SoC 101:

*a.k.a., "**Everything you wanted to know about a computer but were afraid to ask**"*

Lecture 8:
Ramp up and Debug

a.k.a. "Lupulus: A Debug Story"

Prof. Adam Teman

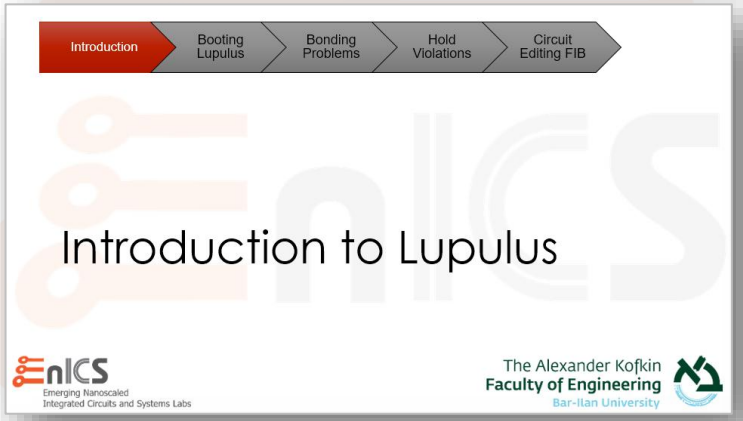
EnICS Labs, Bar-Ilan University

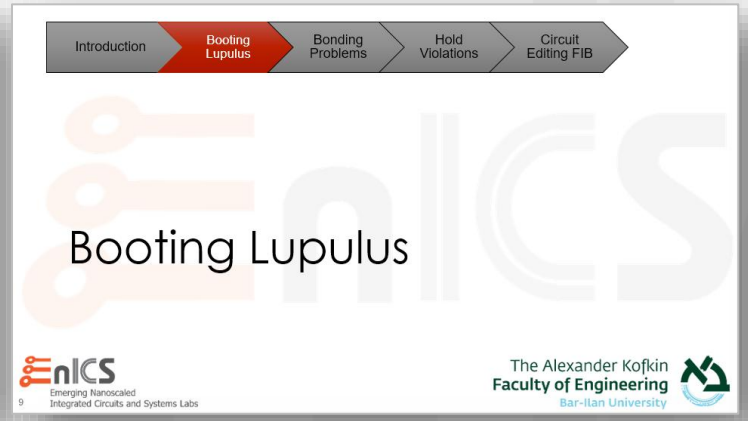
23 June 2025

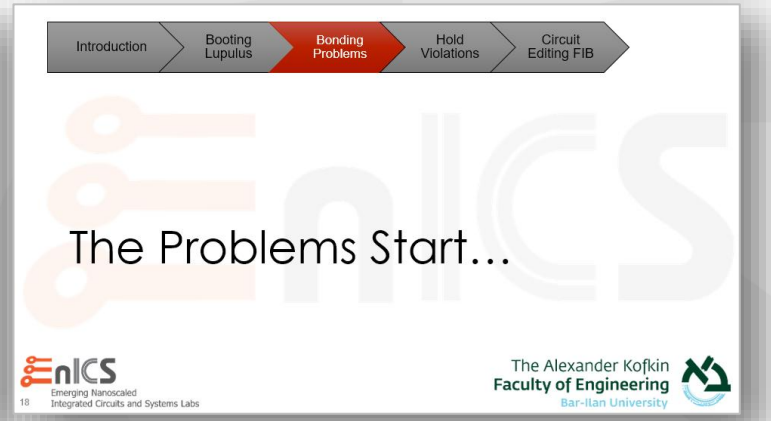
Emerging Nanoscaled
Integrated Circuits and Systems Labs

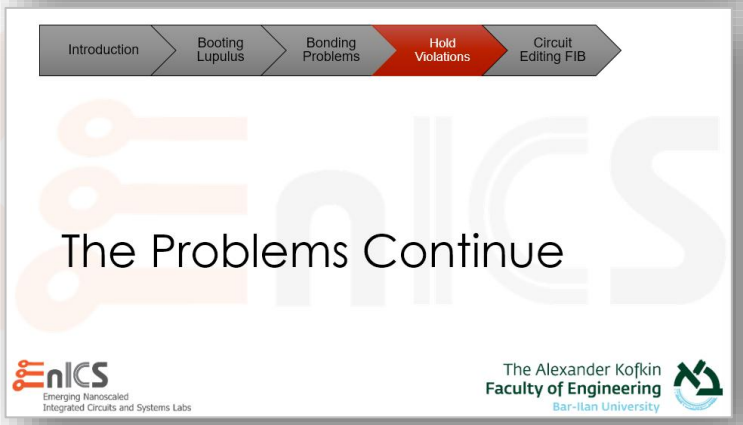
The Alexander Kofkin
Faculty of Engineering
Bar-Ilan University

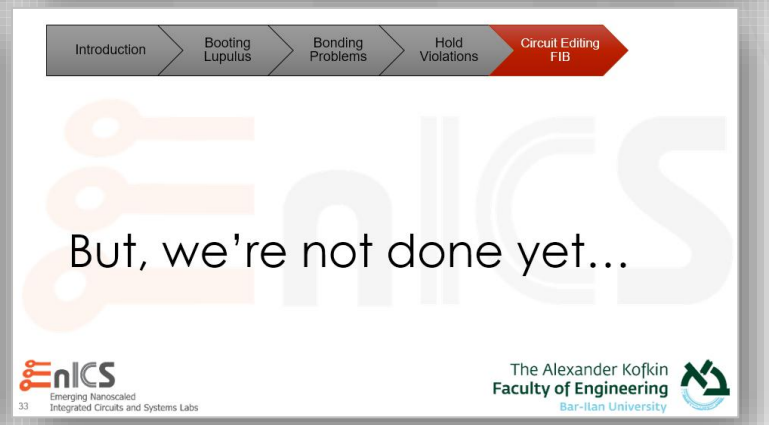
Lecture Overview







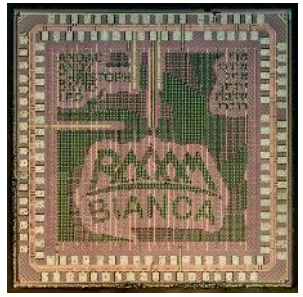


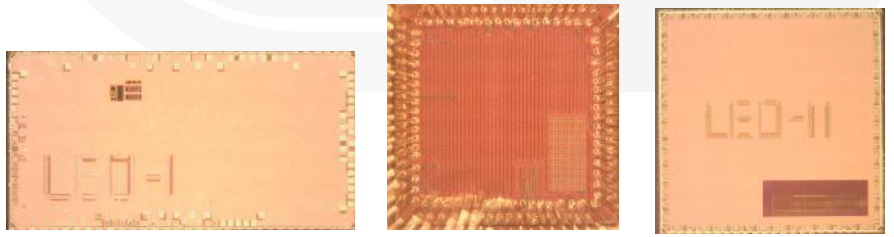


Introduction to Lupulus

A Little Background

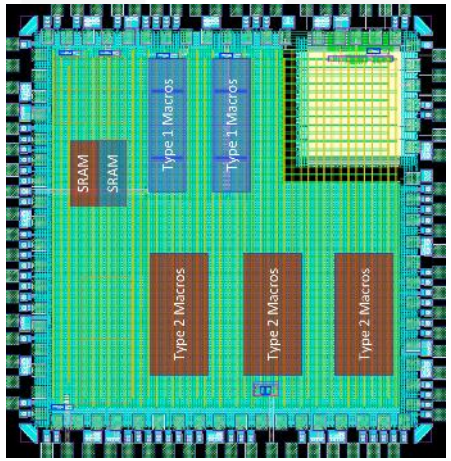
- **The EnICS Labs SoC Platform:**
  - In the framework of the GenPro Consortium, the EnICS Labs were mandated with developing the "Israeli RISC-V Core"
  - HAMSA-DI – a superscalar version of the popular RI5CY Core was the result of this project.
  - HAMSA-DI is delivered as part of PulpEnIX – a full SoC Platform based on the Pulpino Platform from the PULP Project.
  - PulpEnIX/HAMSA-DI had already been fabricated on three test chips in 65nm and 16nm FinFET.
- **In 2022, we taped out "Bianca", a 16nm test chip, based on the PulpEnIX SoC platform.**

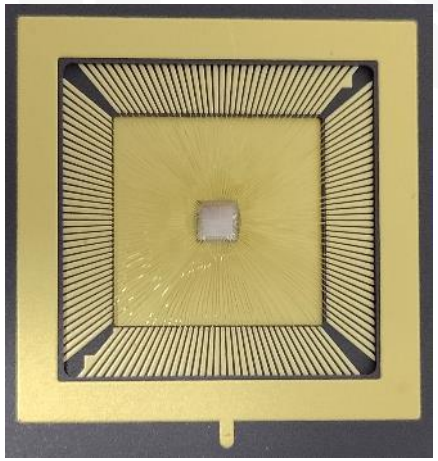


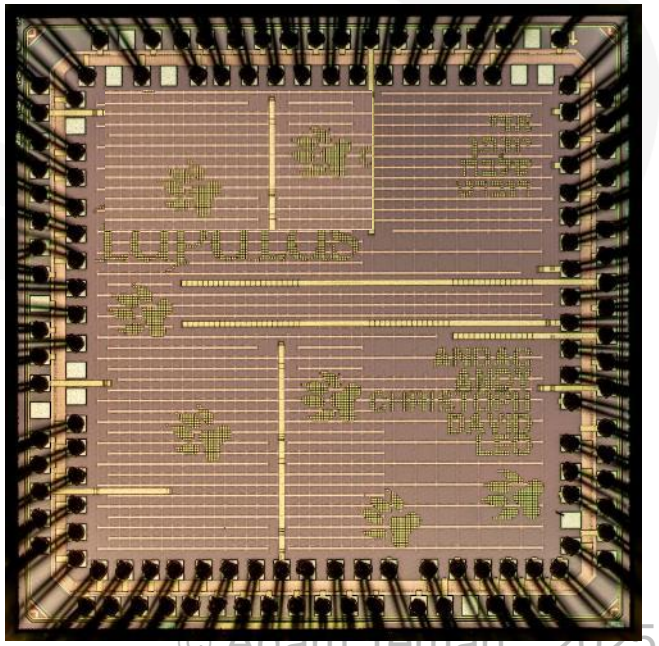


Lupulus

- **As a follow up to Bianca, a second tapeout was sent for fabrication in 2023.**
  - This tapeout included improvements to the test structures.
- **Lupulus was based on the *same Top Level* design as Bianca, including:**
  - PulpEnIX SoC Platform with HAMSA-DI host.
  - PULP-GCC compliant software stack with PulpEnIX bare-metal libraries.
  - Same 144-pin PGA package with almost identical pinout.
  - Compatible with same PCB with minimal modifications.
- **The Lupulus test chip was delivered and bonded in October 2023.**

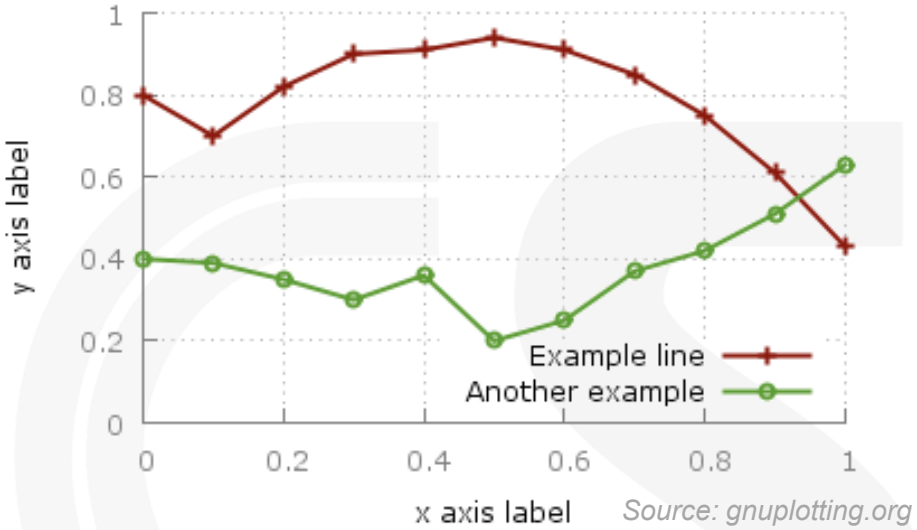






What next?

- We have done some great R&D…
  Made something innovative.
- *But how do we make sure that it actually works?*
  How do we test it and make measurements
  to create those nice plots?

