

Hardware-Accelerated Thresholding for OCR

Overview:

This project explores the implementation and verification of a hardware-accelerated thresholding block for an OCR (Optical Character Recognition) pipeline. The purpose is to offload the grayscale-to-binary conversion step traditionally executed in software to a Verilog hardware module simulated via Cocotb and QuestaSim.

Initially, the entire pipeline including image preprocessing, grayscale conversion, and thresholding was done in software. The final image output was then converted to a format that could be passed to a Verilog testbench for isolated hardware simulation. This approach worked but had major computational latency due to the handoff between full software processing and separate hardware execution.

Motivation and Improvement:

The main improvement involved identifying and accelerating the most time-consuming stage of the pipeline, the thresholding stage. This is the step where grayscale images are converted to binary format by comparing pixel values to a fixed threshold.

By isolating this stage and accelerating it with a hardware module called directly from the software pipeline, the project achieved significant improvement in execution time. This architecture offloads thresholding to hardware to improve the performance of the system.

Step-by-Step Progress Across Assignment Steps:

Step 1: Write a testbench using Cocotb:

I have created a test_threshold.py, a Python-based testbench using Cocotb:

Integrated OCR Hardware Acceleration Mode (via ocr_pipeline.py):

Here, ocr_pipeline.py is the top-level Python program that:

- Loads an input image and converts it to grayscale.
- Performs thresholding either:
 - in software, using NumPy (default),
 - or in hardware, by launching the Verilog module simulation through Cocotb+QuestaSim (--use_hw flag).

When --use_hw is passed:

- ocr_pipeline.py runs a make command, which triggers Cocotb to invoke test_threshold.py.
- Instead of using randomized values, test_threshold.py is now modified to accept pixel inputs from the image.
- The result is a hardware-accelerated binary image, saved and visualized after simulation.

What I have learned:

- Cocotb can be used not just for testbenches, but also to integrate hardware simulation inside software pipelines.

- This architecture enables hardware-software co-design, where the software offloads compute-heavy tasks (like thresholding) to the simulated RTL.
- By comparing software-only and HW-accelerated execution times, i could observe the performance benefits of offloading to RTL.

code:

if args.use_hw:

```
    print("[INFO] Using hardware-accelerated thresholding...")
    cv2.imwrite("results/debug_hw_input.png", img)
    hw_out = simulate_threshold_hw(img)
    cv2.imwrite("results/threshold_hw_output.png", hw_out * 255)
```

else:

```
    print("[INFO] Using software OpenCV thresholding...")
    sw_out = threshold_software(img)
    cv2.imwrite("results/final_combined_threshold.png", sw_out)
```

Step 2: Understand the simulation process with `make SIM=questa`

_ SIM=questa_

Makefile:

```
export WAVEFORM_FORMAT=VCD
SIM ?= questa
TOPLEVEL_LANG = verilog
VERILOG_SOURCES = ../threshold.v
TOPLEVEL = threshold
MODULE = test_threshold
include $(shell cocotb-config --makefiles)/Makefile.sim
```

What I have learned:

- make` flow with QuestaSim compiles and simulates the design via Cocotb.
- VCD waveform can be visualized using GTKWave.

Step 3: Generate and interpret waveform

The `threshold.v` module was updated with `\$dumpvars()` and `\$dumpfile()` to create `waveform.vcd`.

