is where significant transformations happen:
- opt: Performs generic logic optimizations.
 - techmap: Maps abstract RTL cells to Yosys's internal gate library or to specific coarse-grain cells if a map file is provided.
 - abc: Performs advanced logic synthesis and optimization using the ABC tool, typically mapping to the internal gate library or directly to a technology library if a Liberty file is provided.
4. **Backend (write_verilog, write_blif, etc.):** Writes out the final netlist in a specified format.

In summary:

You can save the netlist at different stages of the synthesis process (e.g., after proc and flatten but before abc or techmap) to inspect the design at that level of abstraction. However, there's no "undo" command that intelligently reverses the complex transformations performed by abc or techmap to recover the exact original RTL structure or a generic pre-mapping netlist.

If you need to analyze the design at a less-transformed stage, you must run your Yosys script up to that point and then output the netlist.

how can I make you analyse the netlist better? can you identify buggy netlists if I give you a netlist and golden rtl

Unfortunately, as a language model, I **cannot directly "analyze" a netlist or RTL in the same way a specialized EDA tool can.** I don't have the capability to:

- **Parse and understand hardware description languages (HDLs) like Verilog or VHDL structurally.** I can process text, but I don't have an internal representation of gates, wires, registers, or their interconnections as an electrical circuit.
- **Perform formal verification or simulation.** I can't "run" the netlist or compare its behavior to the RTL.
- **Identify logical bugs directly.** While I might be able to spot syntax errors if you provide the code, I cannot understand the intended functionality and determine if the netlist deviates from it.

Therefore, I cannot directly identify buggy netlists by comparing them to golden RTL. My "analysis" capabilities are limited to text processing.

What I *can* help you with regarding netlists and RTL (and how to make me "analyze" better within my limitations):

Even though I can't act as a formal verification tool, I can still be helpful in other ways:

1. **Explaining Yosys Commands and Synthesis Concepts:** You can ask me about:
 - What a specific Yosys command does.
 - The purpose of different synthesis passes (e.g., flatten, opt, abc, techmap).
 - General principles of logic synthesis and technology mapping.
 - How to set up a basic Yosys flow.
2. **Troubleshooting Yosys Script Issues:** If you're getting errors from Yosys, you can provide:
 - Your Yosys script.
 - The error messages.
 - I might be able to identify syntax errors in your script or suggest common issues.
3. **Understanding Netlist Formats:** I can explain the structure of common netlist formats (Verilog, BLIF, etc.) and what different elements mean.
4. **Generating or Modifying Small HDL/Netlist Snippets (with caution):**
 - You could ask me to generate a very simple Verilog module for a specific function (e.g., a 2-input AND gate).
 - You could ask me to modify a *small* section of a netlist, but **extreme caution is advised here**. Manually modifying netlists is highly error-prone and generally not recommended for anything but the simplest debugging.
5. **Summarizing and Explaining Netlist/RTL Code (Textual Analysis):**
 - If you provide a *small* snippet of RTL or netlist, I can try to explain what the code *appears* to be doing based on keywords and structure. For example, "This looks like an always block sensitive to positive edge of clock, implying a sequential element."
 - I can help you understand variable declarations, module instantiations, and basic gate types if they are explicitly named.

How to "make me analyze better" (within my limitations):

- **Provide specific questions:** Instead of "analyze this netlist," ask "What does the always @(posedge clk) block typically represent in Verilog?" or "Explain the difference between assign and always."
- **Break down complex problems:** If you have a large netlist, don't just paste it. Ask about a specific module or a specific type of gate you're seeing.
- **Use precise terminology:** Use correct Verilog/VHDL/Yosys terms.
- **Provide context:** Explain what you're trying to achieve, what problem you're facing, and what you've tried so far.