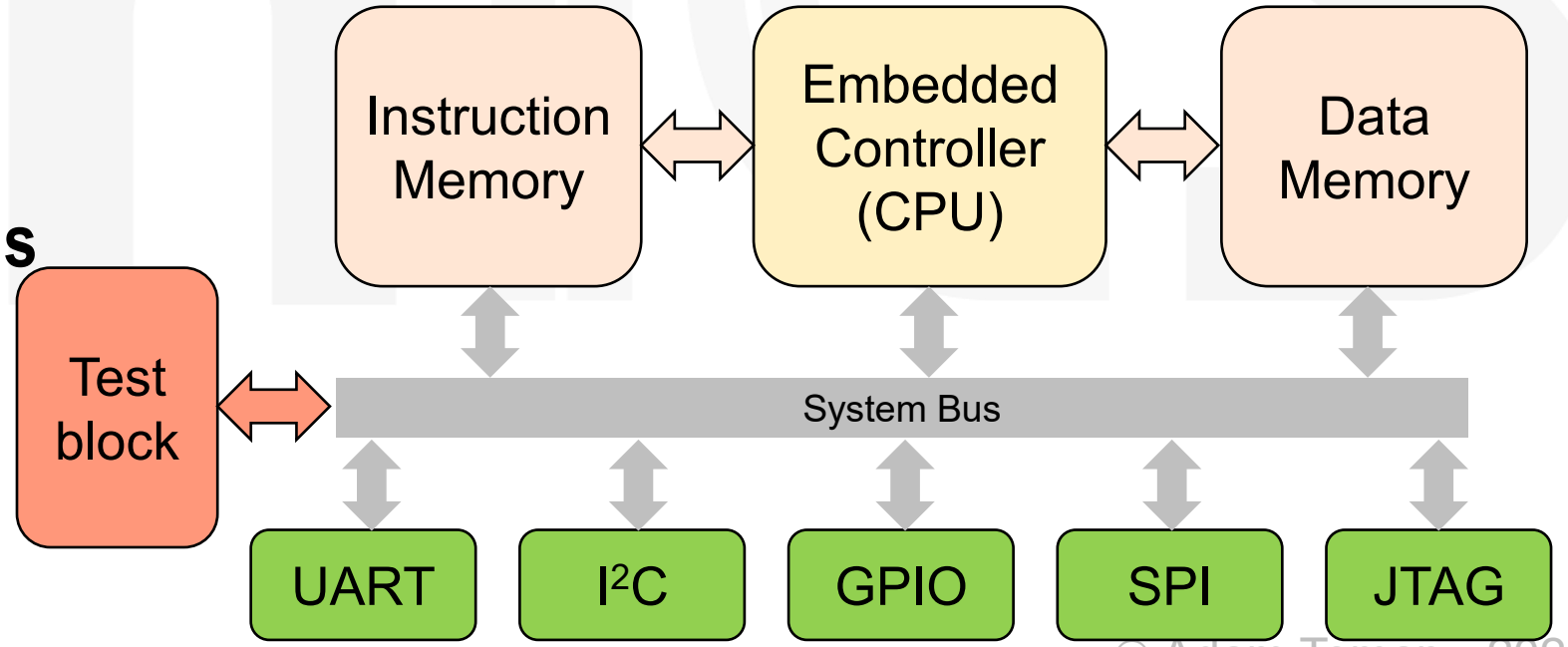


Source: gnuplotting.org

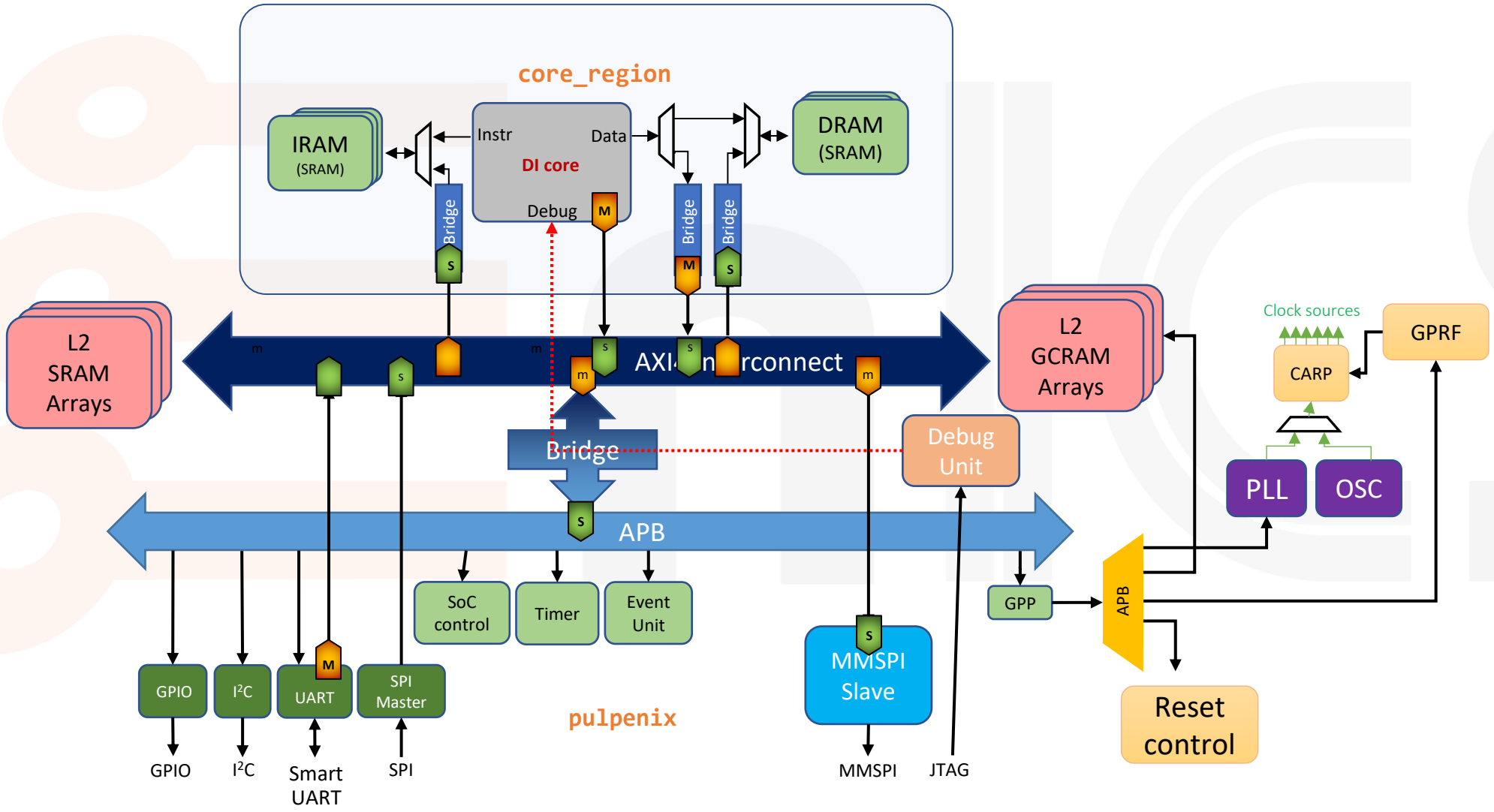
- The best way is usually to integrate your block with a control platform.
  An SoC!
- Let's remember what an SoC is and how it can help us before continuing.

Back to our early SoC101 lectures

- **A basic SoC will probably contain something like the following:**
  - An embedded controller (a CPU) for running C-code
  - Some tightly coupled memory for storing program code and data
  - A bunch of peripherals for communicating with the outside world
  - A system bus to send and receive data between the components
- **Additional components and test blocks will usually be connected to the system bus**
- **All components are memory mapped with predefined address spaces**

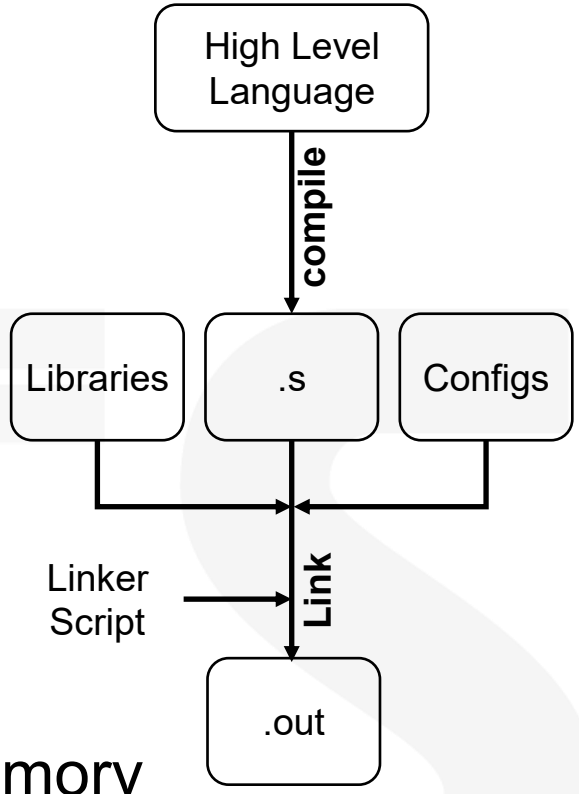


16nm Lupulus SoC



Booting Lupulus

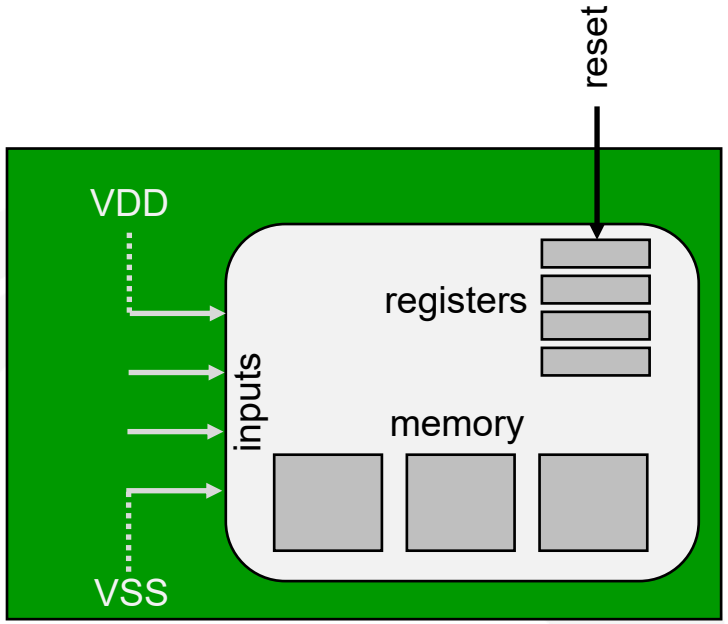
Loading software on to the SoC



- **The SoC is equipped with a software toolchain**
  - Write programs in C code
  - Compile the programs to the ISA (e.g., RISC-V)
  - Use "bare metal" libraries for common operations (e.g., `printf`, `gets`)
  - Link with IRQs and boot code to configure SoC
  - Provide a linker script to direct binaries to instruction/data memory
- **What *actually happens* when we hit the reset button?**
  - A small number of hard coded (RTL) instructions are run
  - They load boot code ("bootloader") from an external source
  - The boot code loads the binaries to instruction and data memory
  - The processor jumps to the first address in the program (`main()`)

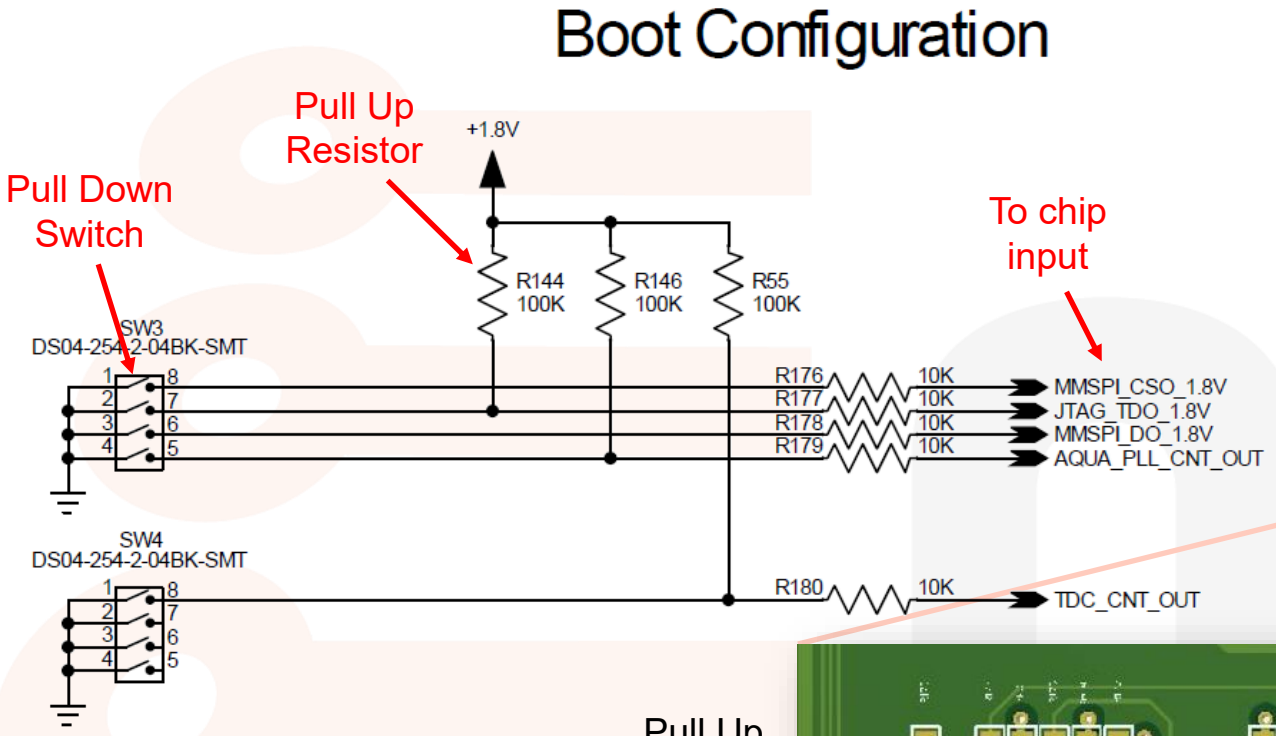
Configuring the Boot Sequence

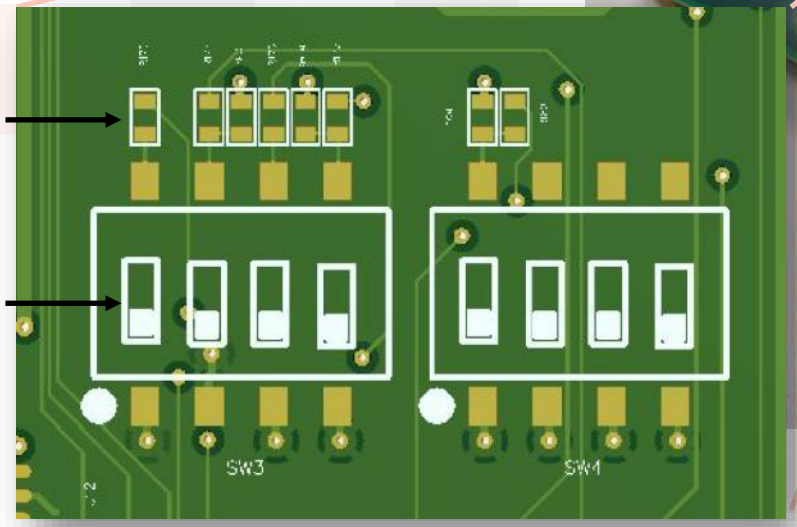
- **Remember that:**
  - The *state* of a system is set by its inputs and memory (including registers).
  - Inputs come from pads that are connected to the board.
  - Every flip-flop on the chip gets a reset value.
  - Memory (SRAM) is garbage at start up.

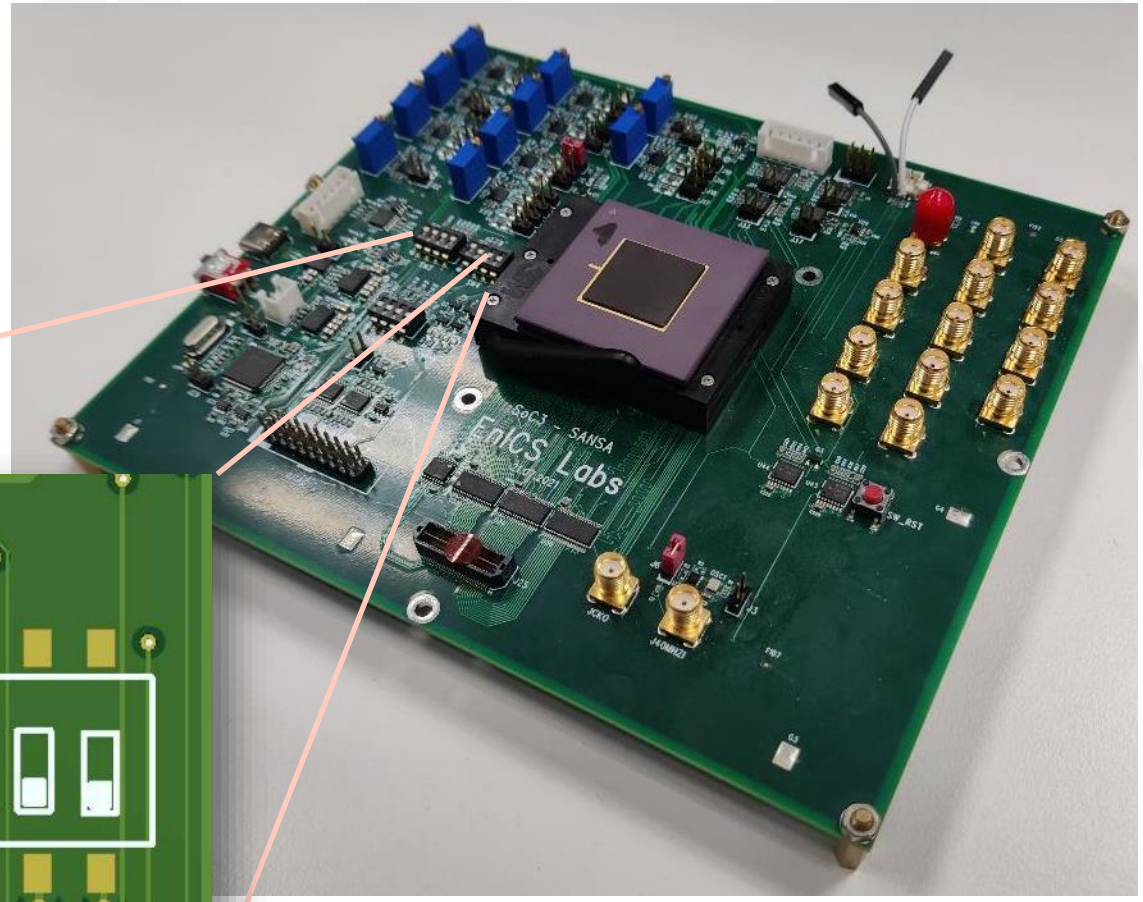


- **Configuration switches ("bootstraps")**
  - The chip will have several inputs to configure startup values.
  - These are either tied to "pull ups" (VDD) or "pull downs" (VSS) on the board.
  - You can often change these by toggling a switch or using a jumper.

