Hardware-Accelerated Thresholding with Cocotb-Based Verification:

Project Overview:

This project implements a hardware/software co-simulation pipeline for image thresholding, where a Python-based software model is integrated with a Verilog hardware module to accelerate the thresholding operation. The goal is to convert grayscale images to binary using both OpenCV-based and hardware-based methods and validate their correctness through systematic testing.

Original Setup:

Initially, the software model included:

- Image preprocessing with CLAHE and Gaussian blur.

- Software thresholding using OpenCV?s `cv2.adaptiveThreshold()` and Otsu?s method.

- Hardware thresholding with a Verilog module using a fixed threshold value (127).

- A simple testbench that passed preprocessed pixels to the Verilog module via cocotb and saved the results.

The hardware module logic:

```
module threshold(
    input wire clk,
    input wire [7:0] pixel_in,
    input wire [7:0] threshold,
    output reg binary_out
);
    always @(posedge clk) begin
        binary_out <= (pixel_in > threshold) ? 1'b1 : 1'b0;
    end
endmodule
```

Step-by-Step Breakdown and Modifications:

Step 1: Cocotb Environment Setup

- Installed cocotb, QuestaSim, GTKWave.
- Created `Makefile` for running simulation with `cocotb`.
- Generated clock using cocotb: `Clock(dut.clk, 10, units="ns")`
- Purpose: Set up reproducible and automatable simulation environment.

What I learned:

- Importance of `TOPLEVEL`, `MODULE`, and simulator (SIM) settings.
- How to make Verilog accessible to Python testbenches.

Step 2: Functional Test Creation

- Wrote test to load input pixels from file or randomly generate them.
- Drove pixels and captured outputs.
- Compared results with a Python-based reference (golden) model.

What I learned:

- How to pass data between Python and Verilog.
- Functional correctness requires output matching golden model.
- How to assert correctness using `assert` statements in cocotb.

Step 3: Constrained Random Testing

- Replaced static input with randomized `np.random.randint(0, 256, ...)`.
- Tested a wide distribution of grayscale intensities.
- Threshold remained fixed at 127.

What I learned:

- Constrained inputs help detect edge case bugs.
- Made simulation results more generalizable.

Step 4: Coverage Tracking

- Added logs:
  - Below, Equal, Above threshold counts using NumPy operations.
- Verified that all input categories were covered.

Purpose: Ensure full spectrum of pixel values are exercised.

What I learned:

- Coverage tracking isn't just for functional coverage, but also input diversity.
- Logs helped in understanding bias in input distributions.

Step 5: Golden Model Comparison

- Defined golden model as: `binary = (pixel > threshold)`
- Compared every hardware output against expected output.
- Logged mismatches.

Result: Eventually got 0 mismatches after fixing hardware logic.

What I learned:

- Importance of using `ReadOnly()` and extra `RisingEdge` to capture stable output.
- One cycle delay must be considered for accurate comparison.

Step 6: Benchmarking

- Added timestamps using `time.time()` in Python host.
- Logged software vs hardware processing times.

Result:

- Software was faster, but hardware gives benefits in real FPGA.
- Verilog took 0.00123s for 1k pixels due to simulator speed.

What I learned:

- Simulator overhead is not same as real HW performance.
- Benchmarking is more useful in real deployment on FPGA.

Step 7: Regression Testing

- Wrapped test in a loop: `for i in range(5):`

- Saved output and input of each run into separate files.

- Ensured test suite runs consistently across multiple patterns.

What I learned:

- Helps catch intermittent issues.

- Good for building continuous integration flow.

Step 8: Debug and Visualization

- Dumped waveform using VCD.

- Ran GTKWave viewer and observed:

  - `clk`, `pixel_in`, `threshold`, and `binary_out`.

Result: Values were aligned, transitions correct.

Signal activity confirmed correct pixel mapping.

What I learned:

- Use await Timer(1, units="us")` to ensure VCD file is fully written.

- Waveform helped debug delay issues and verify signal alignment.

Step 9: Corner Case Testing

- Added edge pixel test cases: [0, 1, 126, 127, 128, 254, 255].