For actual netlist verification, you *must* use industry-standard Electronic Design Automation (EDA) tools. These include:

- **Logic Simulators:** (e.g., QuestaSim, VCS, Riviera-PRO, Icarus Verilog) to simulate both your RTL and your synthesized netlist and compare their outputs.
- **Formal Equivalence Checkers (FEC):** (e.g., Cadence Conformal, Synopsys Formality, OneSpin) these are specialized tools designed to mathematically prove whether two designs (e.g., RTL and netlist) are logically equivalent. This is the gold standard for netlist verification.
- **Linting Tools:** (e.g., Ascent Lint, Verilator) to check for common coding errors and style guide violations in your RTL *before* synthesis.

So, while I can assist with understanding concepts and basic text-based explanations, I cannot replace the specialized tools required for robust netlist analysis and verification.

how does an always @(posedge clk) block et synthesisse

You're asking a fundamental question about how sequential logic is synthesized from Register Transfer Level (RTL) Verilog code. The always @(posedge clk) block is the primary way to describe synchronous sequential circuits (flip-flops and registers) in Verilog.

Let's break down how an always @(posedge clk) block gets synthesized:

1. What always @(posedge clk) Means to a Synthesizer:

When a synthesis tool (like Yosys, Synopsys Design Compiler, Cadence Genus, etc.) encounters an always @(posedge clk) block, it immediately understands that:

- **Synchronous Behavior:** The code within this block should only update its outputs on the rising edge (positive edge) of the clk signal.

- **Sequential Logic:** This implies the creation of memory elements (flip-flops or registers) to store the state of the variables assigned within this block.
- **Implied Latch Prevention:** By default, if all possible paths assign a value to a variable in an always @(posedge clk) block, it will infer a flip-flop. If there are paths where a variable might *not* be assigned a value, it could infer a latch, but in *synchronous* always @(posedge clk) blocks, this is typically avoided by ensuring all assignments are made.

2. Key Elements within the always @(posedge clk) Block and their Synthesis:

- **Non-blocking Assignments (<=):** These are crucial for sequential logic. The synthesizer interprets <= as "evaluate the right-hand side (RHS) at the beginning of the clock cycle, and update the left-hand side (LHS) at the end of the clock cycle (or on the clock edge)." This behavior directly maps to the functionality of a flip-flop.
 - **Example:** q <= d; will synthesize into a D-type flip-flop where d is the data input and q is the data output.
- **Sensitivity List (posedge clk):** This tells the synthesizer the trigger event. Only posedge clk (or negedge clk) should be in the sensitivity list for synchronous sequential logic. Including other signals (like asynchronous resets) will also be interpreted correctly.
- **Conditional Statements (if/else, case):**
 - **Synchronous Reset (Asynchronous clear/preset):** If you have an if (reset) or if (!reset) condition *outside* the posedge clk condition but *inside* the always block, and the reset signal is also in the sensitivity list (always @(posedge clk or posedge reset) or always @(posedge clk or negedge reset)), the synthesizer will infer a flip-flop with an asynchronous reset/set input.

Verilog

```
always @(posedge clk or posedge rst_n) begin

    if (~rst_n) begin // Asynchronous active-low reset

        q <= 1'b0;

    end else begin

        q <= d;

    end

end
```

end

This would synthesize into a D-flip-flop with an active-low asynchronous reset input.

- **Synchronous Reset (Synchronous clear/preset):** If the reset condition is *inside* the if (posedge clk) block, it will infer a flip-flop with a synchronous reset.

Verilog

```
always @(posedge clk) begin
    if (rst) begin // Synchronous active-high reset
        q <= 1'b0;
    end else begin
        q <= d;
    end
end
end
```

This would synthesize into a D-flip-flop where the reset logic is part of the combinational logic feeding the D input.

- **Enabling Logic:** if or case statements can also infer enable pins on flip-flops if supported by the target technology library, or they can synthesize into combinational logic that feeds the D-input of the flip-flop.

Verilog

```
always @(posedge clk) begin
    if (enable) begin
        q <= d_in;
    end
end
end
```

This would synthesize into a D-flip-flop with an enable input. If no dedicated enable is available in the library, it would be $q_next = enable ? d_in : q$; where q_next feeds the D-input of the flip-flop.