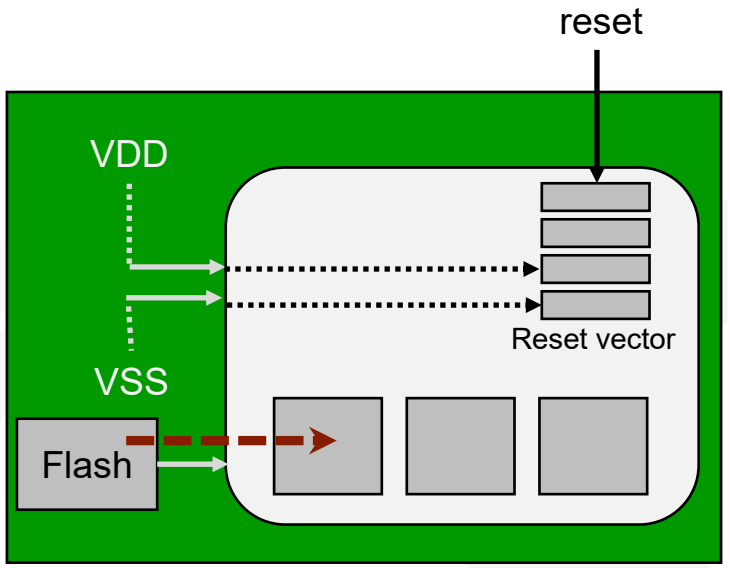
Example of PCB bootstraps







So let's go back to reset…



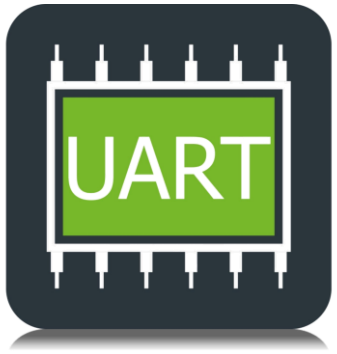
- **Reset Sequence**
  - When we hit reset, registers are set to default values.
  - Some of the registers sample the pull up/pull down value from the board.
    - These can be configured through bootstraps!
  - One of registers is the reset vector, which is the address of the program to run.
  - This address may be mapped to an external device (e.g., Boot ROM or Flash).
  - The short program is the bootloader that copies the program to SRAM.
  - After the program is loaded, the CPU jumps to the instruction memory.

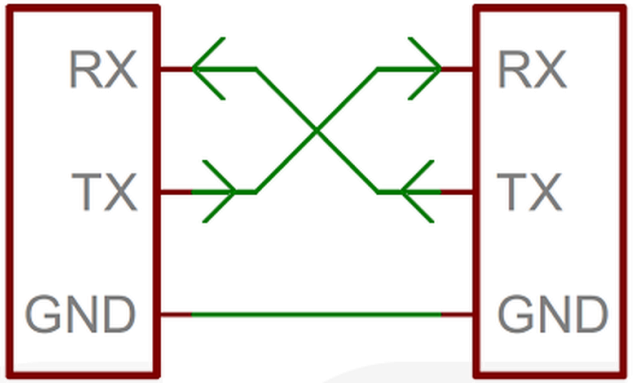
What happens in PulpEnIX


Source: blum

- **Boot is (probably) the single-most "dangerous" part of your chip design**
  - Udi's Drawer Analogy:
    "Slam a drawer shut, but try to throw something into it before it closes"
  - Therefore, always provide several alternatives.
- **PulpEnIX has several methods for booting:**
  - **MMSPI**:
    As described before, Flash is memory mapped and reset vector points to it.
  - **JTAG**:
    Utilize JTAG test ports to control the SoC with the OpenOCD standard.
  - **SmartUART + pyShell**:
    Backdoor into the SoC through the UART port.
    Then control the SoC through Interactive Python interface.

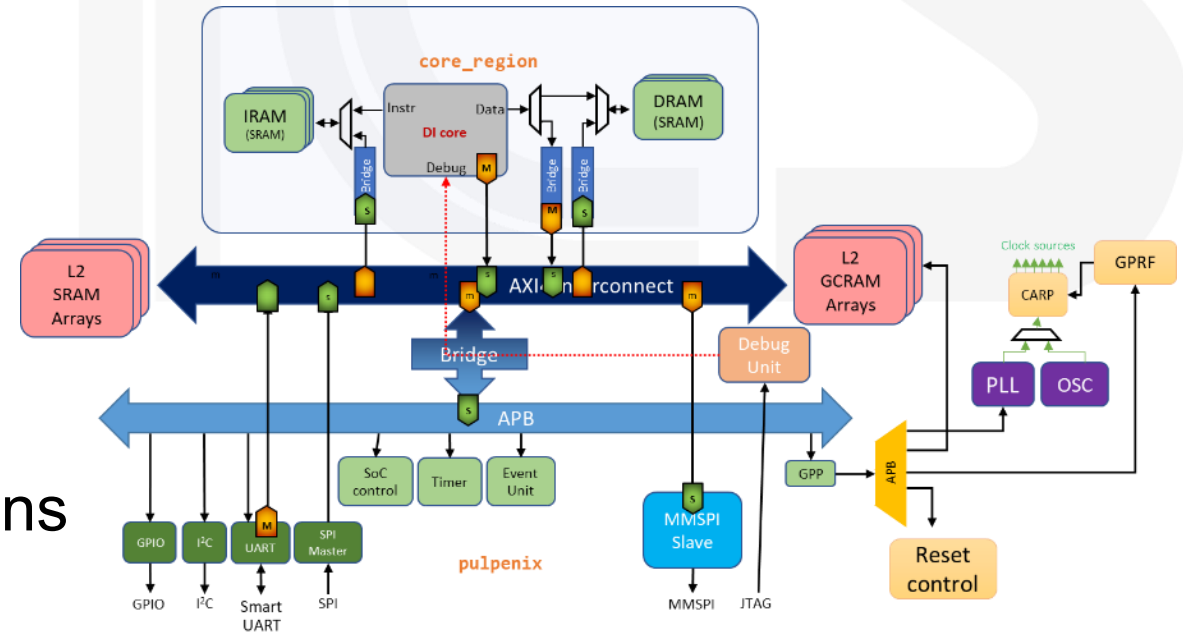
SmartUART



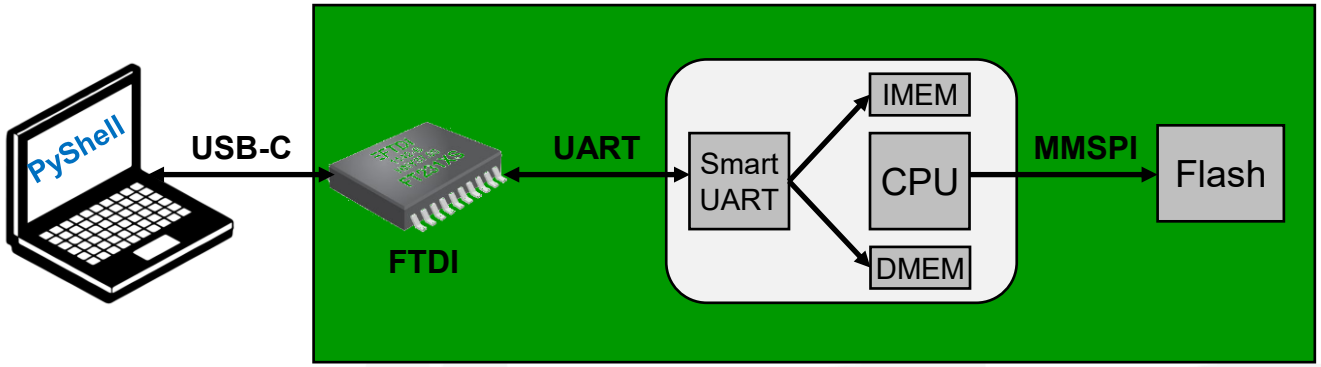


- **Reminder: UART**
  - UART is a very simple, two-wire (TX/RX) serial interface.
  - PulpEnIX has a UART Slave port, i.e., only the CPU can initiate transactions.
  - For example, `bm_printf()` sends characters over AXI/APB to a FIFO in the UART controller. The controller serializes the data and transmits it.
  - This configuration does not provide a good debugging interface.

- **Introducing "SmartUART"**
  - Add an AXI Master port to the UART.
  - Use UART to initiate transactions: read/write to all memory mapped locations



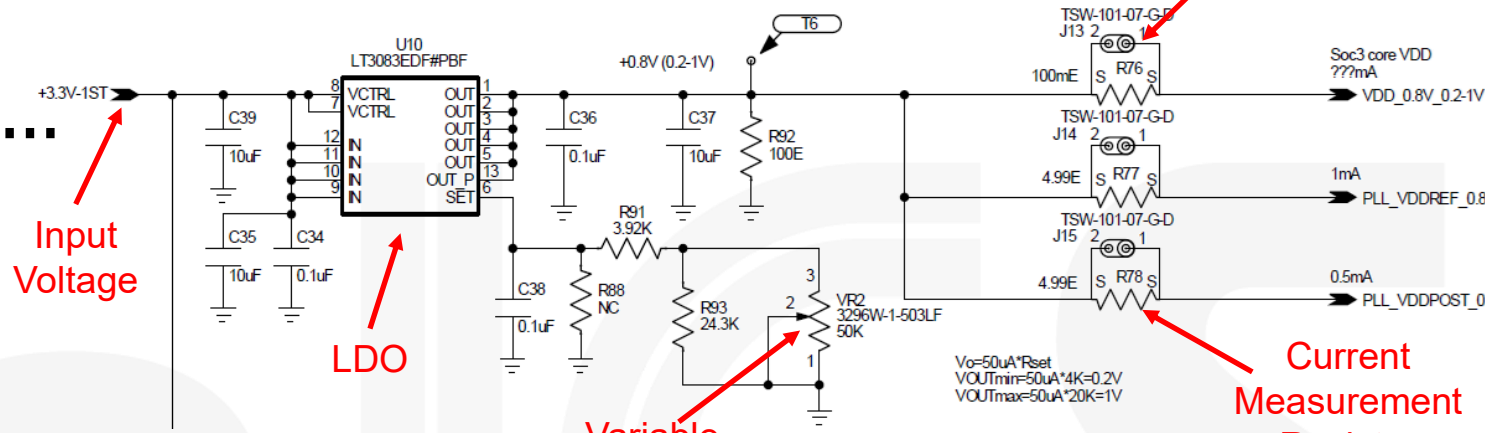
SmartUART



- **How does SmartUART work?**
  - Host PC is connected to FTDI chip, which translates USB-C into a bunch of signals, including UART, that are routed to chip.
  - Reset tries to boot PulpEnIX through MMSPI by default.
  - Host PC sends signal to chip (through FTDI) that stops the CPU.
  - SmartUART sends write commands (Host PC through FTDI to AXI Master) to program registers and to load code into SRAM.
  - Read commands from SmartUART bypass FIFO.
- **How does SmartUART signal a read/write command?**
  - UART transmits bytes (8 bits), usually representing ASCII chars.
  - Bit 7 (MSB) is unused in ASCII. This bit is utilized to signal transactions.
  - Python interface on host (PyShell) initiates transactions and polls for response.

Why did we go through all of this?

- **Well, our chips have just arrived…**
  What do we do now?
  - Set up our voltages!
  - Connect power supply and/or configure LDOs.
- **Okay, Voltages all set. What now?**
  - Run a program, of course!
- **But it doesn't work…**
  - Never does, unfortunately…
- **So, we need to start debugging…**
  - Thank God for SmartUART!



The Problems Start…

Hello, Who?