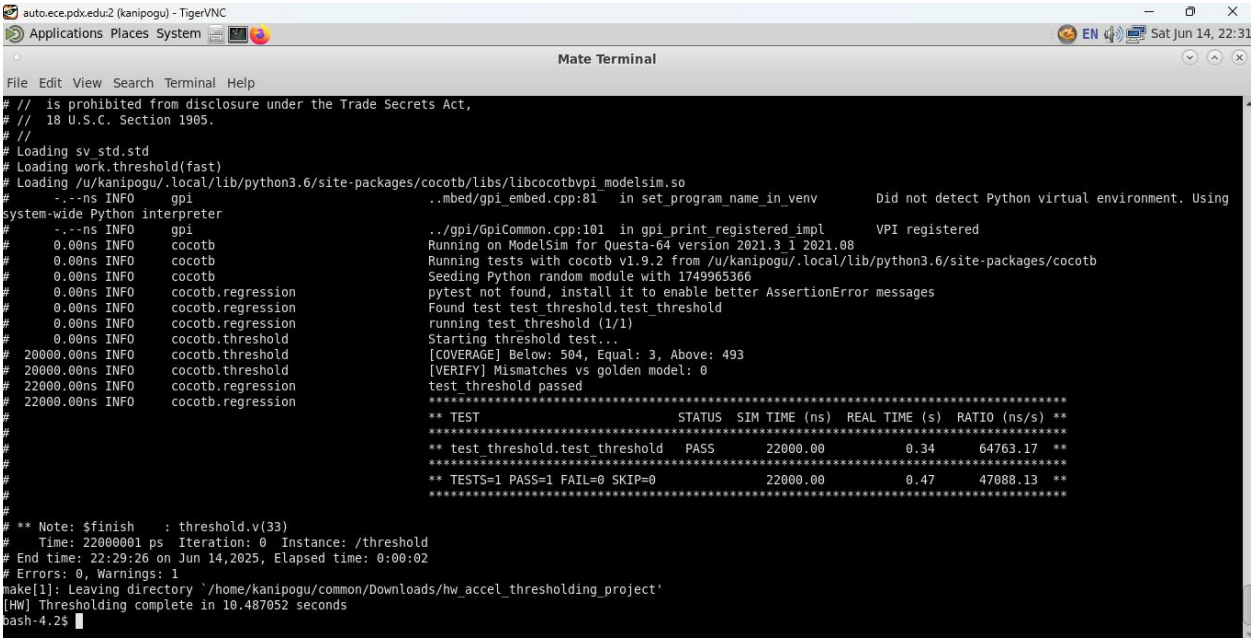
- Verified each edge case matched expected output.

