



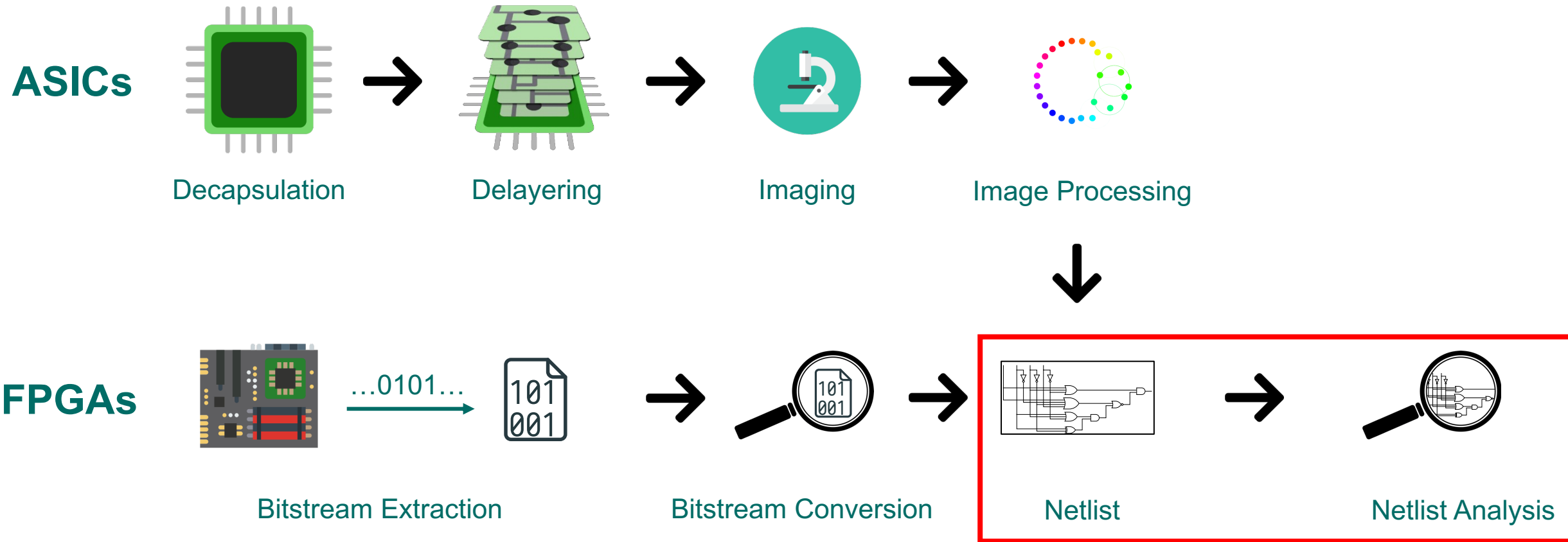
# HAL – A MODULAR FRAMEWORK FOR NETLIST REVERSE ENGINEERING



JULIAN SPEITH  
MPI-SP, BOCHUM, GERMANY