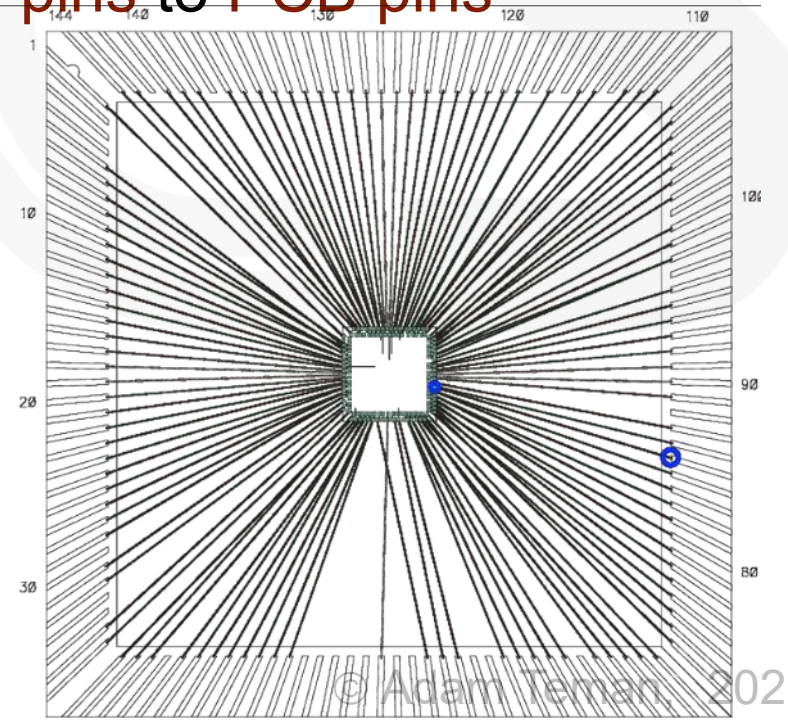
- **Lupulus was supposed to be a "plug-and-play" chip**
  - PulpEnIX platform taped out several times before.
  - Package compatible with silicon-proven "Sansa" board
  - Toplevel design (almost) equivalent to "Bianca" chip
  - Really, just put the chip in the socket, run the "Hello, World!" program from Bianca, and no reason it won't work.
- **So, why doesn't the chip react?**
  - Check voltages – look fine.
  - Check clock signal – looks fine.
  - Where is the program crashing?
    Looks like no UART communication.
- **What now?**

```
(base) PS C:\Users\Lidor Geri\Documents\measurements_new_environment\measurements_new_environment\lupulus\runspace> lupu
Compiling Application helloworld with  link define=p , march=rv32imxpulpv3 , opt=O3 , di=DI_ON , period=5 , baudrate = 9
..\sw_apps\libs\sys_lib\src\rcg.c: In function 'init_pll_rcg':
..\sw_apps\libs\sys_lib\src\rcg.c:10:3: warning: implicit declaration of function 'bm_printf' [-Wimplicit-function-decla
   bm_printf("starting init_pll\n");
   ^~~~~~~~~~
..\sw_apps\libs\sys_lib\src\rcg.c:13:58: warning: variable 'rd_data_cfg3' set but not used [-Wunused-but-set-variable]
   unsigned int rd_data_cfg0, rd_data_cfg1, rd_data_cfg2, rd_data_cfg3;
                                                          ^~~~~~~~~~~~
..\sw_apps\libs\sys_lib\src\rcg.c: In function 'set_rand_pll_div':
..\sw_apps\libs\sys_lib\src\rcg.c:135:18: warning: implicit declaration of function 'rand' [-Wimplicit-function-declarat
    refdiv    = 1+rand() % 3;
                  ^~~~
..\sw_apps\libs\sys_lib\src\rcg.c:132:41: warning: variable 'frac' set but not used [-Wunused-but-set-variable]
    int fbdiv, refdiv, posdiv2, posdiv1, frac, tmp, tmp_ref;
                                         ^~~~
..\sw_apps\libs\sys_lib\src\rcg.c:132:8: warning: variable 'fbdiv' set but not used [-Wunused-but-set-variable]
    int fbdiv, refdiv, posdiv2, posdiv1, frac, tmp, tmp_ref;
        ^~~~~
{'type': 7, 'id': 67330065, 'description': b'Quad RS232-HS B', 'serial': b'B'}

List of active serial ports:

port:COM3 ; desc:USB Serial Port (COM3) ; hwid:USB VID:PID=0403:6011 SER=6
USB Serial Port (COM3) OK
port:COM4 ; desc:USB Serial Port (COM4) ; hwid:USB VID:PID=0403:6011 SER=6
USB Serial Port (COM4) OK
port:COM5 ; desc:USB Serial Port (COM5) ; hwid:USB VID:PID=0403:6011 SER=6
USB Serial Port (COM5) OK
port:COM6 ; desc:USB Serial Port (COM6) ; hwid:USB VID:PID=0403:6011 SER=6
USB Serial Port (COM6) OK

Found USB Serial Port at COM5
```

Was the chip bonded properly?

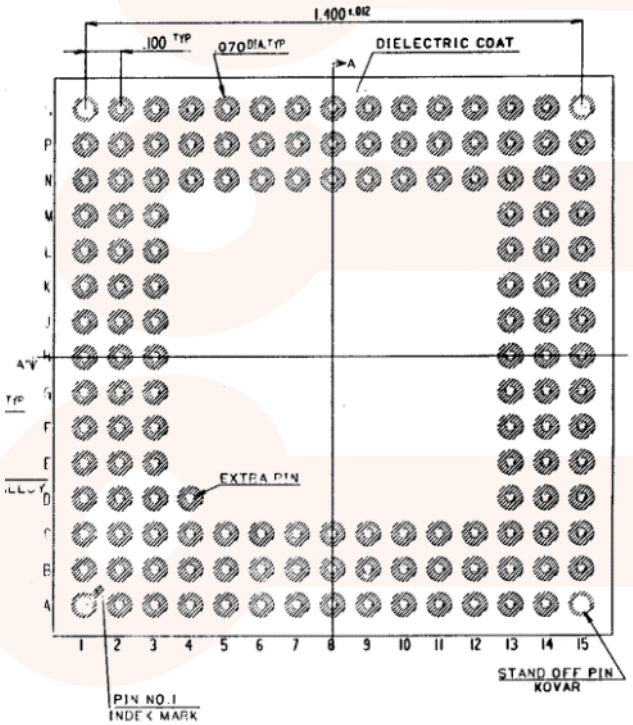
- **One *possible point of failure* is the chip pinout**
  - Chip passes LVS, so we know I/Os are compatible with design.
  - But *how* did we actually check the bonding?
- **Bonding is undoubtedly a potential single point of failure…**
  - We have a spreadsheet mapping I/O pins to package pins to PCB pins
  - We also send a schematic of the connection between the chip and the package

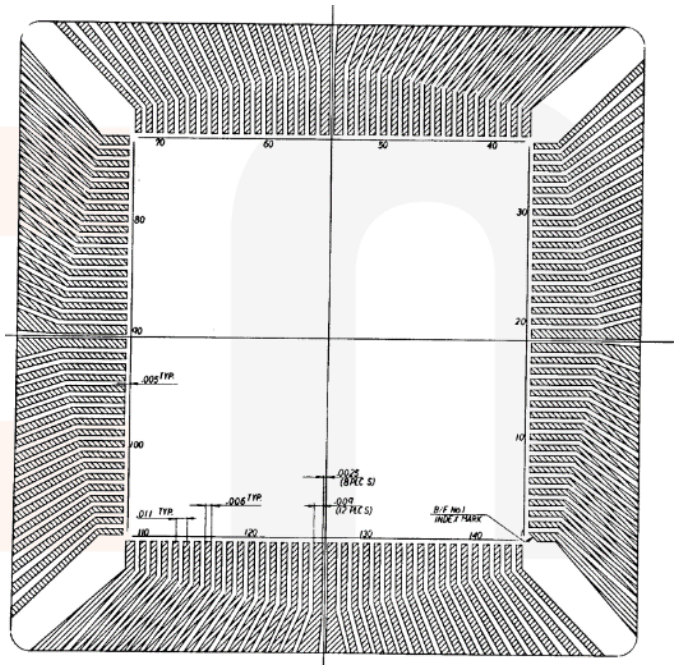
	A	B	C	D
1	DIE PAD #	NET	PCK PIN #	PCB PIN #
2	1	PAD_GPIO_1	1	D3
3	2	PAD_MUL2B_PTL_VDD		
4	3	PAD_MUL2B_DML_VDD		
5	4	PAD_MAC_CMOS_VDD		
6	5	VDDIO	5	E3
7	6	VSS	6	C1
8	7	VDD	7	E2
9	8	VSS	8	D1
10	9	PAD_TDC_CNT_OUT	9	F3
11	10	PAD_AQUA_IN_22	10	F2
12	11	PAD_AQUA_IN_21	11	E1
13	12	PAD_AQUA_IN_11	12	G2
14	13	PAD_RST_N	13	G3
15	14	PAD_AQUA_PLL_CNT_OUT	14	F1
16	15	PAD_TOP_EDRAM_VDD_0	15	G1
17	16	PAD_AQUA_IN_9	16	H2
18	17	VDDIO	17	H1
19	18	VSS	18	H3



We also need the package spec…

- **Not that nice, considering the 144-pin PGA was designed in 1985…**



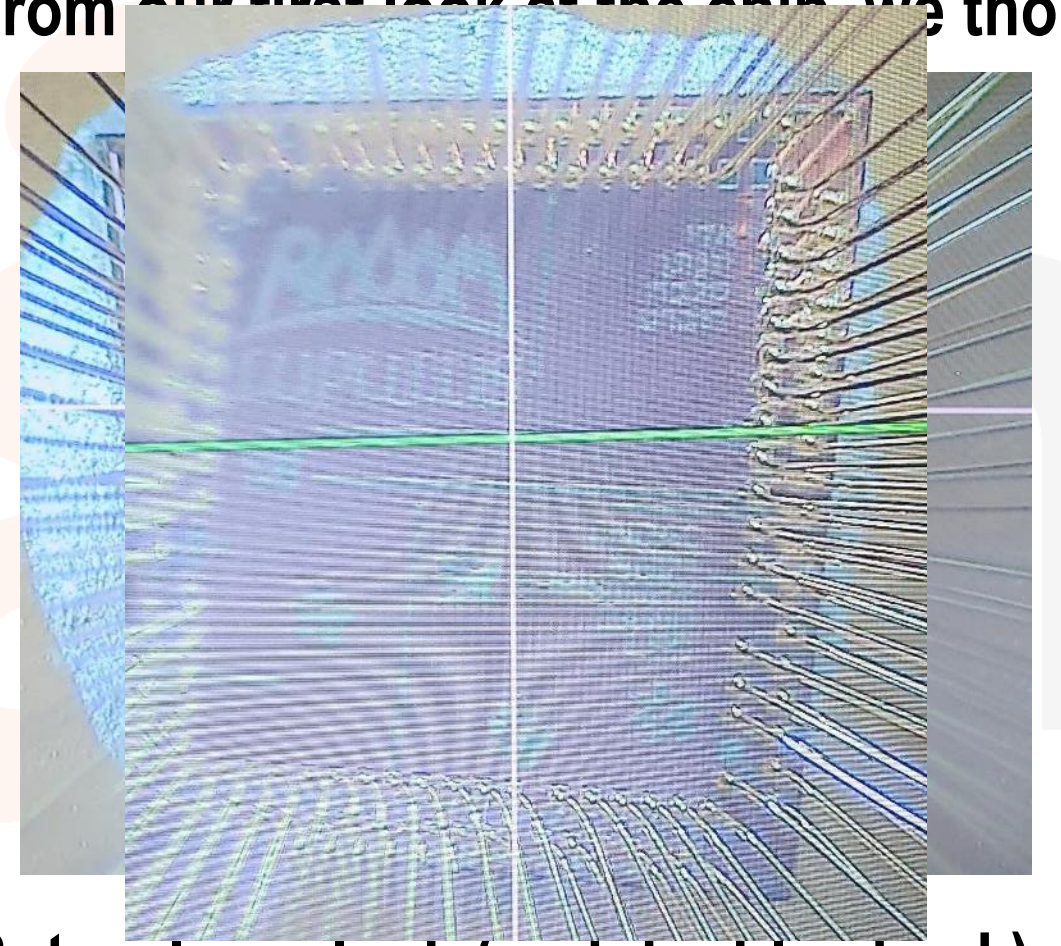


WIRE BOND PAD/CONNECTOR PIN INTERCONNECTION PLAN

BONDING PAD	PIN No.	BONDING PAD	PIN No.	BONDING PAD	PIN No.	BONDING PAD	PIN No.	BONDING PAD	PIN No.	BONDING PAD	PIN No.	BONDING PAD	PIN No.
1	D 3	22	J 2	43	P 5	64	Q 13	85	J 13	106	C 13	127	C 7
2	C 2	23	K 2	44	Q 4	65	P 12	86	K 15	107	B 14	128	A 7
3	B 1	24	K 3	45	N 6	66	N 11	87	J 15	108	A 15	129	A 6
4	D 2	25	L 1	46	P 6	67	P 13	88	H 14	109	C 12	130	B 7
5	E 3	26	L 2	47	Q 5	68	Q 14	89	H 15	110	B 13	131	B 6
6	C 1	27	M 1	48	P 7	69	N 12	90	H 13	111	A 14	132	C 6
7	E 2	28	N 1	49	N 7	70	N 13	91	G 13	112	B 12	133	A 5
8	D 1	29	M 2	50	Q 6	71	P 14	92	G 15	113	C 11	134	B 5
9	F 3	30	L 3	51	Q 7	72	Q 15	93	F 15	114	A 13	135	A 4
10	E 2	31	N 2	52	P 8	73	M 13	94	G 14	115	B 11	136	A 3
11	E 1	32	P 1	53	Q 8	74	N 14	95	F 14	116	A 12	137	B 4
12	G 2	33	M 3	54	N 8	75	P 15	96	F 13	117	C 10	138	C 5
13	G 3	34	N 3	55	N 9	76	M 14	97	E 15	118	B 10	139	B 3
14	F 1	35	P 2	56	Q 9	77	L 13	98	E 14	119	A 11	140	A 2
15	G 1	36	Q 1	57	Q 10	78	N 15	99	D 15	120	B 9	141	C 4
16	H 2	37	N 4	58	P 9	79	L 14	100	C 15	121	C 9	142	C 3
17	H 1	38	P 3	59	P 10	80	M 15	101	D 14	122	A 10	143	B 2
18	H 3	39	Q 2	60	N 10	81	K 13	102	E 13	123	A 9	144	A 1
19	J 3	40	P 4	61	Q 11	82	K 14	103	C 14	124	B 8		
20	J 1	41	N 5	62	P 11	83	L 15	104	B 15	125	A 8		
21	K 1	42	Q 3	63	Q 12	84	J 14	105	D 13	126	C 8		

So, let's try to look at the chip…

- From our first look at the chip, we thought it was rotated

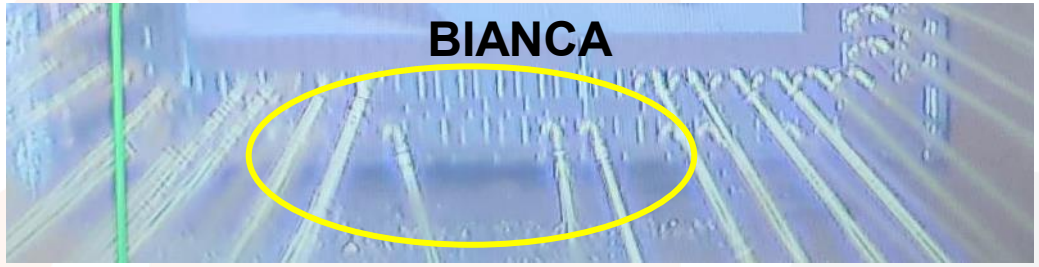


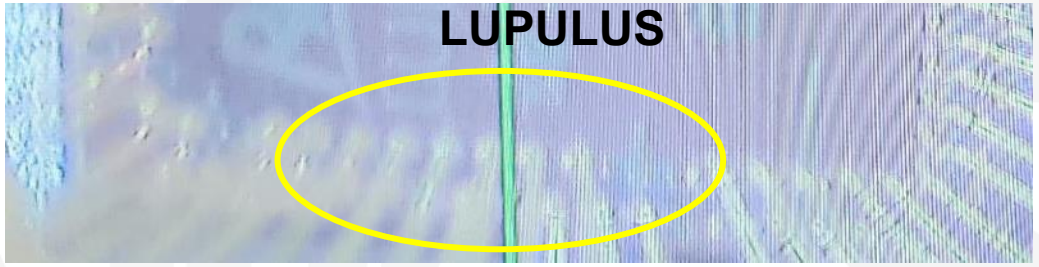
LUPULUS

- But a closer look (and double check) showed that the orientation was correct.