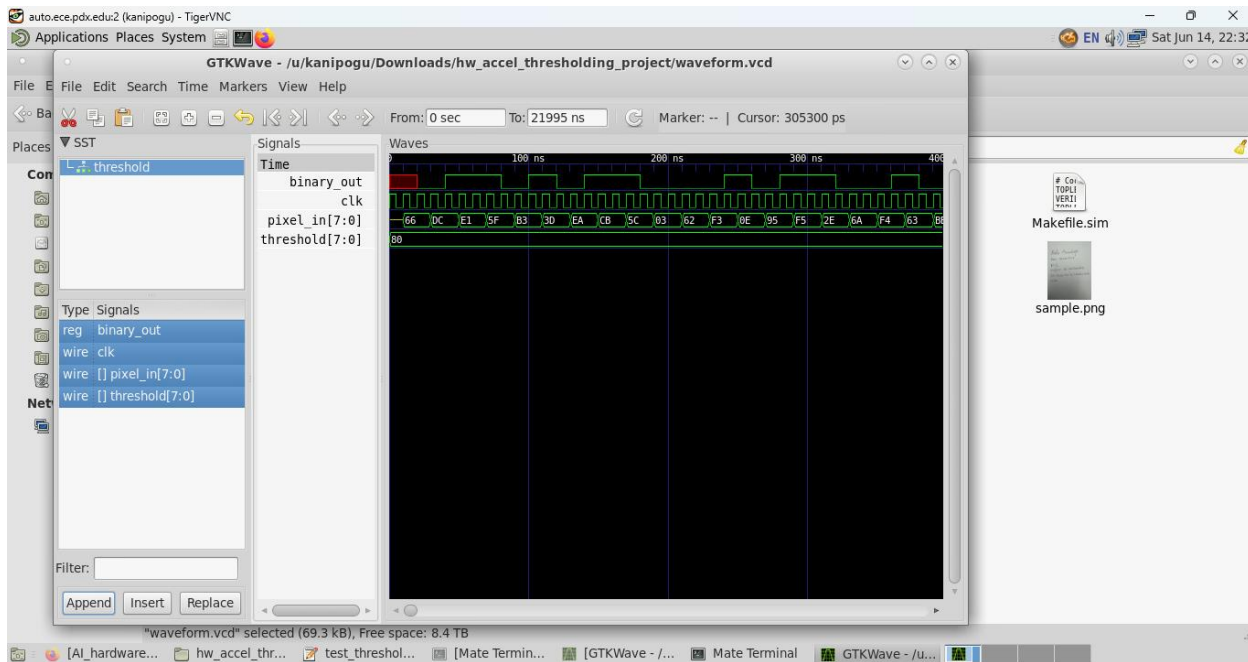
How to view:

```
``bash gtkwave waveform.vcd
```

Waveform Interpretation:

- `clk`: toggling every 10ns.
- `pixel_in`, `threshold`: driven by the testbench.
- `binary_out`: 1 if `pixel_in > threshold`, else 0.

Result: The waveform confirmed the correct behavior over 1000 pixel inputs.



Step 4: Add coverage and corner-case analysis

Testbench was enhanced with:

```
corner_cases = np.array([0, 1, 126, 127, 128, 254, 255], dtype=np.uint8)
```

Logs like: Below: 504, Equal: 3, Above: 493

What I have learned:

- The design was tested over full 8-bit range.
- Coverage ensures correct behavior around edge thresholds.

```
# 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.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
```

Step 5: Integration with Software Pipeline

I have integrated the testbench to be invoked from Python (`ocr_pipeline.py`) using:

```
```bash
```