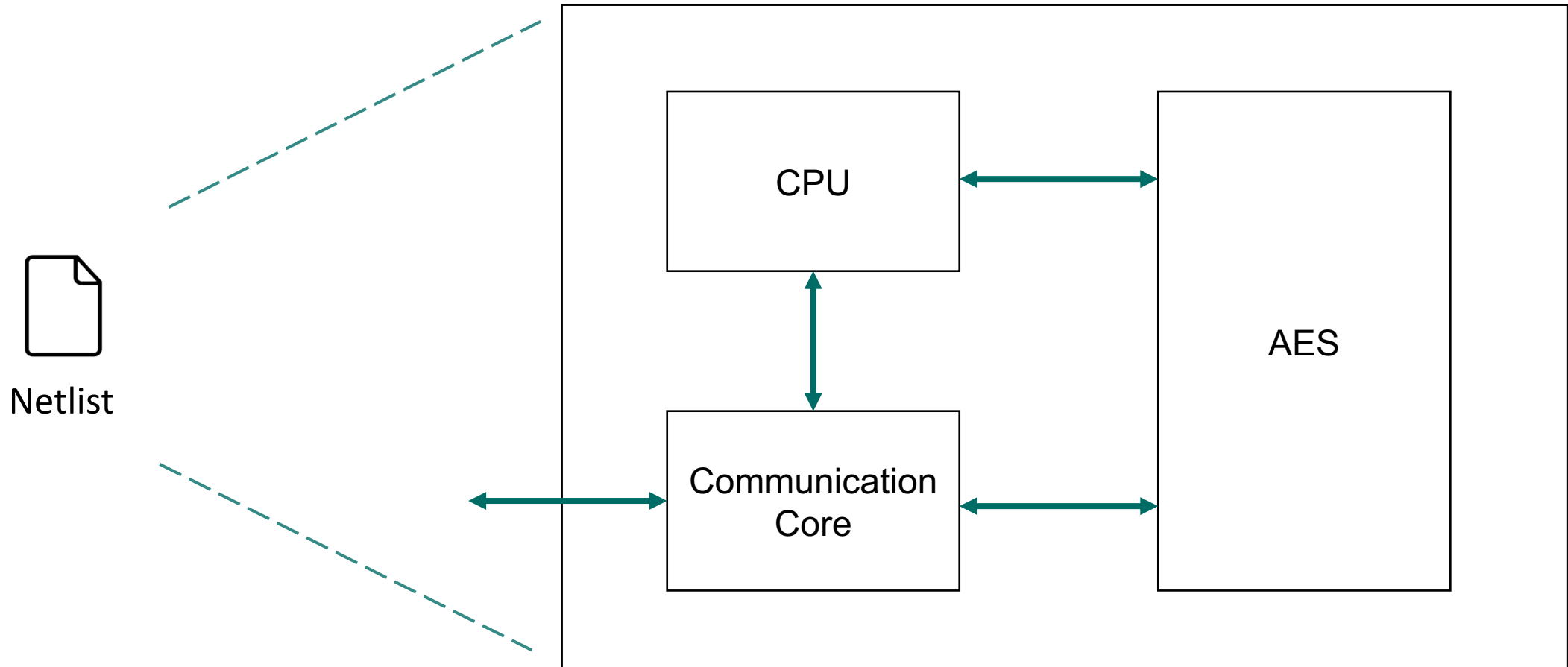
# HARDWARE REVERSE ENGINEERING



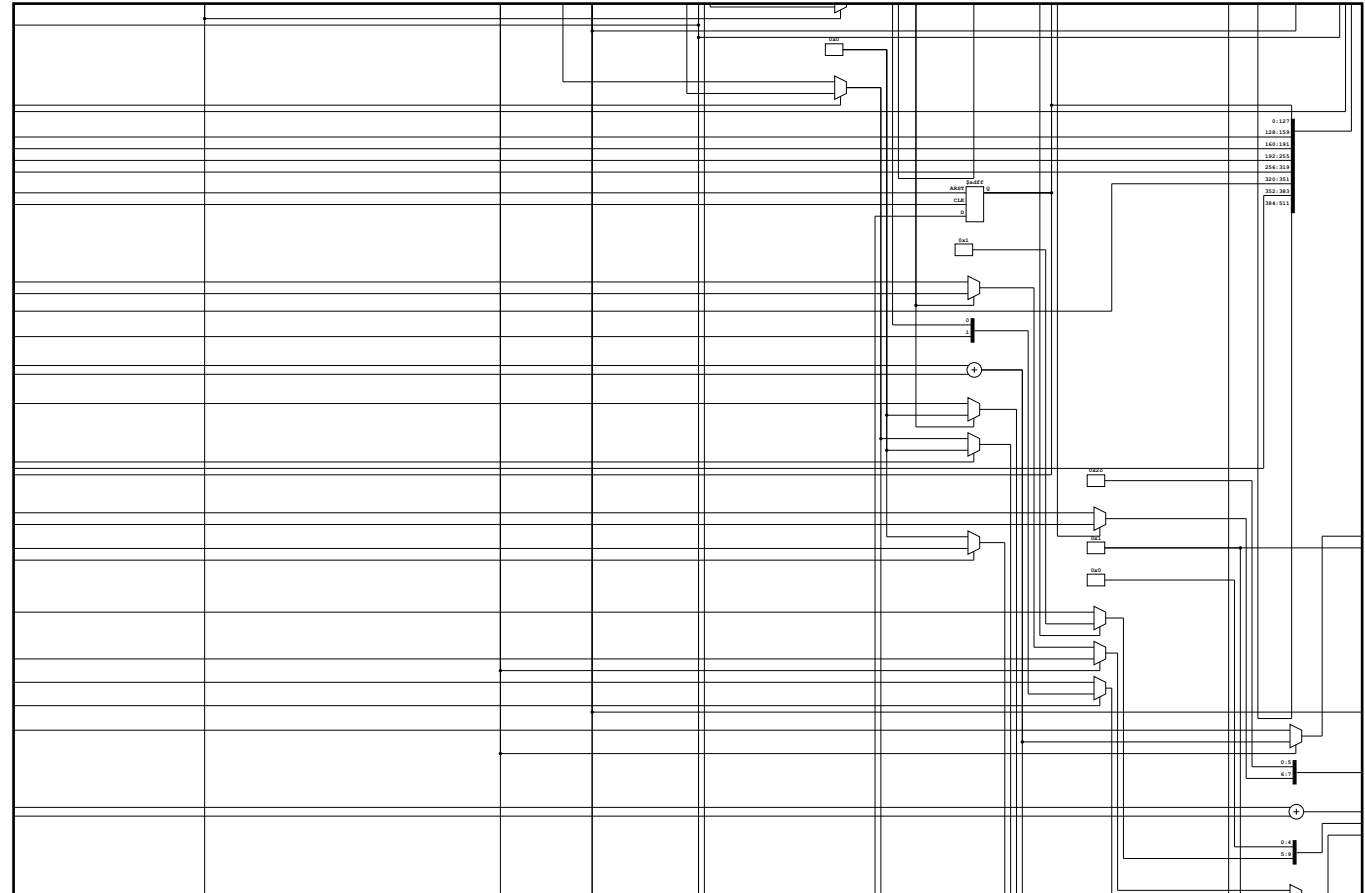


# NETLIST

How you might