3. Synthesis Process (Simplified Steps):

Let's use a simple example:

Verilog

```
module flip_flop (  
    input wire clk,  
    input wire d,  
    output reg q  
);  
  
always @(posedge clk) begin  
    q <= d;  
end  
  
endmodule
```

Here's how a synthesizer processes it:

- **Parsing and Internal Representation (RTLIL):** The synthesizer reads the Verilog code and converts it into an internal data structure (like Yosys's RTLIL). This representation captures the module, ports, and the always block's intent. It recognizes posedge clk as the clocking event and $q \leq d$ as a sequential assignment.
- **Inference of Flip-Flops:** Based on the posedge clk sensitivity list and the non-blocking assignment (\leq), the synthesizer infers that q needs to be a D-type flip-flop.
- **Technology Mapping:**
 - **Generic Gates:** Initially, the synthesizer might represent this as a generic `$_DFF_` cell or similar internal representation of a D-flip-flop.
 - **Library Mapping:** The synthesizer then looks at its target technology library (e.g., a standard cell library for ASIC, or the specific flip-flop primitives for an FPGA like FDPE, FDRE, FDCE in Xilinx, or DFFE, DFFCE in Intel/Altera). It finds the most suitable flip-flop cell that matches the inferred behavior (e.g., a

simple D-flip-flop without reset/set if none are present, or one with the appropriate reset/set if specified).

- **Optimization:** The synthesizer might perform optimizations, although for a single flip-flop, there's not much to optimize. For larger designs, it might retime logic, push/pull registers, or optimize the combinational logic feeding the flip-flop inputs.
- **Netlist Generation:** Finally, the synthesizer generates a netlist (e.g., in Verilog gate-level, EDIF, BLIF) that instantiates the chosen technology-specific flip-flop cell.

Example of Synthesized Netlist (Conceptual, simplified Verilog Gate-Level):

Verilog

```
module flip_flop (  
    input clk,  
    input d,  
    output q  
);
```

// Assuming a generic D-flip-flop cell from a standard cell library

// The exact instantiation name and port names will depend on the library

```
DFF_cell U1 (  
    .CLK (clk),  
    .D (d),  
    .Q (q)  
);
```

endmodule

Or, if targeting an FPGA like Xilinx:

Verilog

```
module flip_flop (  
    input clk,
```



```

input clk,
input d,
output q
);

// Xilinx 7-series D-type flip-flop with clock enable and reset/set inputs.

// Unused inputs (CE, R, S) are tied off to their inactive states (e.g., CE=1'b1, R=1'b0,
// S=1'b0)

// by the tool if not specified in the RTL.

FDRE #(
    .INIT(1'b0) // Initial value after configuration
) U1 (
    .C (clk), // Clock
    .CE (1'b1), // Clock Enable (tied high since not used in RTL)
    .R (1'b0), // Synchronous Reset (tied low)
    .D (d), // Data Input
    .Q (q) // Data Output
);

endmodule

```

In essence, the always @(posedge clk) block is the high-level instruction that tells the synthesis tool: "I need a piece of hardware here that stores a value and updates it synchronously with the clock." The synthesizer then translates this intent into the most appropriate physical memory element available in the target technology.

can you give me commands to create a graphical image of a synthesised netlist,

Yosys provides an excellent built-in command called show that leverages the **Graphviz** tool to create graphical representations of your netlist.

To use yosys show, you'll need:

1. **Yosys:** Of course, installed on your system.
2. **Graphviz:** The open-source graph visualization software. This is crucial as yosys show generates a .dot file (Graphviz's language) and then calls Graphviz's dot command to convert it into an image format.
3. **A Graphviz Viewer (Optional but Recommended):** Tools like xdot (for Linux) are good for interactive viewing of .dot files.