```
python3 ocr_pipeline.py --use_hw
```

If `--use\_hw` is passed:

- Hardware thresholding is triggered via `make SIM=questa`.

What I have learned:

- HW-SW co-design requires careful interface handling.

## **Step 6: Optimization over naive design**

Initial Design:

- Software ran first.
- Saved grayscale image to `.txt`.
- Hardware separately read `.txt` and output binary image.

**Can check the earlier design inside project\_final folder in my Github:**

**[\[https://github.com/kanipogu/AI-Hardware/tree/main/Project\\_final\]](https://github.com/kanipogu/AI-Hardware/tree/main/Project_final)**

Problem:

- Unnecessary delays and poor pipeline cohesion.

Improved Design

- Python calls hardware inline with `threshold\_image\_hw(grayscale\_img)`.

## **Step 7: Synthesis of Verilog Module**

Tools Used:

- Synopsys Design Compiler
- **OSU Standard Cell Library**

Results:

Area:

Number of ports:	18
Number of nets:	48
Number of cells:	31
Number of combinational cells:	30
Number of sequential cells:	1
Number of macros/black boxes:	0
Number of buf/inv:	15
Number of references:	5
Combinational area:	5679.000000
Buf/Inv area:	2160.000000
Noncombinational area:	864.000000
Macro/Black Box area:	0.000000
Net Interconnect area:	undefined (No wire load specified)
Total cell area:	6543.000000

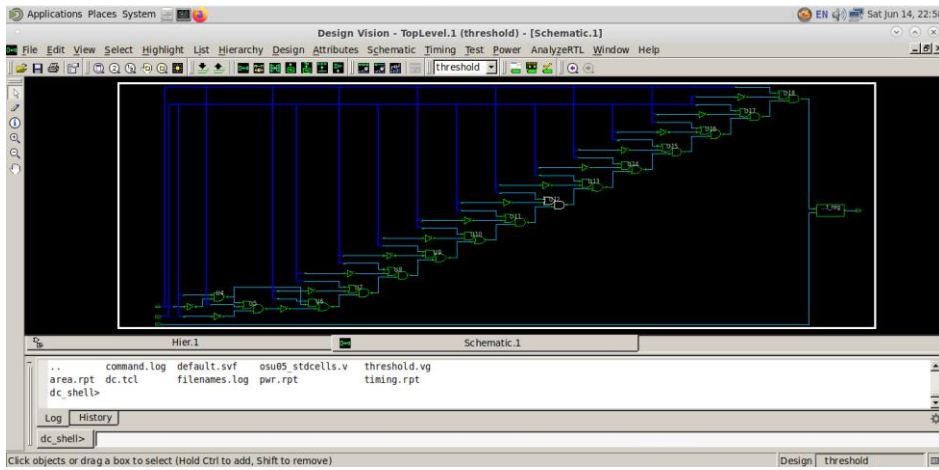
Timing Slack:

27	threshold[0] (in)	0.00	2.00	r
28	U19/Y (INVX2)	0.05	2.05	f
29	U4/Y (NAND2X1)	0.16	2.20	r
30	U5/Y (OAI21X1)	0.13	2.33	f
31	U27/Y (INVX2)	0.08	2.42	r
32	U6/Y (AOI21X1)	0.12	2.53	f
33	U7/Y (OAI21X1)	0.12	2.65	r
34	U8/Y (OAI21X1)	0.09	2.75	f
35	U9/Y (AOI21X1)	0.10	2.84	r
36	U10/Y (AOI21X1)	0.12	2.97	f
37	U11/Y (OAI21X1)	0.11	3.08	r
38	U12/Y (OAI21X1)	0.09	3.17	f
39	U13/Y (AOI21X1)	0.10	3.27	r
40	U14/Y (AOI21X1)	0.12	3.39	f
41	U15/Y (OAI21X1)	0.11	3.50	r
42	U16/Y (OAI21X1)	0.08	3.58	f
43	U17/Y (OAI21X1)	0.10	3.68	r
44	U18/Y (OAI21X1)	0.07	3.76	f
45	binary_out_reg/D (DFFPOSX1)	0.00	3.76	f
46	data arrival time		3.76	
47				
48	clock clk (rise edge)	10.00	10.00	
49	clock network delay (ideal)	0.00	10.00	
50	binary_out_reg/CLK (DFFPOSX1)	0.00	10.00	r
51	library setup time	-0.30	9.70	
52	data required time		9.70	
53	-----			
54	data required time		9.70	
55	data arrival time		-3.76	
56	-----			
57	slack (MET)		5.94	

### Power:

Power Group	Internal Power	Switching Power	Leakage Power	Total Power	( % )	Attrs
-----						
io_pad	0.0000	0.0000	0.0000	0.0000	( 0.00%)	
memory	0.0000	0.0000	0.0000	0.0000	( 0.00%)	
black_box	0.0000	0.0000	0.0000	0.0000	( 0.00%)	
clock_network	0.0000	0.0000	0.0000	0.0000	( 0.00%)	
register	0.1892	0.0000	0.2433	0.1892	( 36.40%)	
sequential	0.0000	0.0000	0.0000	0.0000	( 0.00%)	
combinational	0.1960	0.1346	1.5157	0.3306	( 63.60%)	
-----						
Total	0.3852 mW	0.1346 mW	1.7589 nW	0.5199 mW		
1						

### Synthesis Schematic:



## Genrated Netlist:

////////////////////////////////////