think it looks like...



## How it really looks like...



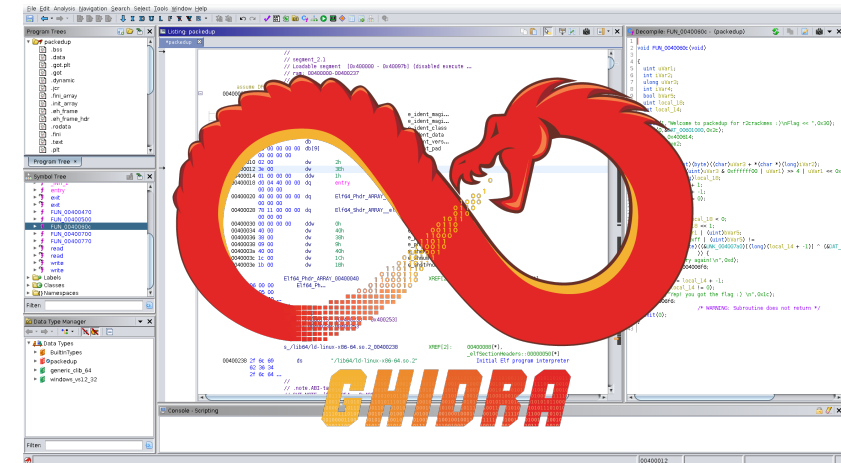
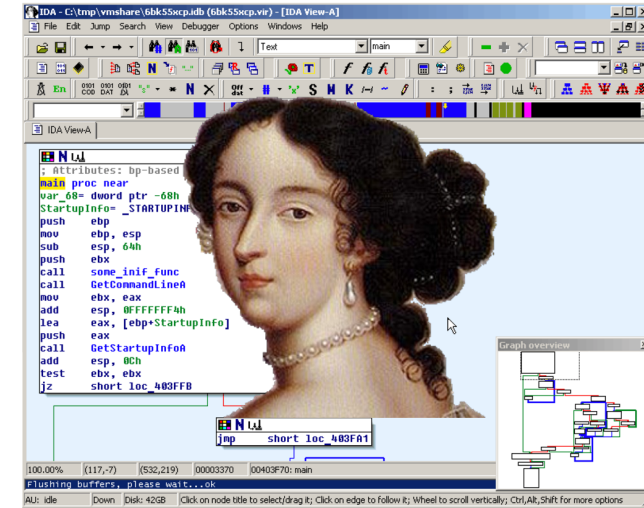