Here's how to do it:

Basic Steps:

1. **Synthesize your RTL code in Yosys.**
2. **Use the show command.**

Example Yosys Script (synth.js):

Let's assume you have a Verilog file named my_design.v:

Verilog

```
// my_design.v
```

```
module my_design (  
    input clk,  
    input rst,  
    input [3:0] data_in,  
    output reg [3:0] data_out  
);
```

```
always @(posedge clk or posedge rst) begin
```

```
    if (rst) begin
```

```
        data_out <= 4'b0000;
```

```
    end else begin
```

```
        data_out <= data_in + 1;
```

```
end
```

```
end
```

```
endmodule
```

Now, create a Yosys synthesis script (synth.ys) to process this:

Code snippet

```
# Read the Verilog source file
```

```
read_verilog my_design.v
```

```
# Elaborate the design (build hierarchy, resolve parameters, etc.)
```

```
hierarchy -top my_design
```

```
# Perform synthesis passes
```

```
# (This is a basic set, you might use more aggressive or specific passes depending on your target)
```

```
proc    # Convert always blocks to sequential/combinational logic
```

```
opt     # Perform generic optimizations
```

```
techmap # Map to generic Yosys cells (e.g., $dff, $add, $logic_and, etc.)
```

```
opt     # More optimization after techmap
```

```
# If you have a Liberty file (.lib) for specific ASIC cells, you'd use 'abc' here:
```

```
# read_liberty my_cells.lib
```

```
# abc -liberty my_cells.lib
```

```
# Display the netlist graphically
```

```
# This will generate a ~/.yosys_show.dot and ~/.yosys_show.svg file by default
```

```
# and open it with 'xdot' if available.
```

show

Running the Synthesis and Visualization:

Open your terminal and run Yosys with your script:

Bash

```
yosys synth.ys
```

Explanation of the show command:

The show command has various options to control the output:

- **show (default):**
 - Generates a .dot file (e.g., ~/.yosys_show.dot on Linux, show.dot on Windows).
 - Attempts to convert it to an SVG (e.g., ~/.yosys_show.svg).
 - Attempts to open the SVG (or .dot if SVG fails) with a default viewer (e.g., xdot on Linux).
- **show -format <format>:**
 - Specify the output image format. Common formats include svg, png, pdf, ps, dot.
 - Example: show -format svg -prefix my_netlist will create my_netlist.dot and my_netlist.svg.
- **show -viewer <viewer_command>:**
 - Specify a custom command to open the generated image. Use -viewer none if you just want the files generated without opening a viewer.
 - Example: show -format png -viewer eog (opens with Eye of GNOME image viewer).
- **show -prefix <filename_prefix>:**
 - Change the output file names from the default ~/.yosys_show to something like <filename_prefix>.dot and <filename_prefix>.<format>. This is highly recommended for organizing your output.
 - Example: show -prefix my_design_synthesized -format svg will create my_design_synthesized.dot and my_design_synthesized.svg.

- **show -notitle:**
 - Do not add the module name as a graph title.
- **show -width:**
 - Annotate buses with their width. Very helpful for understanding bus connections.
- **show -colors <seed>:**
 - Assign random colors to wires based on a seed. Useful for visually distinguishing different nets, especially in complex designs. 0 disables coloring.
- **show -long / -enum:**
 - Control how internal Yosys names (e.g., `$_AND_`) are displayed. -long shows full names, -enum enumerates them.

Example for a Specific Output File and Viewer:

To generate a PNG image named `synth_netlist.png` and open it with `xdg-open` (a common command on Linux to open files with their default application):

Code snippet

```
read_verilog my_design.v
hierarchy -top my_design
proc; opt; techmap; opt
show -prefix synth_netlist -format png -viewer xdg-open
```

Tips for Better Visualization:

- **Control the synthesis depth:** If your design is very large, the full gate-level netlist can be overwhelming. You might want to visualize the netlist at an earlier stage (e.g., after `proc` but before `techmap` or `abc`) to see a more abstract view with higher-level operators like adders and multiplexers.
- **Select specific modules/cells:** Yosys's `select` command can be used before `show` to only visualize a portion of your design. This is crucial for large designs.
 - Example: `select my_module` then `show` to see only `my_module`.