// Created by: Synopsys DC Expert(TM) in wire load mode

// Version : Q-2019.12-SP3

// Date : Sat Jun 14 22:56:59 2025

////////////////////////////////////

module threshold ( clk, pixel in, threshold, binary out );

input [7:0] pixel in;

input [7:0] threshold;

input clk;

output binary out;

wire N0, n2, n3, n4, n5, n6, n7, n8, n9, n10, n11, n12, n13, n14, n15, n16,

n17, n18, n19, n20, n21, n22, n23, n24, n25, n26, n27, n28, n29, n30;

DFFPOSX1 binary out reg ( .D(N0), .CLK(clk), .Q(binary out) );

NAND2X1 U4 ( .A(pixel in[0]), .B(n16), .Y(n3) );

OAI21X1 U5 ( .A(n3), .B(threshold[1]), .C(n23), .Y(n2) );

AOI21X1 U6 ( .A(threshold[1]), .B(n3), .C(n24), .Y(n4) );

OAI21X1 U7 ( .A(pixel in[2]), .B(n17), .C(n4), .Y(n5) );

OAI21X1 U8 ( .A(threshold[2]), .B(n25), .C(n5), .Y(n6) );

AOI21X1 U9 ( .A(pixel in[3]), .B(n18), .C(n6), .Y(n7) );  
AOI21X1 U10 ( .A(threshold[3]), .B(n26), .C(n7), .Y(n8) );  
OAI21X1 U11 ( .A(pixel in[4]), .B(n19), .C(n8), .Y(n9) );  
OAI21X1 U12 ( .A(threshold[4]), .B(n27), .C(n9), .Y(n10) );  
AOI21X1 U13 ( .A(pixel in[5]), .B(n20), .C(n10), .Y(n11) );  
AOI21X1 U14 ( .A(threshold[5]), .B(n28), .C(n11), .Y(n12) );  
OAI21X1 U15 ( .A(pixel in[6]), .B(n21), .C(n12), .Y(n13) );  
OAI21X1 U16 ( .A(threshold[6]), .B(n29), .C(n13), .Y(n14) );  
OAI21X1 U17 ( .A(pixel in[7]), .B(n22), .C(n14), .Y(n15) );  
OAI21X1 U18 ( .A(threshold[7]), .B(n30), .C(n15), .Y(N0) );  
INVX2 U19 ( .A(threshold[0]), .Y(n16) );  
INVX2 U20 ( .A(threshold[2]), .Y(n17) );  
INVX2 U21 ( .A(threshold[3]), .Y(n18) );  
INVX2 U22 ( .A(threshold[4]), .Y(n19) );  
INVX2 U23 ( .A(threshold[5]), .Y(n20) );  
INVX2 U24 ( .A(threshold[6]), .Y(n21) );  
INVX2 U25 ( .A(threshold[7]), .Y(n22) );  
INVX2 U26 ( .A(pixel in[1]), .Y(n23) );  
INVX2 U27 ( .A(n2), .Y(n24) );  
INVX2 U28 ( .A(pixel in[2]), .Y(n25) );  
INVX2 U29 ( .A(pixel in[3]), .Y(n26) );  
INVX2 U30 ( .A(pixel in[4]), .Y(n27) );  
INVX2 U31 ( .A(pixel in[5]), .Y(n28) );  
INVX2 U32 ( .A(pixel in[6]), .Y(n29) );  
INVX2 U33 ( .A(pixel in[7]), .Y(n30) );

endmodule

[https://github.com/kanipogu/AI-Hardware/tree/main/Project final/Final updated project thresholding/hw accel thresholding project/d](https://github.com/kanipogu/AI-Hardware/tree/main/Project%20final/Final%20updated%20project%20thresholding/hw%20accel%20thresholding%20project/d%20c)  
[c](#)

## Step 9: Lessons Learned

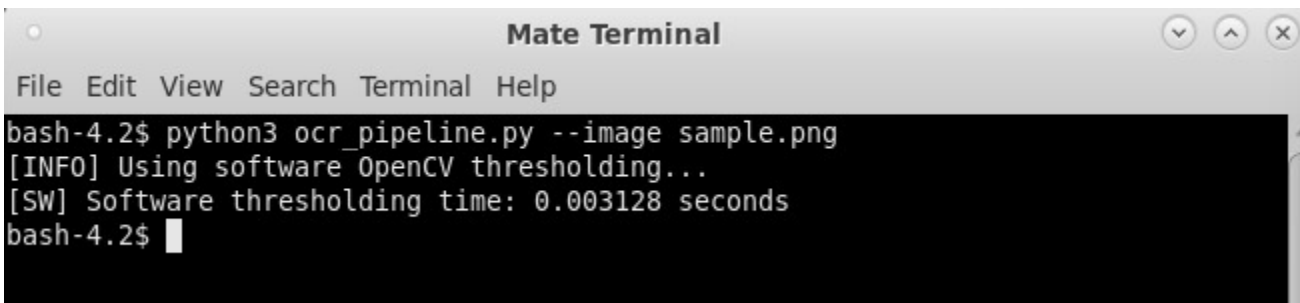
- Cocotb is a powerful tool for rapid prototyping and validation of hardware modules.

- HW-SW Co-Design improves performance only when bottlenecks are correctly identified.
- Synthesis provides concrete proof of timing/power compliance.
- Waveform debugging is critical when hardware fails.
- Regression tests ensure consistency across changes.

## How to Run the Project

### Software-Only:

```
```bash
python3 ocr_pipeline.py --image sample.png
```
```



```

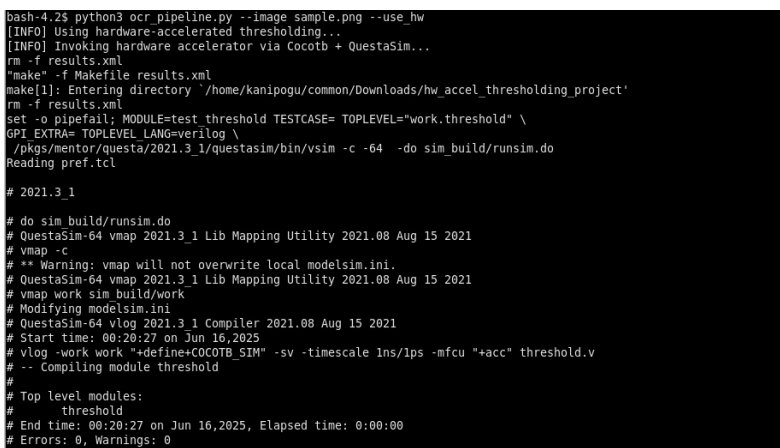
Mate Terminal
File Edit View Search Terminal Help
bash-4.2$ python3 ocr_pipeline.py --image sample.png
[INFO] Using software OpenCV thresholding...
[SW] Software thresholding time: 0.003128 seconds
bash-4.2$

```

This Generates the output using software model. The purpose of this is to compare software time with Hardware Time.

### Hardware-Accelerated:

```
```bash
python3 ocr_pipeline.py --image sample.png --use_hw
```
```



```

bash-4.2$ python3 ocr_pipeline.py --image sample.png --use_hw
[INFO] Using hardware-accelerated thresholding...
[INFO] Invoking hardware accelerator via Cocotb + QuestaSim...
rm -f results.xml
"make" -f Makefile results.xml
make[1]: Entering directory '/home/kanipogu/common/Downloads/hw_accel_thresholding_project'
rm -f results.xml
set -o pipefail; MODULE=test threshold TESTCASE= TOPLEVEL="work.threshold" \
GPI EXTRA= TOPLEVEL LANG=verilog \
./pkg/mentor/questa/2021.3_1/questasim/bin/vsim -c -64 -do sim_build/runsim.do
Reading pref.tcl