So, we started to compare the two chips

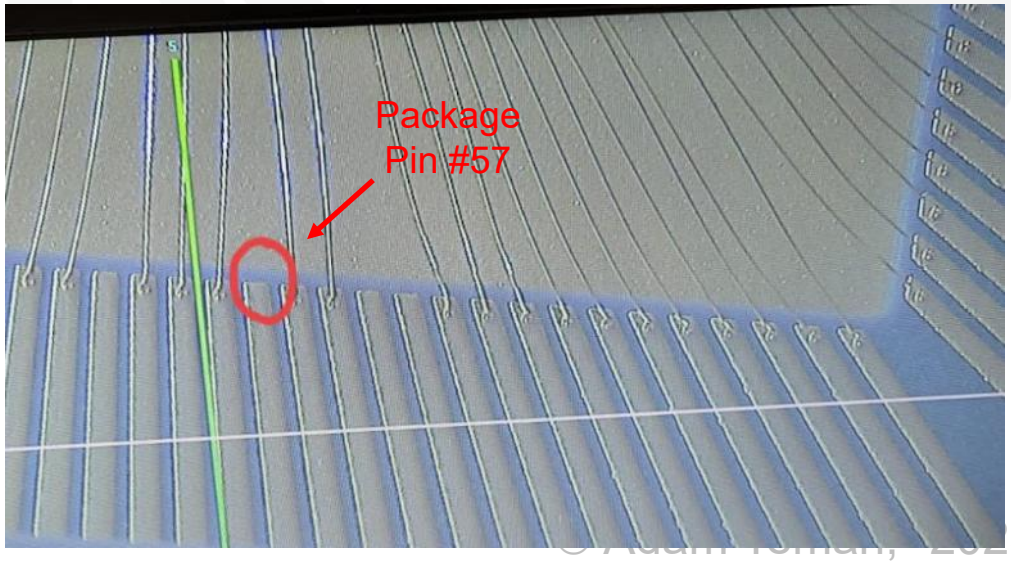
- Something was *very strange* on the bottom edge…





- It looked a little fishy…
- So we dug deeper…
  - Pin 57 isn't connected!
- What is it?
  - **…the clock!**

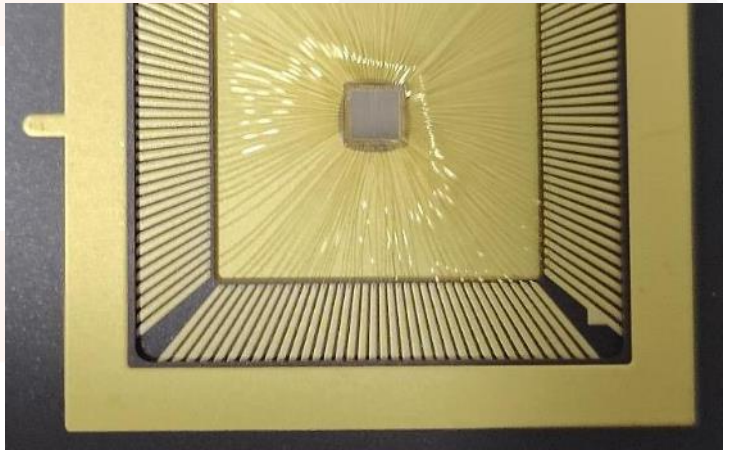
47	PAD_AQUA_IN_3	47	Q5
48	PAD_AQUA_IN_4	48	P7
49	PAD_EXT_40M_CLK	57	Q10
50	PAD_JTAG_TMS	50	Q6
51	PAD_JTAG_TDO	51	Q7
52	PAD_JTAG_TCK	52	P8
53	VDDIO		
54	VSS	54	N8
55	PAD_JTAG_TRSTN	55	N9



We found the culprit. Life is good!

- **Not connecting the clock is a good reason for nothing to work…**
  - Basically, removing the chip from the socket gave the exact same response…
- **We all breathed a sigh of relief and got ready for measurements.**
- **A quick fix at the bond house and everything will be good.**

- **Not so fast!**



- **We got the exact same response from the chip!**

```
..\sw_apps\libs\sys_lib\src\rcg.c:132:8: warning: variable 'fbdiv' set but not use
    int fbdiv, refdiv, posdiv2, posdiv1, frac, tmp, tmp_ref;
        ^~~~~
{'type': 7, 'id': 67330065, 'description': b'Quad RS232-HS B', 'serial': b'B'}

List of active serial ports:

port:COM3 ; desc:USB Serial Port (COM3) ; hwid:USB VID:PID=0403:6011 SER=6
USB Serial Port (COM3) OK
port:COM4 ; desc:USB Serial Port (COM4) ; hwid:USB VID:PID=0403:6011 SER=6
USB Serial Port (COM4) OK
port:COM5 ; desc:USB Serial Port (COM5) ; hwid:USB VID:PID=0403:6011 SER=6
USB Serial Port (COM5) OK
port:COM6 ; desc:USB Serial Port (COM6) ; hwid:USB VID:PID=0403:6011 SER=6
USB Serial Port (COM6) OK

Found USB Serial Port at COM5
```

The Problems Continue

Hello, Nothing

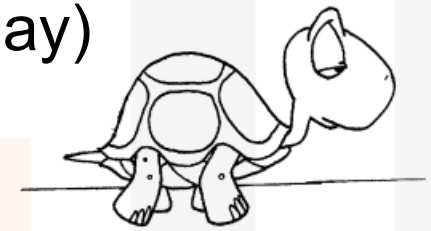
- **So, we know that initially we had no clock.**
  - Obviously, nothing worked.
- **But we double-checked under the microscope and the clock is connected.**
  - So how come it's as if the chip doesn't exist.
- **Okay, there are *some things* that are different now…**
  - Playing with the bootstraps, gets some activity from the chip…
  - For example, we can see that it is trying to fetch code from the MMSPI…
- **But the UART doesn't work, and UART is really, really simple**
  - It's such a slow interface that lowering the frequency wouldn't help…
  - Furthermore, we've used this IP on many chips, so it's not a logic bug…
  - If it's not a bug and not the frequency – could it be a hold violation?

Back to our STA basics

- **Max Delay (Setup)**
  - The data arrives at the capture reg later than the following clock edge:

$$T + \delta_{skew} > t_{cq} + t_{logic} + t_{setup} + \delta_{margin}$$

- **Possible reasons:**
  - Slow corner (long delay)
  - High frequency
  - Negative skew
- **Solution?**
  - Lower the clock frequency

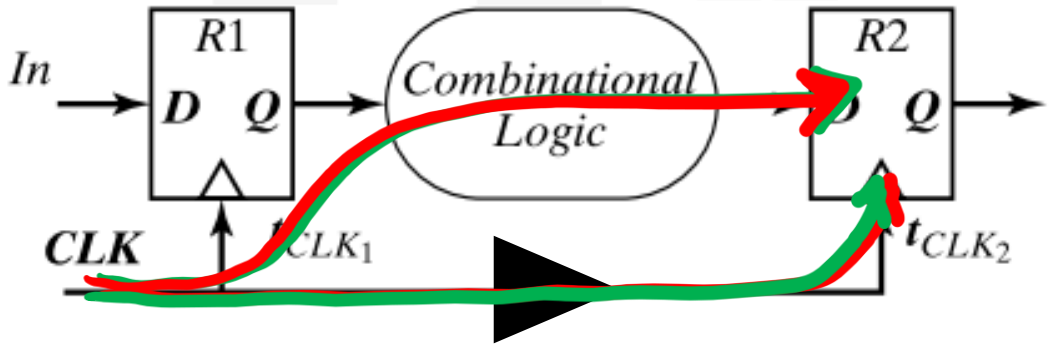


- **Min Delay (Hold)**
  - The data arrives at the capture reg before the same clock edge:

$$t_{cq} + t_{logic} - \delta_{margin} > t_{hold} + \delta_{skew}$$

- **Possible reasons:**
  - Fast corner (short delay)
  - Positive skew

- **Solution?**
  - Throw away the chip?
  - Maybe we can make the chip slower???

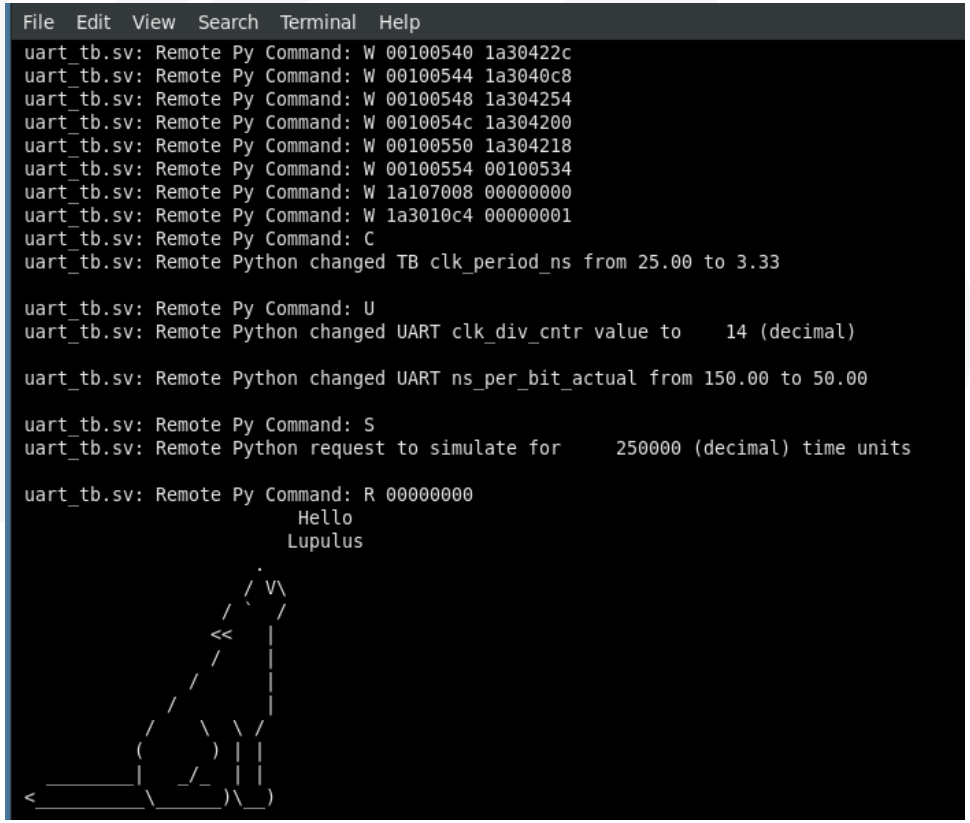


How can we make the chip slower?

- **Remember, PVT = Process, Voltage, Temperature**
  - Process… well, we probably have a typical chip. We could try several, but…
  - Temperature… yeah, we have an oven. We could heat it up (or cool it down…)
  - Voltage… hey, that's pretty easy, let's just lower the voltage!
- **Slightly lowering VDD and something starts happening!**
  - We perform write to several addresses from the UART and can successfully read them back!
  - But when we write to certain addresses, we read back "`deadbeef`"
- **What in the world is "`deadbeef`"?**
  - It's a hexa number that tells us that something isn't right…
  - PulpEnIX responds with `deadbeef` when you access an illegal address…
  - That means… that… something is working… READ is working…

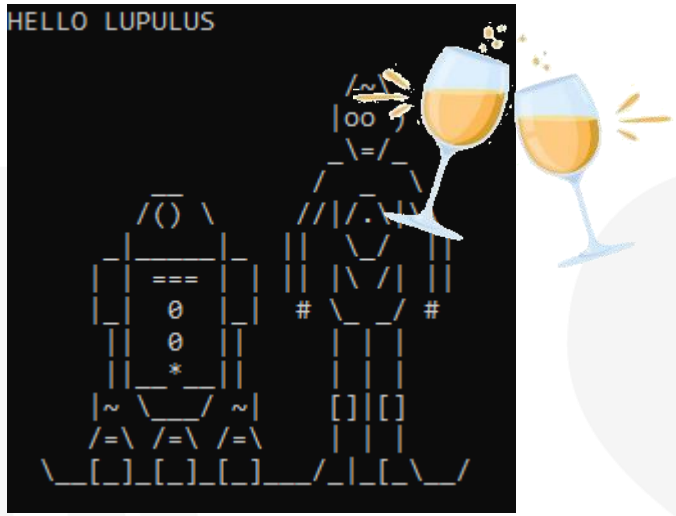
Maybe we changed the memory map?

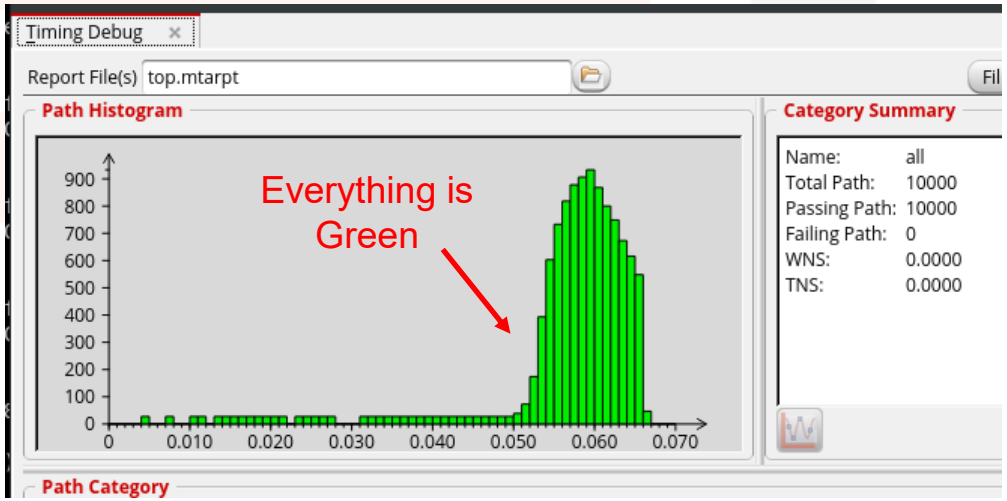
- **Remember our good old "memory map"?**
  - When we design a chip, we give each device a region of addresses.
  - These are hard coded into the chip – the synthesized hardware checks the address according to the defined memory map.
  - This is another *potential point of failure*, since we define it manually (in an Excel sheet…)
- **But, no…**
  - The memory map is the same as in Bianca.
  - Gate-level simulations of booting through UART show everything is okay.