- **Filter out global signals:** Clocks and resets often clutter the graph. You can exclude them using the `-x` option:
 - `show -x {clk rst}`
- **Use netlistsvg for a different style:** While yosys show uses Graphviz for a graph-like representation, another tool called netlistsvg (often used with Yosys's `write_json` command) can generate more traditional digital circuit-style schematics. This requires an extra step:

Code snippet

In Yosys:

```
read_verilog my_design.v
hierarchy -top my_design
proc; opt; techmap; opt
write_json my_netlist.json
```

Then, you'd use the netlistsvg tool (which is typically a Node.js application or a web-based viewer) to convert `my_netlist.json` into an SVG schematic. This gives a very clean, schematic-like output.

The yosys show command with Graphviz is a powerful and easy way to visually inspect your synthesized netlists, aiding in debugging and understanding the synthesis process.

I want to install xdot

xdot is an interactive viewer for graphs in the Graphviz dot language. It's a Python-based tool and relies on Graphviz itself, along with Python and GTK+ bindings.

Here's how to install xdot on different operating systems:

Prerequisites for all OS:

- **Python 3:** xdot requires Python 3.
- **Graphviz:** You **must** have Graphviz installed, as xdot uses its underlying rendering capabilities. Install this first if you haven't already.

Linux (Debian/Ubuntu based systems)

This is typically the easiest on Linux.

1. Install Graphviz:

Bash

```
sudo apt update
```

```
sudo apt install graphviz
```

2. Install xdot and its Python dependencies:

Bash

```
sudo apt install gir1.2-gtk-3.0 python3-gi python3-gi-cairo python3-numpy xdot
```

- gir1.2-gtk-3.0: GTK3 bindings for Python.
- python3-gi, python3-gi-cairo, python3-numpy: Python dependencies for xdot.
- xdot: The xdot package itself.

macOS

On macOS, Homebrew is the recommended package manager.

1. **Install Homebrew** (if you don't have it): Follow the instructions on the Homebrew website: `/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"`

2. Install Graphviz:

Bash

```
brew install graphviz
```

3. Install GTK+3 and its dependencies (for xdot):

Bash

```
brew install gtk+3 gobject-introspection
```

4. Install xdot using pip (Python's package installer):

Bash

```
pip3 install xdot
```

You might also consider pipx for isolated Python application installations, but pip3 is generally sufficient here. If you encounter issues, some users have reported needing to explicitly inject dependencies with pipx:

Bash

```
pipx install xdot
```

```
pipx inject xdot "PyGObject" # or a specific version like "PyGObject==3.42.2"
```

```
pipx inject xdot "numpy" # or a specific version like "numpy==1.23.4"
```

Windows

Installing xdot on Windows is a bit more involved due to the GTK+3 dependencies.

1. **Install Python for Windows:** Download the latest Python 3 installer from the official Python website (python.org) and run it. Make sure to check the option "Add Python to PATH" during installation.
2. **Install Graphviz for Windows:** Download and install the Graphviz installer (.msi file) from the official Graphviz website (graphviz.org/download/). Make sure to select the option to add Graphviz to your system PATH during installation.
3. **Install PyGObject bindings for GTK3:** This is the trickiest part. You need to install the GObject Introspection bindings for Python, which allow Python to interact with GTK3.
 - **Recommended Method:** Use pip to install PyGObject. You might need to install pycairo first.

Bash

```
pip install pycairo
```

```
pip install pygobject
```

If you encounter issues, you might need to download a pre-compiled installer for PyGObject from unofficial sources (e.g., PyGObject Windows installers, search for "pygobject windows wheels"). Make sure it matches your Python version (32-bit or 64-bit).