2021.3_1

do sim_build/runsim.do
QuestaSim-64 vmap 2021.3_1 Lib Mapping Utility 2021.08 Aug 15 2021
vmap -c
** Warning: vmap will not overwrite local modelsim.ini.
QuestaSim-64 vmap 2021.3_1 Lib Mapping Utility 2021.08 Aug 15 2021
vmap work sim_build/work
Modifying modelsim.ini
QuestaSim-64 vlog 2021.3_1 Compiler 2021.08 Aug 15 2021
Start time: 00:20:27 on Jun 16,2025
vlog -work work "+define+COCOTB_SIM" -sv -timescale 1ns/1ps -mfcu "+acc" threshold.v
-- Compiling module threshold
#
Top level modules:
threshold
End time: 00:20:27 on Jun 16,2025, Elapsed time: 0:00:00
Errors: 0, Warnings: 0

```



```

// is prohibited from disclosure under the Trade Secrets Act.
// 18 U.S.C. Section 1905.
//
//
Loading vtr.tcl
Loading work.threshold(fast)
Loading /u/kaniopu/.local/lib/python3.6/site-packages/cocotb/libs/libcocotbvpi_modelsim.so
--no INFO gpi ..nbed/gpi_embed.cpp:61 in set_program_name in venv Did not detect Python virtual environment. Using
system-wide Python interpreter
--no INFO gpi ..gpi/gpiCommon.cpp:101 in gpi_print_registered_impl VPI registered
0.00ms INFO cocotb Running on ModelSim for Questa-64 version 2021.3.1 2021.08
0.00ms INFO cocotb Running tests with cocotb v1.9.2 from /u/kaniopu/.local/lib/python3.6/site-packages/cocotb
0.00ms INFO cocotb Sending Python random module with IP3065429
0.00ms INFO cocotb.pytest not found, install it to enable better AssertionError messages
0.00ms INFO cocotb.regression Found test test_threshold.test_threshold
0.00ms INFO cocotb.regression running test threshold (1/1)
0.00ms INFO cocotb.threshold Starting threshold test...
20000.00ms INFO cocotb.threshold [COVERAGE] Below: 364, Equal: 3, Above: 493
20000.00ms INFO cocotb.threshold [VPI] Hits: 364 vs golden model: 0
22000.00ms INFO cocotb.regression test threshold passed
23000.00ms INFO cocotb.regression

** TEST STATUS SIM TIME (ns) REAL TIME (s) RATIO (ns/s) **

** test_threshold.test_threshold PASS 22000.00 0.36 60875.72 **

** TESTS: 1 PASS=1 FAIL=0 SKIP=0 22000.00 0.33 41748.97 **

** Note: $finish : threshold.v(33)
Time: 22000001 ps Iteration: 0 Instance: /threshold
End time: 00:20:29 on Jun 16, 2023, Elapsed time: 0:00:02
Errors: 0 Warnings: 1
make[1]: Leaving directory /home/kaniopu/common/Downloads/hw_accel_thresholding_project/
[hw] Thresholding complete in 10.536406 seconds
make[1]:

```

Make sure `questa` is in PATH, and `Makefile` is correctly written.

## ChatGpt Prompts I have Used:

1) “This is my ocr\_pipeline.py... how to preprocess and do thresholding using software or hardware based on a flag?”

Ans) Impact: Helped design a dual-mode pipeline (--use\_hw) to compare software (OpenCV) vs hardware (Verilog) thresholding.

2) “I still see noise... where should I modify this?”

“Apply same preprocessing for hardware input — where should I do that?”

Ans) Impact: Clarified the need to apply CLAHE, resize, blur consistently to both hardware and software branches to ensure fair comparison.

3) “Where do I compare golden reference with DUT?”

“How is data randomized, and where is coverage reported?”

Ans) Impact:

Helped you use:

python

reference = (pixel\_data > threshold\_val).astype(np.uint8)

for ground truth comparison, and added logs:

python

dut.\_log.info(f"[COVERAGE] Below: {below}, Equal: {equal}, Above: {above}")

4) “Why is my waveform empty?”

“How to enable waveform dumping?”

Ans) Resolved by adding:

initial begin