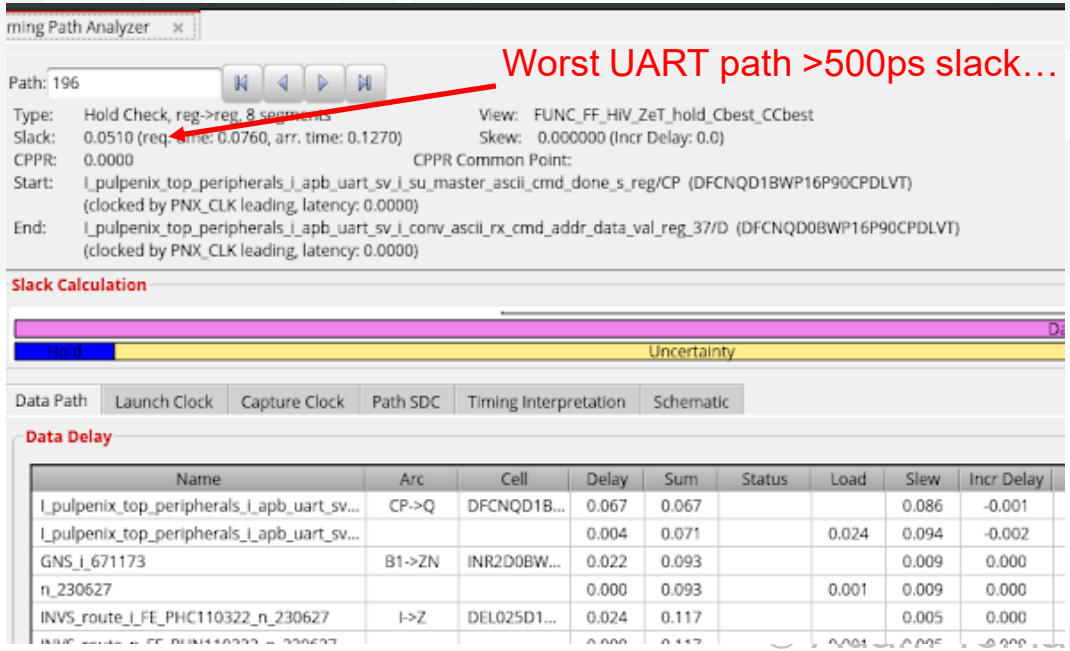


So, it really looks like a hold violation…

- **Let's further reduce the voltage…**
  - More addresses start to work.
- **Even more…**
  - At 0.67V we get a "`Hello, World!`"
- **But it can't be…**
  we checked hold before tapeout…

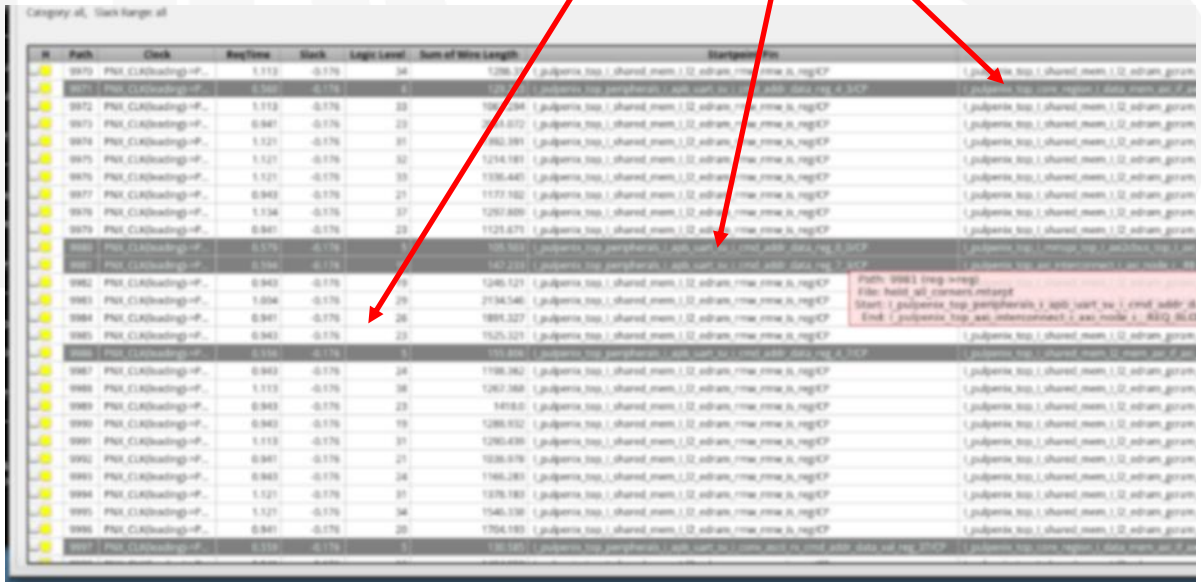






Are you sure you checked all corners?

- **Well, didn't we just say that…?**
  - Worst case setup is in the Slow corner.
  - …and worst-case hold is in the Fast (best case) corner.
- **Yeah, but at nanoscaled technologies, things get weird**
  - During signoff, you have to check hold at *all corners*.
  - …and there are a lot of them!
  - Remember, RCbest, RCworst, etc.



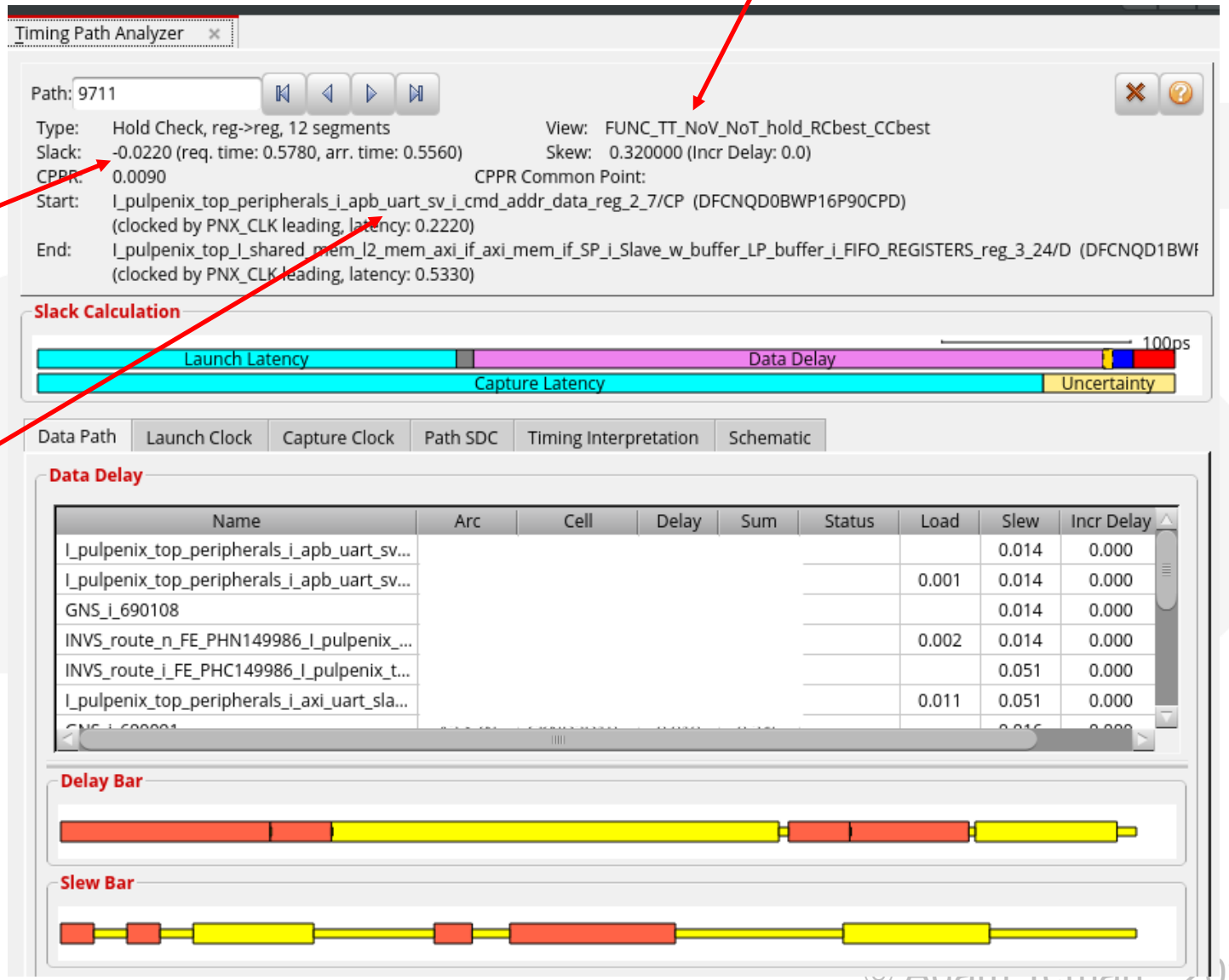
No, no, I mean *all corners*

- **Even… typical?**
  - Yes, even typical…

-22ps negative slack

A path through the UART

TT Corner



- **So, yes… hold is real…**

But, we're not done yet…

Emerging Nanoscaled Integrated Circuits and Systems Labs

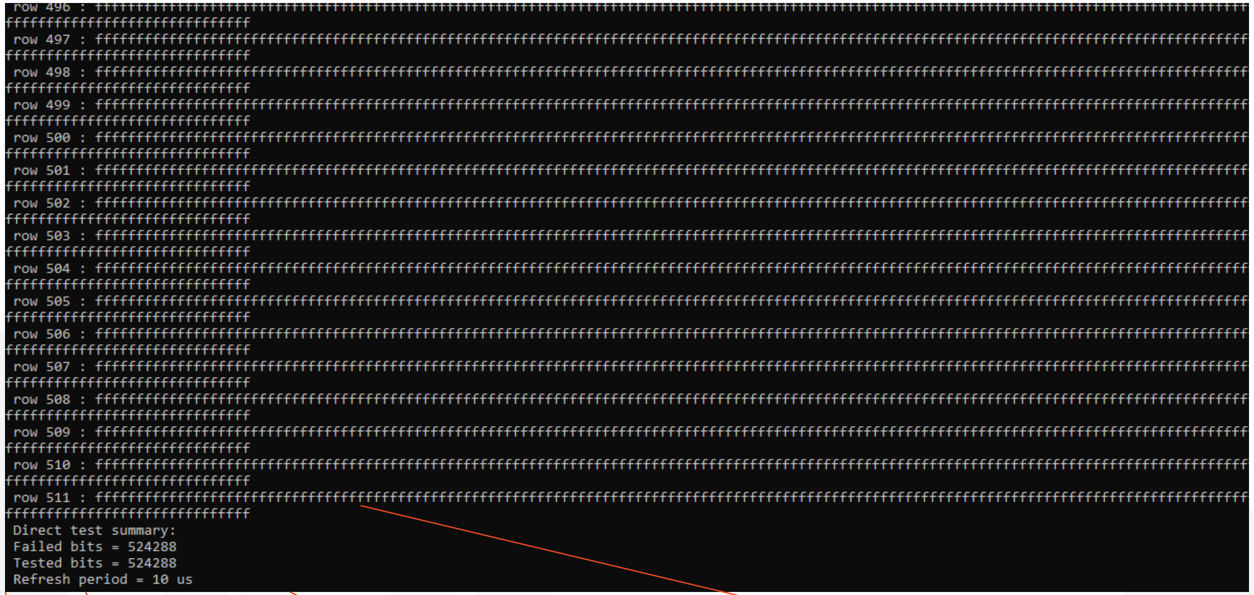
The Alexander Kofkin Faculty of Engineering
Bar-Ilan University

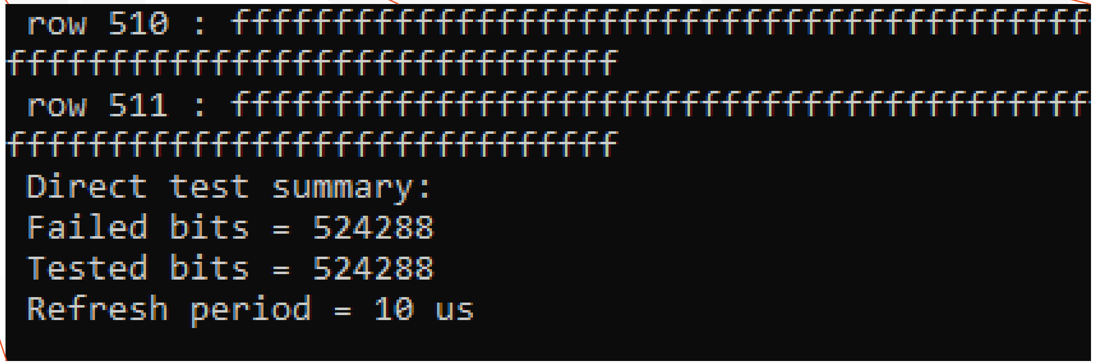
Ready, set, go!

- Clock is connected.
- UART works when voltage is lowered.
- CPU works great.
- What about our memories?

- They don't do anything, of course…
  - Whatever we do, they always read out '1'

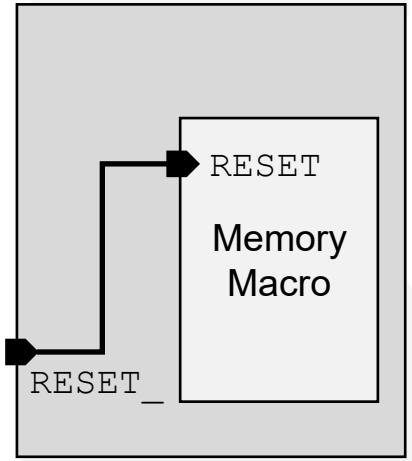
- Ahhhhrrrrggghhhhh!





Reset?

- **After a few days of searching for a needle in a haystack we found the problem.**
- **The memory macro has a `RESET` input**
  - This disconnects the sense amplifiers and outputs a constant '1'…
- **But the SoC has a `RESET_` signal**
  - So, the minute the SoC releases reset, the macro is reset…
- **How did this happen?**
  - Another *potential point of failure* – the Behavioral Model.
  - Logic simulations are run with a Verilog model of the IP that is written by hand.
  - The behavioral model was not equivalent to the analog IP.
- **Solution**
  - Always run (at least basic) mixed-signal verification!