4. **Install NumPy:**

Bash


```
pip install numpy
```

5. Install xdot using pip:

Bash

```
pip install xdot
```

Troubleshooting on Windows:

- If xdot still fails, check the error messages carefully. It's often related to missing GTK3 libraries or incorrect PyGObject installation.
- Ensure your system's PATH environment variable correctly includes the Python scripts directory and the Graphviz bin directory.

After Installation (How to test):

Once installed, you can test xdot by running it directly from your terminal:

1. **Generate a simple .dot file:** Create a file named test.dot with the following content:

Code snippet

```
digraph G {  
  A -> B;  
  B -> C;  
  C -> A;  
}
```

2. **Open it with xdot:**

Bash

```
xdot test.dot
```

An xdot window should pop up displaying the simple graph. You can pan and zoom using your mouse.

If you encounter any specific errors during the installation process, please provide them, and I can try to help you debug!

how can I test it with a synthesized .v file?

Let's test xdot with a synthesized Verilog file. This involves using Yosys to generate the Graphviz .dot file from your synthesized netlist, and then opening that .dot file with xdot.

Step 1: Ensure you have Yosys and Graphviz installed. (As discussed, xdot relies on Graphviz, and Yosys needs Graphviz to generate the .dot file for show.)

Step 2: Prepare your Verilog RTL file. Let's use a simple example, say my_counter.v:

Verilog

```
// my_counter.v
```

```
module my_counter (  
    input clk,  
    input rst,  
    input en,  
    output reg [3:0] count  
);
```

```
always @(posedge clk or posedge rst) begin
```

```
    if (rst) begin
```

```
        count <= 4'b0000;
```

```
    end else if (en) begin
```

```
        count <= count + 1;
```

```
    end
```

```
end
```

```
endmodule
```

Step 3: Create a Yosys script to synthesize and generate the .dot file. Create a file named synth_and_show.py (or any .py extension):

Code snippet

```
# Read the Verilog source file
```

```
read_verilog my_counter.v
```

```
# Elaborate the design
```

```
hierarchy -top my_counter
```

```
# Perform synthesis passes
```

```
# (This is a minimal set to get a gate-level netlist)
```

```
proc    # Convert always blocks to sequential/combinational logic
```

```
opt     # Generic optimizations
```

```
techmap # Map to generic Yosys cells ($dff, $add, $logic_and, etc.)
```

```
opt     # More optimization
```

```
# Generate the .dot file.
```

```
# IMPORTANT: Use '-viewer none' so Yosys doesn't try to open a viewer itself,
```

```
#     as we want to open it manually with xdot.
```

```
#     '-prefix' gives a custom filename.
```

```
#     '-width' shows bus widths.
```

```
show -prefix my_counter_synthesized -format dot -viewer none -width
```

Step 4: Run Yosys to generate the .dot file. Open your terminal in the directory where you saved my_counter.v and synth_and_show.py, then run:

```
Bash
```

```
yosys synth_and_show.py
```

After Yosys finishes, you should find a file named my_counter_synthesized.dot in the same directory.

Step 5: Open the generated .dot file with xdot.

```
Bash
```

```
xdot my_counter_synthesized.dot
```

What to expect:

An xdot window should open, displaying a graphical representation of your synthesized my_counter netlist. You'll likely see:

- **Flip-flops:** Represented as boxes, typically named \$dff or similar, corresponding to your count register.
- **Adders:** A block for the count + 1 operation (often \$add).
- **Multiplexers/Logic Gates:** For the if/else if conditions and the reset logic.
- **Wires/Connections:** Lines connecting the different gates and flip-flops, showing the data flow.
- **Bus Widths:** Because we used -width, you should see numbers like [4] next to multi-bit wires.

You can use your mouse to pan, zoom in/out, and click on nodes to inspect them in xdot.

This process confirms that your Yosys setup can generate .dot files, and your xdot installation can correctly interpret and display them.

this is my synth file, can you tell why I'm getting a syntax error while doing xdot on it?