```
$dumpfile("waveform.vcd");
```

```
$dumpvars(1, clk, pixel_in, threshold, binary_out);
```

End

and launching with:

gtkwave waveform.vcd

5) “In my earlier code, grayscale was passed from software to hardware after the full image. How to accelerate the thresholding step only?”

Ans) Impact:

You restructured your project to accelerate thresholding alone (real-time style), reducing overhead and better matching hardware use-cases.

6) “Why don’t I see [COVERAGE] and [VERIFY] printed in my terminal after hardware simulation?”

Ans) Impact:

Diagnosed and fixed issues with:

- Incorrect indentation or Cocotb assert failure before logs
- Ensured print statements ran before the test exited

Improvement: Better debugging confidence and log completeness in hardware test automation.

7) “How do I save and compare hardware-generated binary image against the software output?”

Ans) Impact:

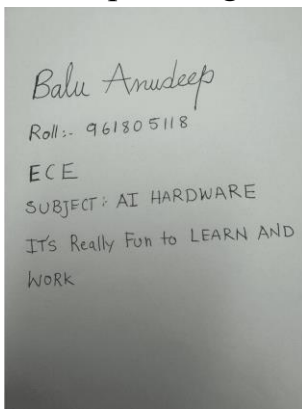
You created:

```
cv2.imwrite("results/threshold_hw_output.png", hw_out * 255)
```

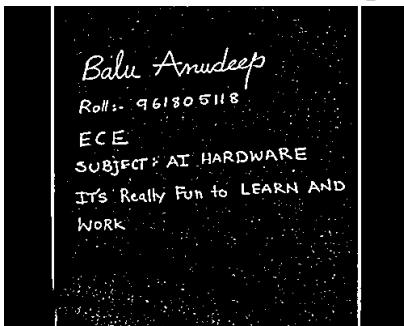
Enabled real-world testing on actual document images.

## **Results:**

Input Sample Image:



Software Generated Output:

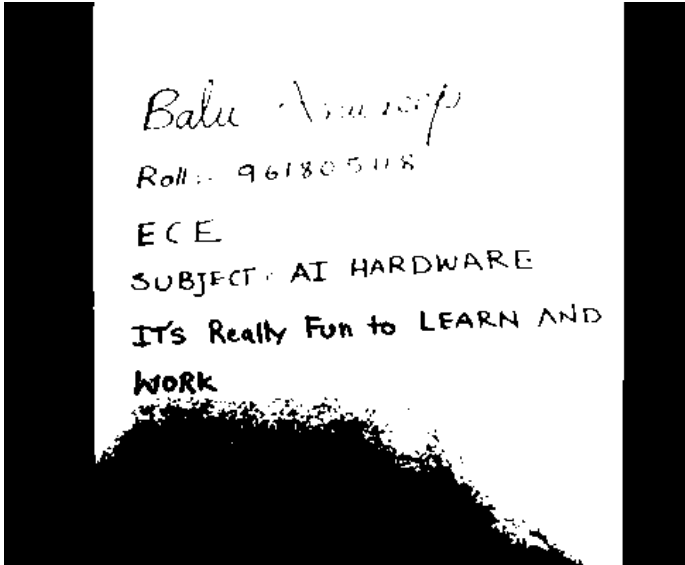


**Software-Generated Output**

- This image contains a lot of dot noise (especially visible in the white background).
- It was generated by ocr\_pipeline.py without the --use\_hw flag.

- Uses OpenCV's adaptive + Otsu thresholding.
- File path: results/final\_combined\_threshold.png

### Hardware-Generated Output:



### Hardware-Generated Output

- This is cleaner, with less noise and a sharper binary result.
- Generated by ocr\_pipeline.py --use\_hw, which offloads the thresholding to your Verilog hardware via Cocotb + QuestaSim.
- File path: results/threshold\_hw\_output.png

### Conclusion:

This project evolved from a naive software model to a performance optimized HW/SW co-design with thorough simulation, verification, and synthesis. It demonstrates:

- Proper bottleneck identification
- Seamless hardware invocation from software
- Improved execution time(When a real hardware is connected Ex: FPGA)
- Verified correctness and waveform integrity.