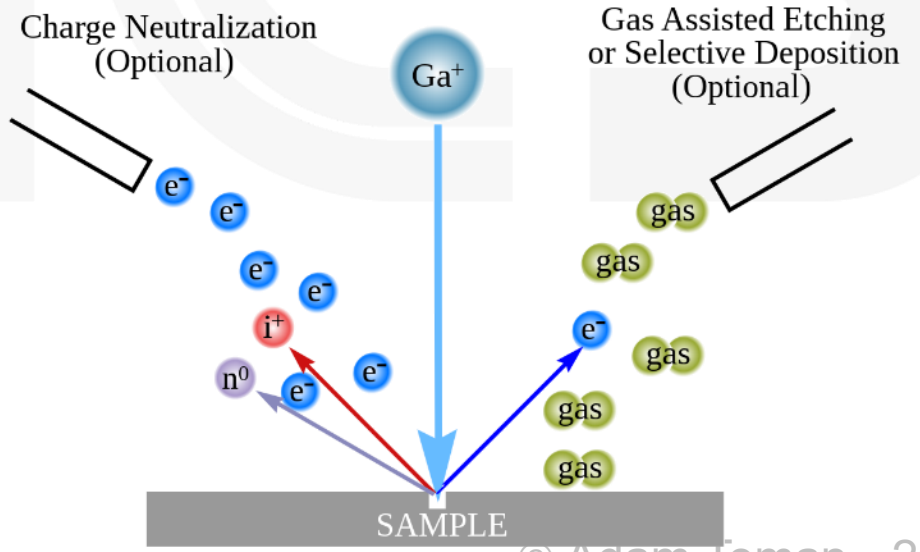



```
module memory(..., RESET_);
   always@(posedge clk or negedge RESET_)
      ...
```

So, the chip is dead…

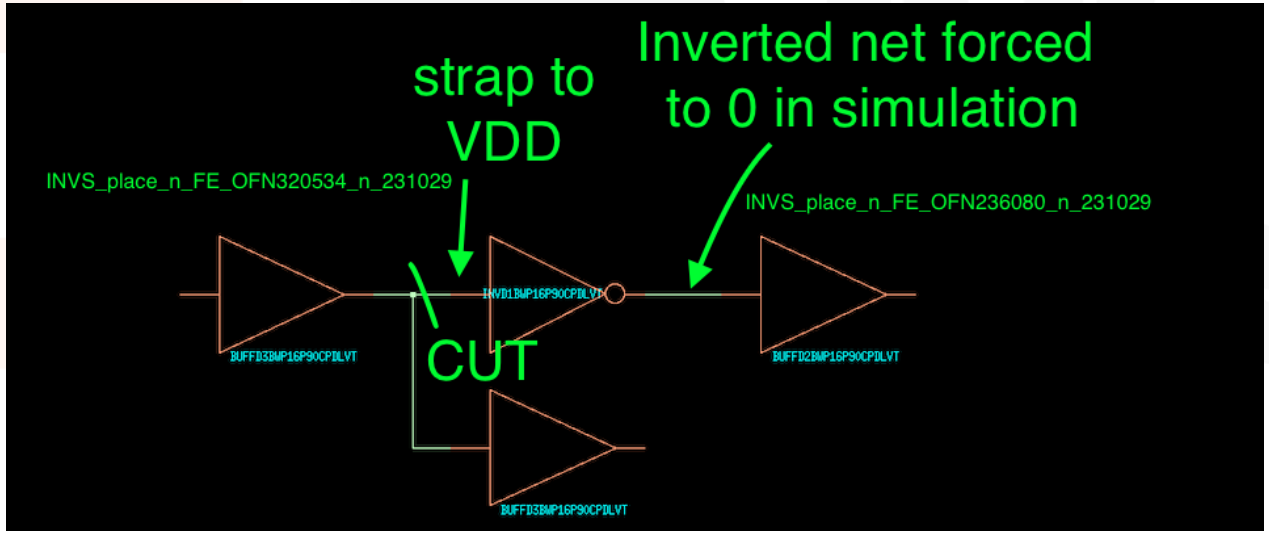
- **Not so fast!**
  - We have one more rabbit up our hat…
- **The FIB!**
  - A Focused Ion Beam is a machine that enables post fabrication circuit editing.
  - At low currents, a FIB enables high resolution imaging (similar to a SEM).
  - At high currents, the FIB can drill down (etch) into the chip with sub micrometer precision.
  - It also can be used to deposit a conductive metal layer to make a new connection.
- **Could we fix the memory with a FIB?**
  - Luckily, Yoav is an expert!

Source: wikipedia

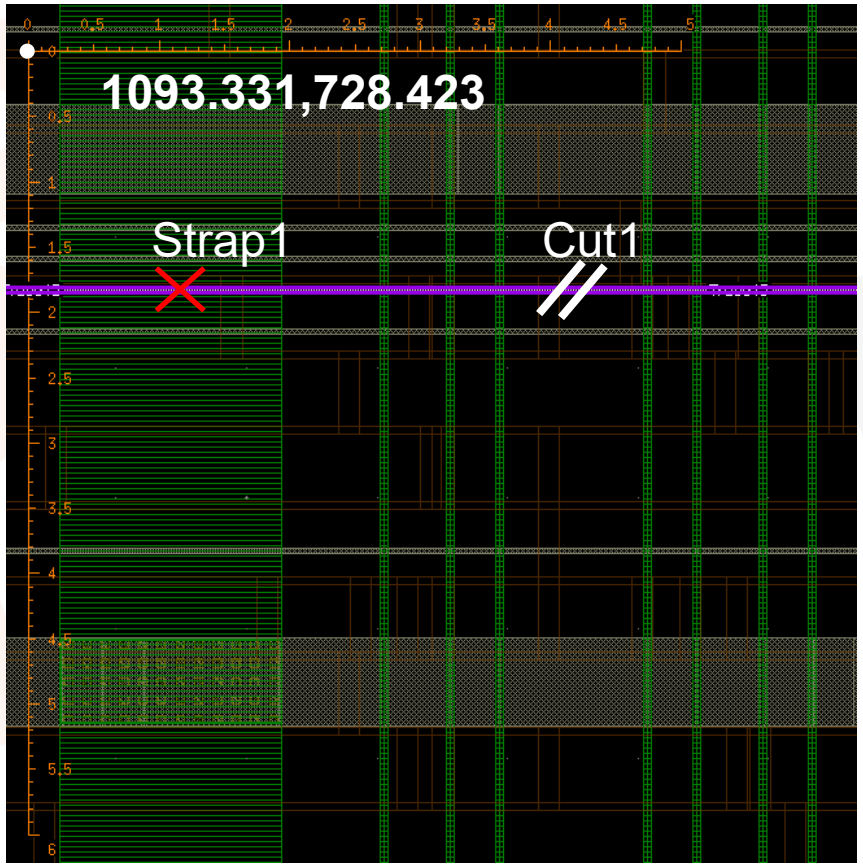


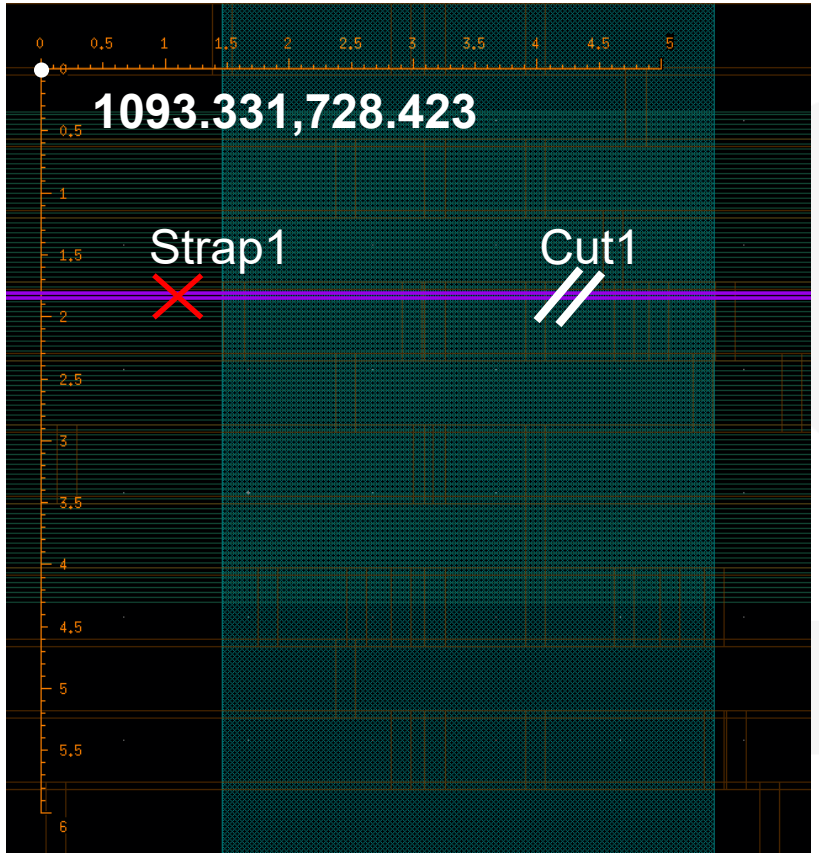
Step 1: Simulate the fix

- Find a reset net that is close to the macro.
- Connect it to VDD.
- Simulate it and see that all is okay.



Step 2: Find a place to drill





M9

M8 Interested Line

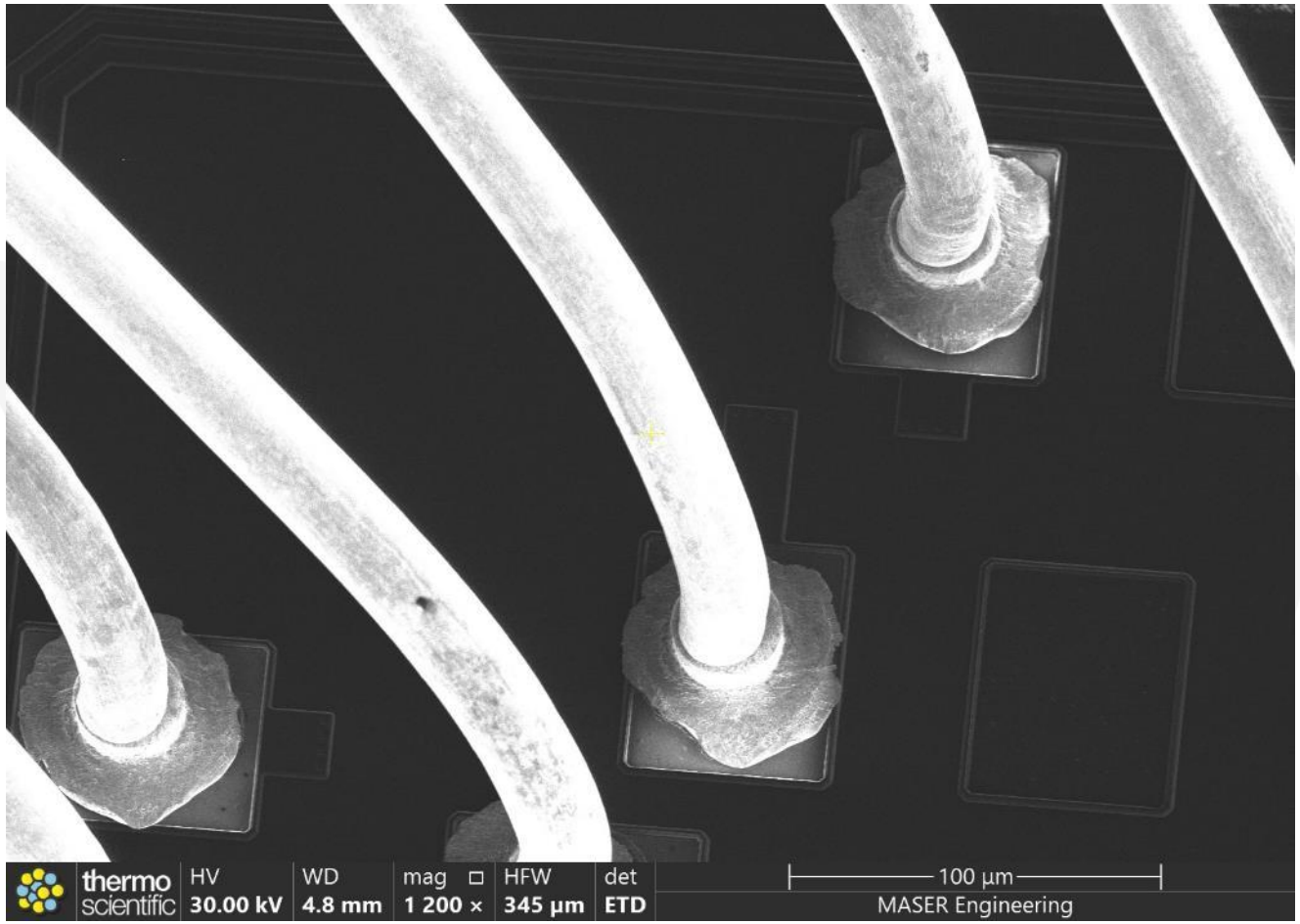
M8

M10 (Over interested line)