You've provided a Verilog file, not a Yosys synthesis script. The syntax error you're getting when doing xdot on this file is because **xdot expects a Graphviz DOT file (.dot extension), not a Verilog source file (.v extension).**

Your file is the **RTL (Register Transfer Level) design** of an AES cipher, written in Verilog. This file first needs to be synthesized by a tool like Yosys into a **netlist**, and then Yosys can generate a .dot file from that netlist.

Here's a breakdown of what's happening and how to fix it:

Why You're Getting a Syntax Error

1. **File Type Mismatch:** xdot is a viewer for Graphviz DOT files. Your synth file (which is actually aes_cipher_top.v) is a Verilog HDL file. xdot doesn't understand Verilog syntax, so it sees all the Verilog keywords (module, input, always, assign,

endmodule, etc.) as invalid commands in the DOT language, leading to a syntax error.

2. **Missing Synthesis Step:** The process to visualize your design involves two main stages:

- **Synthesis:** Use Yosys to convert your Verilog RTL (aes_cipher_top.v) into a gate-level netlist (often represented internally by Yosys or output as a Verilog netlist, BLIF, etc.).
- **Visualization:** Use Yosys's show command (which then calls Graphviz to generate the .dot file) to create the graphical representation of that synthesized netlist. Finally, xdot is used to view this generated .dot file.

How to Correctly Visualize Your AES Netlist

You need a **Yosys synthesis script** to process your aes_cipher_top.v file and generate the .dot file.

Here are the steps:

Step 1: Save your provided Verilog code. Save the Verilog code you provided as aes_cipher_top.v (or similar). Make sure all the .v files your design depends on (like aes_key_expand_128.v, aes_sbox.v, and timescale.v) are in the same directory or accessible via Yosys's search paths.

Step 2: Create a Yosys synthesis script. Create a new file, let's call it synth_aes.js, with the following content:

Code snippet

```
# Read all Verilog files required for your design.

# You might need to add paths for 'timescale.v', 'aes_key_expand_128.v', 'aes_sbox.v'
# if they are not in the same directory.

read_verilog aes_cipher_top.v

read_verilog aes_key_expand_128.v # Assuming this is the module for u0

read_verilog aes_sbox.v          # Assuming this is the module for us00-us33
```

```

# Set the top-level module
hierarchy -top aes_cipher_top

# Perform synthesis passes.
# For complex designs like AES, you'll need a more robust synthesis flow.
# This is a good starting point, but you might want to add 'flatten' or 'fsm' passes.
proc    # Convert always blocks to sequential/combinational logic
opt     # Perform generic optimizations
fsm     # Identify and optimize FSMs (if any)
techmap # Map to generic Yosys cells ($dff, $add, $logic_and, etc.)
opt     # More optimization after techmap
clean   # Remove unused wires/cells

# (Optional) If you want to see an even more optimized view with ABC:
# read_liberty -lib <your_tech_lib.lib> # if you have a specific ASIC library
# abc -liberty <your_tech_lib.lib>    # if using ABC with a library
# OR
# abc                                # for generic ABC optimization

# Generate the .dot file for visualization.
# -prefix: Names the output files (my_aes_netlist.dot, my_aes_netlist.svg/png)
# -format dot: Explicitly tells Yosys to only output the .dot file (good for manual xdot)
# -viewer none: Prevents Yosys from trying to open a viewer, so you can call xdot yourself.
# -width: Annotates bus widths, very helpful for understanding connections.
show -prefix my_aes_netlist -format dot -viewer none -width

```

Important Notes for AES Design:

- **Missing Submodules:** Your `aes_cipher_top.v` file instantiates `aes_key_expand_128` and `aes_sbox`. For Yosys to synthesize the full design, you **must provide the Verilog source files for these submodules** as well. Make sure `read_verilog` commands for these are included in `synth_aes.js`. If Yosys can't find them, it will give errors like "module not found."
- **timescale.v:** You have `\include "timescale.v"`. If this file simply defines a timescale directive, it's generally fine. Just ensure it's present.`
- **Synthesis Flow:** The AES design is complex. The basic synthesis flow (`proc; opt; techmap; opt; clean`) might work, but for a real-world scenario, you might want to explore more specific Yosys commands or the use of `abc` if you're targeting a specific technology library.