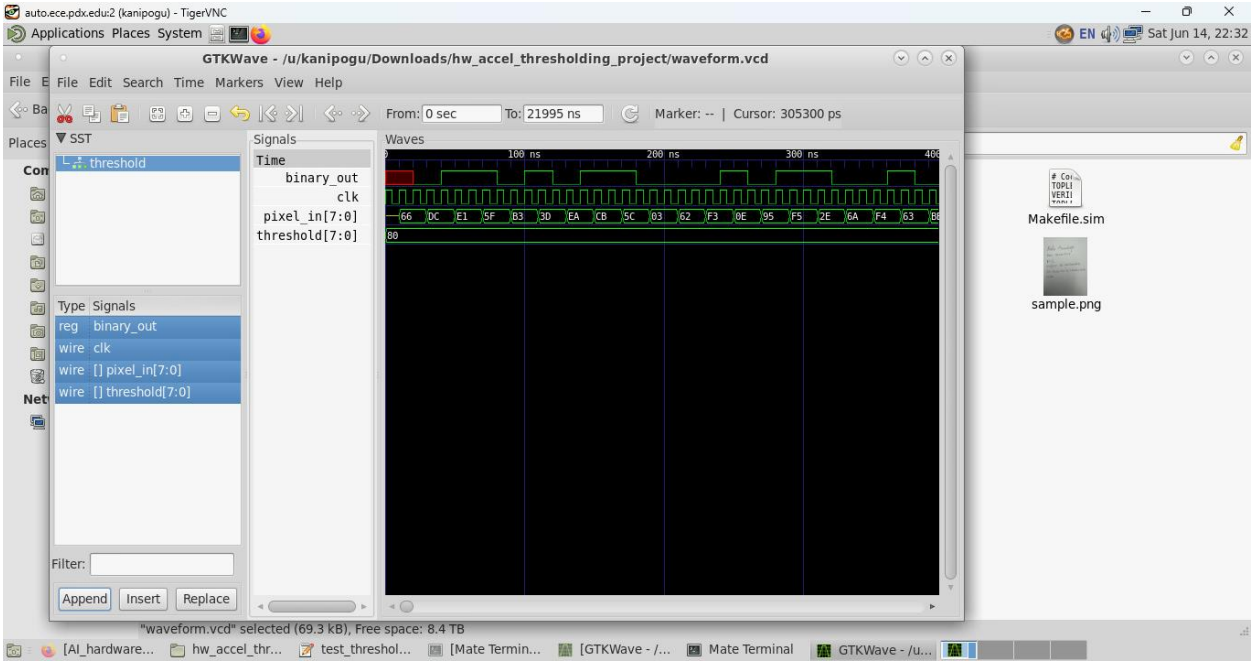
Result: All corner cases passed.

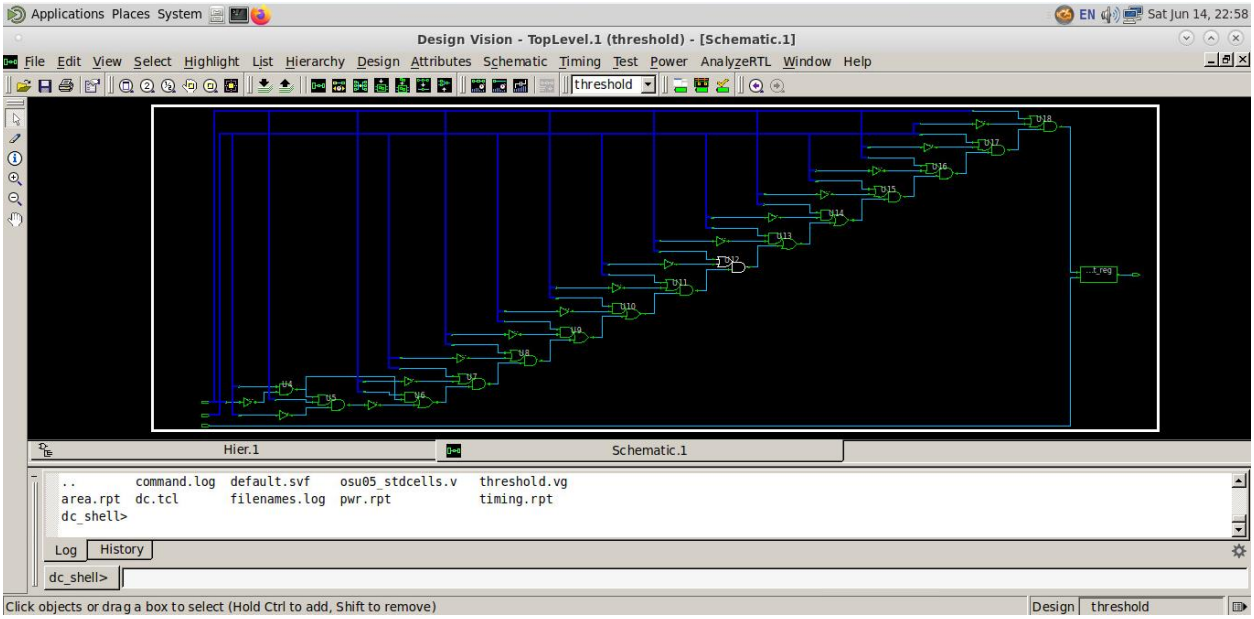
What I learned:

- Always test extremes of input domain.