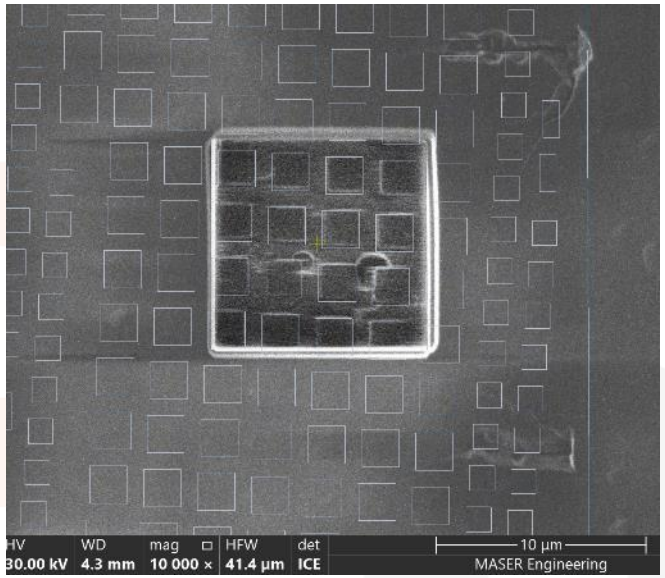
M11

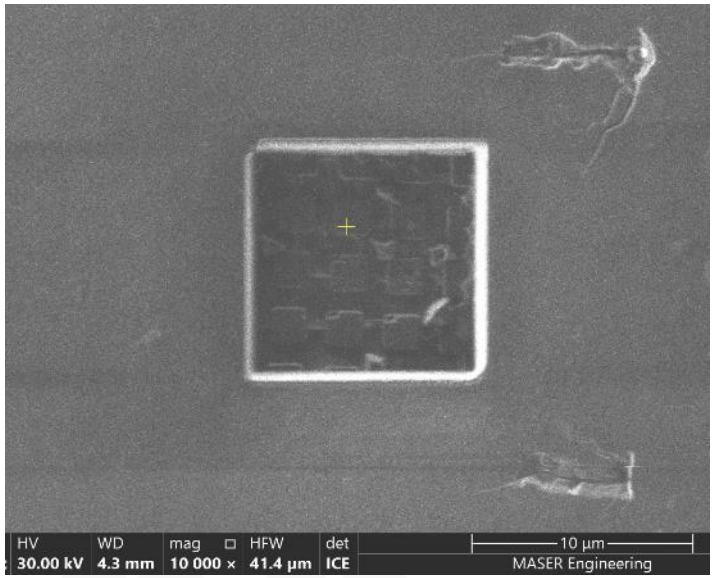
Step 3: Go to the FIB

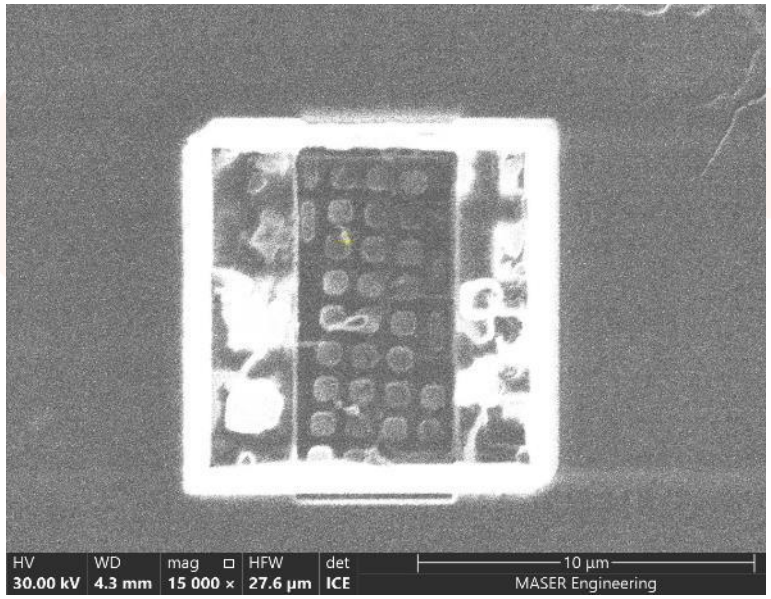
- **It's all or nothing now….**

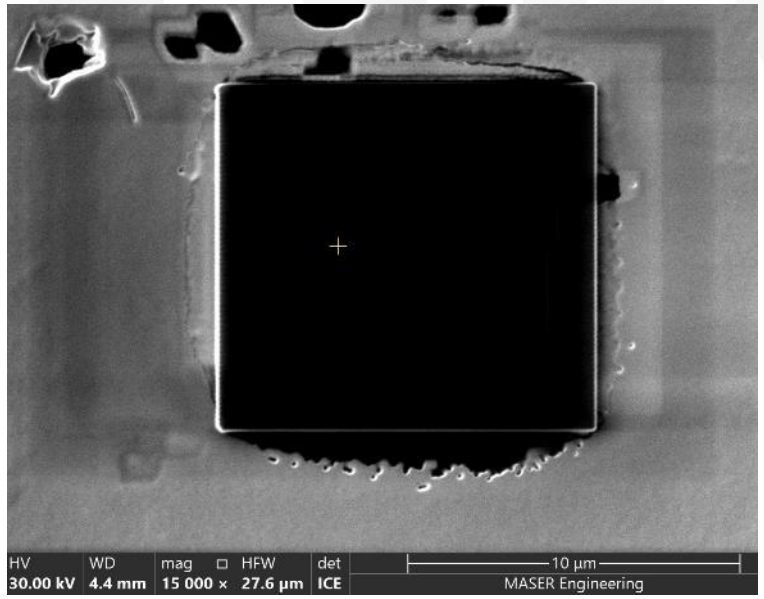


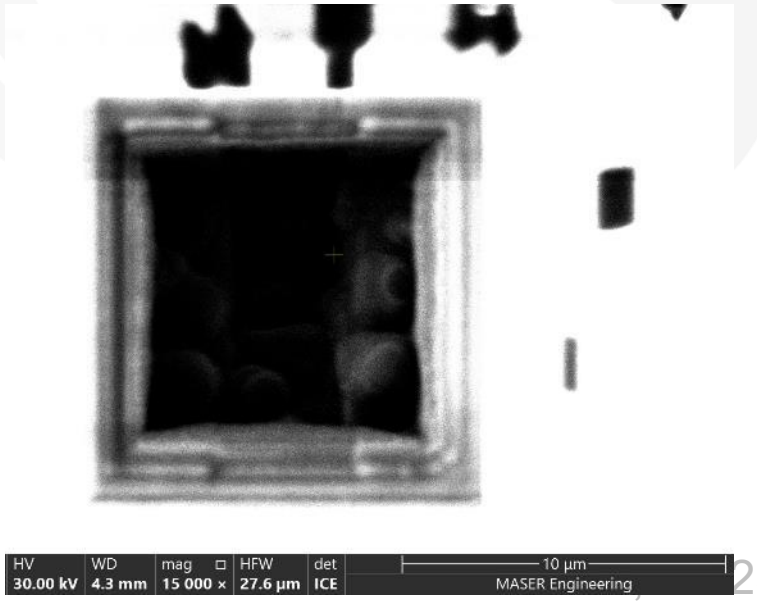
Focusing in on the ROI



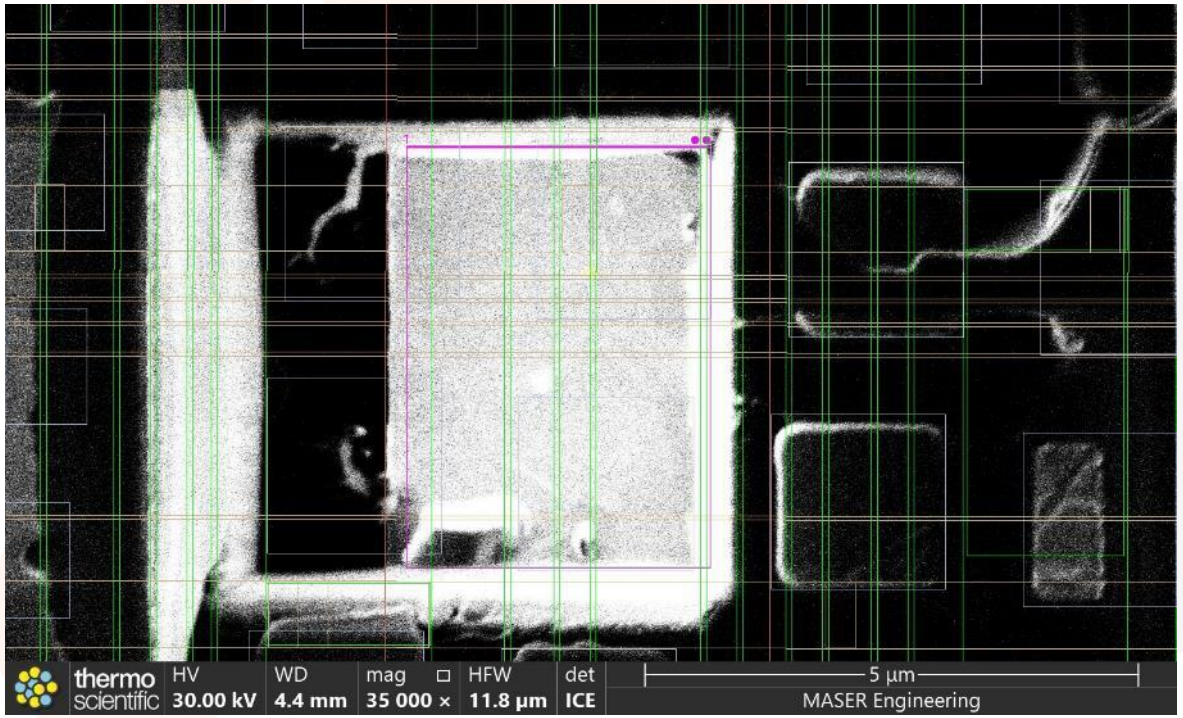


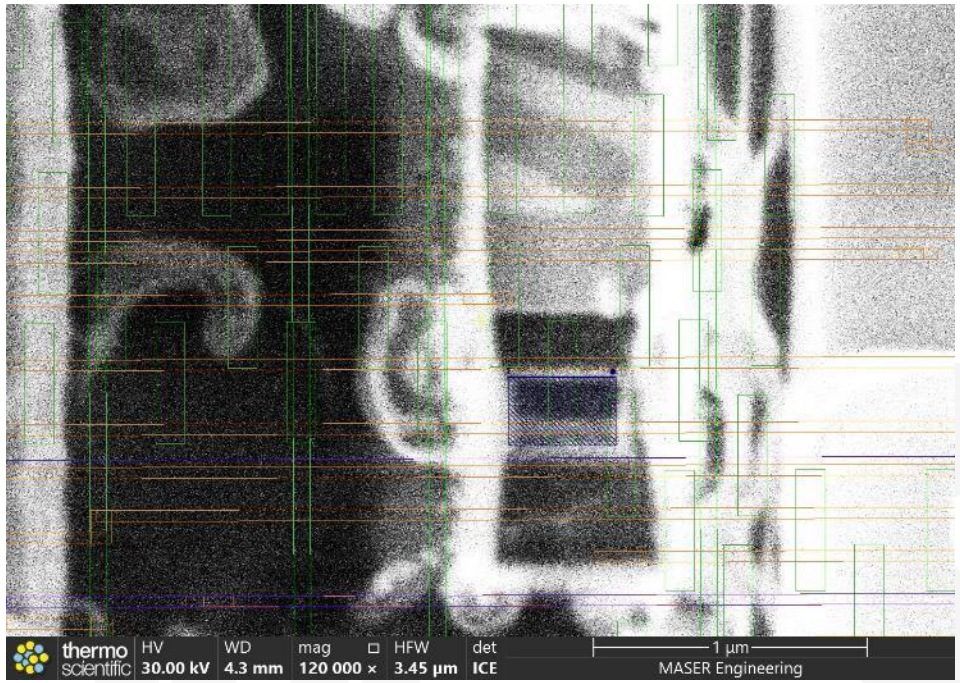


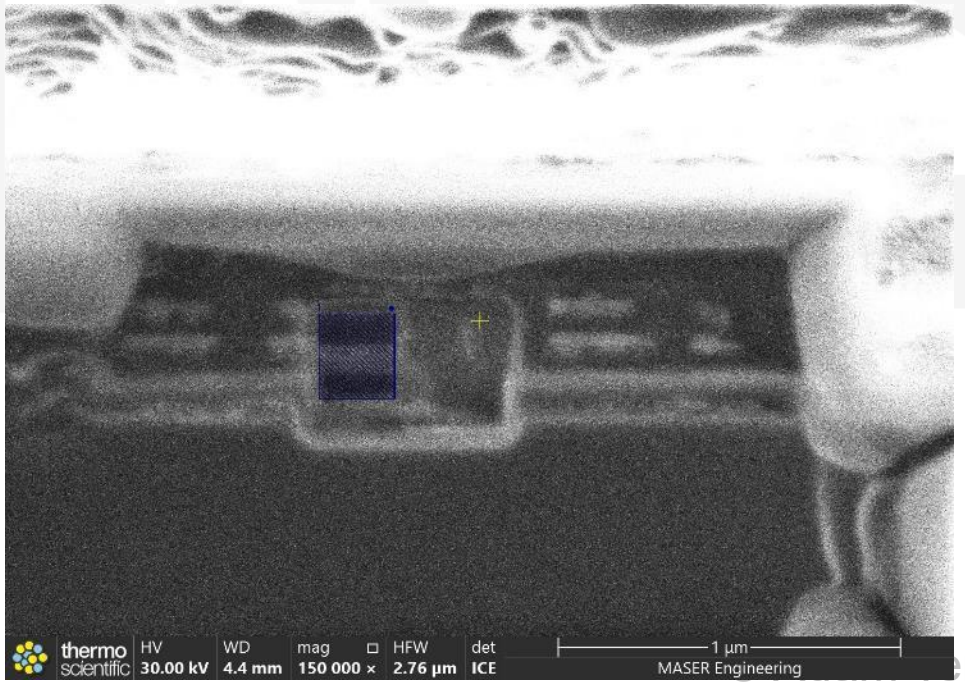




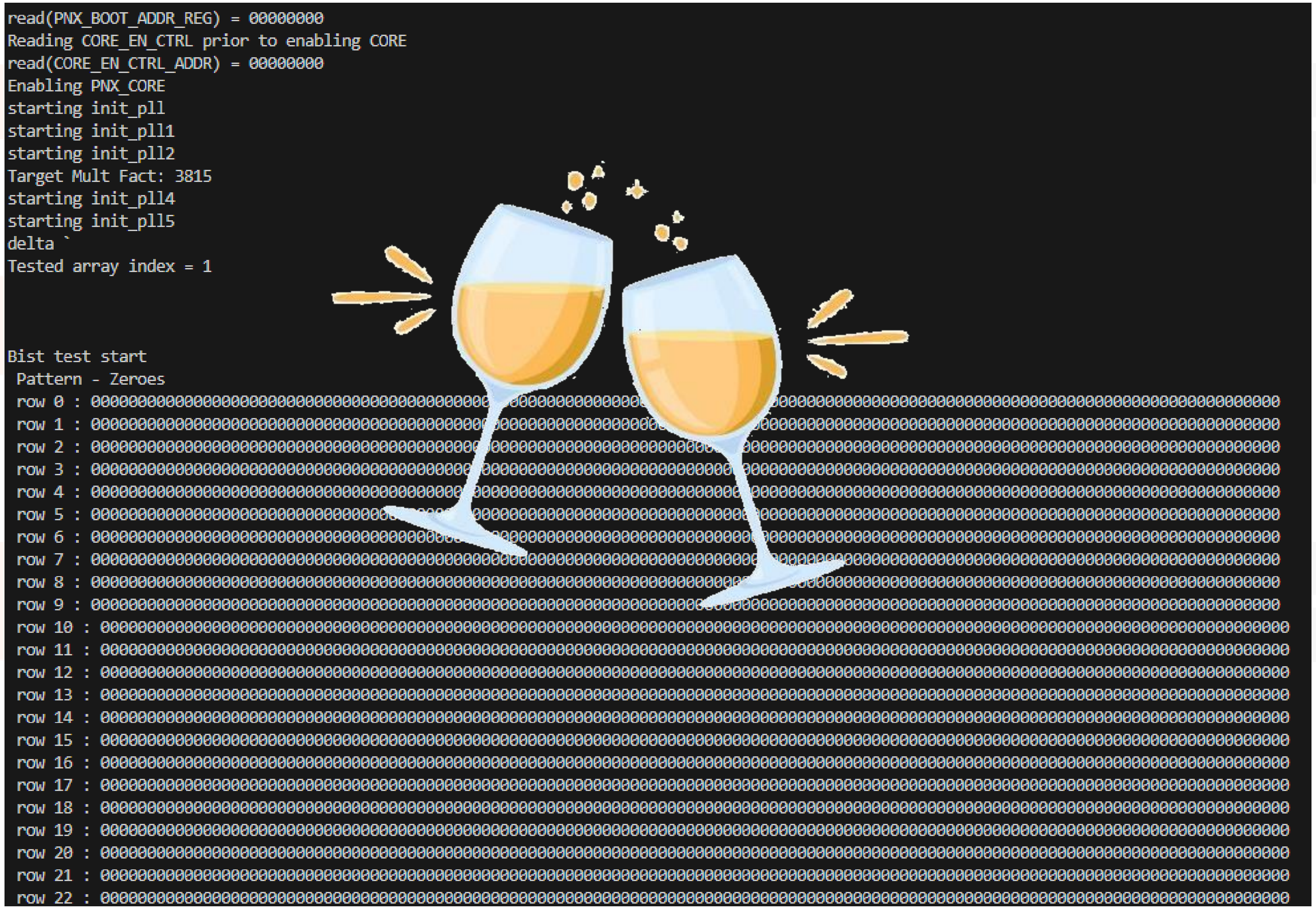
Going deeper







And here goes nothing…



Conclusions

- **Adhere to Signoff guidelines**
  - Signoff checklists are “written in blood”. Don’t take them for granted.
- **Identify potential points of failure**
  - Anything that is not automated and independently verified is a bug waiting to happen.
- **Patience and Persistence**
  - Nothing ever works on the first shot.
  - Don’t give up. There’s almost always another workaround or fix. You just need to find it.
- **Know all the fine details**
  - Nothing “magic” happens in engineering.
  - Debugging is like a mystery. You need all the clues to reach the solution.

Acknowledgements

- This adventure required a lot of help to get to the finish line.

- Thanks to the EnICS Labs, RAAAM Technologies, Eurofins MASER, Beckermus Technologies for their participation in the debugging process.

- Special thanks to Udi Kra, Yoav Weitzman, Yonatan Shoshan and Christoph Mueller.