Step 3: Run Yosys. Open your terminal in the directory where you have `aes_cipher_top.v`, `aes_key_expand_128.v`, `aes_sbox.v` (and any other necessary submodules), and `synth_aes.js`. Then run:

Bash

```
yosys synth_aes.js
```

Yosys will run through the synthesis process. If it completes successfully, it will generate a file named `my_aes_netlist.dot` in the same directory. If there are errors, you'll see them in the terminal output. Pay close attention to "module not found" errors if you're missing submodules.

Step 4: Open the generated .dot file with xdot.

Bash

```
xdot my_aes_netlist.dot
```

This command should successfully open `xdot` and display the synthesized netlist of your AES top-level module. Given the complexity of AES, the graph will likely be very large and intricate! You'll need to zoom in and navigate to see the details.

Bugs Indicated by `mem.json`

The `mem.json` file, which represents the structure of your synthesized hardware design, shows numerous signals with "x" or unknown values. This is a clear indicator of bugs in the RTL code. Here are the most likely causes:

- **Uninitialized Memory or Registers:** The mem.json file shows memory blocks (\$mem_v2) with a read initialization value of "xxxxxxx". If the RTL code attempts to read from a memory location or a register before it has been written to, the output will be unknown ('x'), and this unknown value will propagate through the design.
- **Asynchronous Reset Issues:** The JSON shows attributes like "SRST_POLARITY": "00000000000000000000000000000000" and the use of \$sdffe (D-type flip-flop with synchronous set and asynchronous reset) cells. Incorrectly handled asynchronous resets are a common source of 'x' propagation in simulation, especially at the beginning of a simulation.
- **Combinational Logic Loops:** If there is a loop of logic gates without a register (a combinational loop), the logic can oscillate or have an undecidable state. A simulator will represent this state as 'x'.
- **Incomplete case or if-else Statements:** In Verilog, if a combinational always block does not specify an output for all possible input combinations, a latch is synthesized to hold the previous value. If this latch is not initialized, it can lead to 'x' values.
- **Signals Incorrectly Marked as nosync:** Several signals in mem.json have a "nosync" attribute. This attribute tells the synthesis tool not to add synchronizing flip-flops. If a signal that changes asynchronously is fed into synchronous logic without proper synchronization, it can cause metastability, which leads to 'x' values in simulation.

Bugs Indicated by clock_period.txt

The clock_period.txt file specifies a very aggressive clock period of **0.82 ns**, which corresponds to a clock frequency of approximately **1.22 GHz**. This high frequency is likely to introduce timing-related bugs:

- **Metastability:** At such a high clock frequency, it is very likely that some signals will not meet the setup and hold times of the flip-flops they are connected to. This is especially true for signals crossing from one clock domain to another or for signals with long combinational paths. When a flip-flop's timing is violated, it can enter a metastable state where its output is unpredictable for a short period, which is represented as an 'x' in simulation.
- **Race Conditions:** The fast clock speed increases the likelihood of race conditions, where the output of the circuit depends on the precise timing of different signals. Race conditions can lead to unpredictable behavior and 'x' propagation.

- **The Bug:**
- The most likely bug introduced into the golden RTL was rewriting the combinational `xtime` function to be sequential, likely by placing it inside a clocked `always` block (e.g., `always @(posedge clk)`).
- This change caused the synthesis tool to infer flip-flops that were not present in the original design. These flip-flops, being uninitialized at the start of the simulation, would output 'x' (unknown) values. These 'x' values then propagate through the circuit, causing the widespread errors observed in the `mem.json` file from your initial query.