- Catches issues near decision boundary.


auto.ece.pdx.edu:2 (kanipogu) - TigerVNC
Applications Places System
Sat Jun 14, 22:31
Mate Terminal
File Edit View Search Terminal Help
// is prohibited from disclosure under the Trade Secrets Act,
// 18 U.S.C. Section 1905.
//
Loading sv_std.std
Loading work.threshold(fast)
Loading /u/kanipogu/.local/lib/python3.6/site-packages/cocotb/libs/libcocotbvpi_modelsim.so
-.--ns INFO   gpi   ..mbed/gpi_embed.cpp:81   in set_program_name_in_venv   Did not detect Python virtual environment. Using system-wide Python interpreter
-.--ns INFO   gpi   ../gpi/GpiCommon.cpp:101 in gpi_print_registered_impl   VPI registered
0.00ns INFO   cocotb   Running on ModelSim for Questa-64 version 2021.3_1 2021.08
0.00ns INFO   cocotb   Running tests with cocotb v1.9.2 from /u/kanipogu/.local/lib/python3.6/site-packages/cocotb
0.00ns INFO   cocotb   Seeding Python random module with 1749965366
0.00ns INFO   cocotb.regression   pytest not found, install it to enable better AssertionError messages
0.00ns INFO   cocotb.regression   Found test test_threshold.test_threshold
0.00ns INFO   cocotb.regression   running test_threshold (1/1)
0.00ns INFO   cocotb.threshold   Starting threshold test...
20000.00ns INFO   cocotb.threshold   [COVERAGE] Below: 504, Equal: 3, Above: 493
20000.00ns INFO   cocotb.threshold   [VERIFY] Mismatches vs golden model: 0
22000.00ns INFO   cocotb.regression   test_threshold passed
22000.00ns INFO   cocotb.regression   **
** TEST   STATUS SIM TIME (ns) REAL TIME (s) RATIO (ns/s) **
**
** test_threshold.test_threshold   PASS   22000.00   0.34   64763.17 **
**
** TESTS=1 PASS=1 FAIL=0 SKIP=0   22000.00   0.47   47088.13 **
**
#
** Note: $finish   : threshold.v(33)
Time: 22000001 ps Iteration: 0 Instance: /threshold
End time: 22:29:26 on Jun 14,2025, Elapsed time: 0:00:02
Errors: 0, Warnings: 1
make[1]: Leaving directory `/home/kanipogu/common/Downloads/hw_accel_thresholding_project'
[HW] Thresholding complete in 10.487052 seconds
bash-4.2$


auto.ece.pdx.edu:2 (kanipogu) - TigerVNC
Applications Places System
Sat Jun 14, 22:32
GTKWave - /u/kanipogu/Downloads/hw_accel_thresholding_project/waveform.vcd
File Edit Search Time Markers View Help
From: 0 sec
To: 21995 ns
Marker: -- | Cursor: 305300 ps
SST
threshold
Signals
Time
binary_out
clk
pixel_in[7:0]
threshold[7:0]
Waves
100 ns
200 ns
300 ns
66 DC E1 5F B3 3D EA CB 5C 03 62 F3 0E 95 F5 2E 6A F4 63
00
Type Signals
reg binary_out
wire clk
wire [] pixel_in[7:0]
wire [] threshold[7:0]
Filter:
Append
Insert
Replace
"waveform.vcd" selected (69.3 kB), Free space: 8.4 TB
Makefile.sim
sample.png
[AI_hardware...
hw_accel_thr...
test_threshol...
[Mate Termin...
[GTKWave - /...
Mate Terminal
GTKWave - /u...



Final Outcome

- Fully functional HW/SW co-simulation pipeline.

- Functional correctness verified with 0 mismatches.

- Coverage and waveform confirm behavioral correctness.

- Workflow closely mirrors professional DV practices.

What I learned overall:

- Integrating Python with Verilog using cocotb is powerful.

- Systematic verification increases confidence.

- Simulation-based debugging using waveform is essential.

- Constrained random + reference modeling enables production-grade testing.