# SOFTWARE REVERSE ENGINEERING

Software RE has a very active community throughout industry and academia

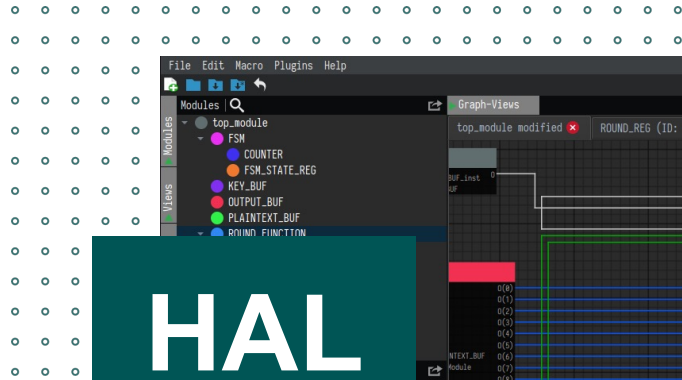
- Popular tools: GHIDRA and IDA Pro
- Advanced frameworks for binary analysis
- Modularity through plugins
- Open-source software (GHIDRA) available

The situation for Hardware RE is different ...









# HAL

The screenshot displays the HAL software interface, which is used for netlist reverse engineering. The main window shows a complex netlist diagram with various modules and their connections. The modules listed include top\_module, FSM, COUNTER, FSM\_STATE\_REG, KEY\_BUF, OUTPUT\_BUF, PLAINTEXT\_BUF, ROUND\_FUNCTION, and SBOX. The netlist diagram shows a hierarchical structure with modules like PLAINTEXT\_BUF, ROUND\_REG, KEY\_BUF, ROUND\_FUNCTION, and SBOX. The Python console on the right shows a script for setting pin names. The bottom panel displays a log of events, including netlist serialization and execution of Python editor content.

**Modules:**

- top\_module
- FSM
- COUNTER
- FSM\_STATE\_REG
- KEY\_BUF
- OUTPUT\_BUF
- PLAINTEXT\_BUF
- ROUND\_FUNCTION

**Netlist Diagram:**

The netlist diagram shows a hierarchical structure with modules like PLAINTEXT\_BUF, ROUND\_REG, KEY\_BUF, ROUND\_FUNCTION, and SBOX. The connections are color-coded: green for data, red for control, and blue for power.

**Python Console:**

```
1 mod = netlist.get_module_by_id(8)
2 pg_name = "0"
3
4 pg = mod.get_pin_group_by_name(pg_name)
5
6 for pin in pg.get_pins():
7     mod.set_pin_name(pin, pg_name + "(" + str(pg.get_index(pin)) + ")")
```

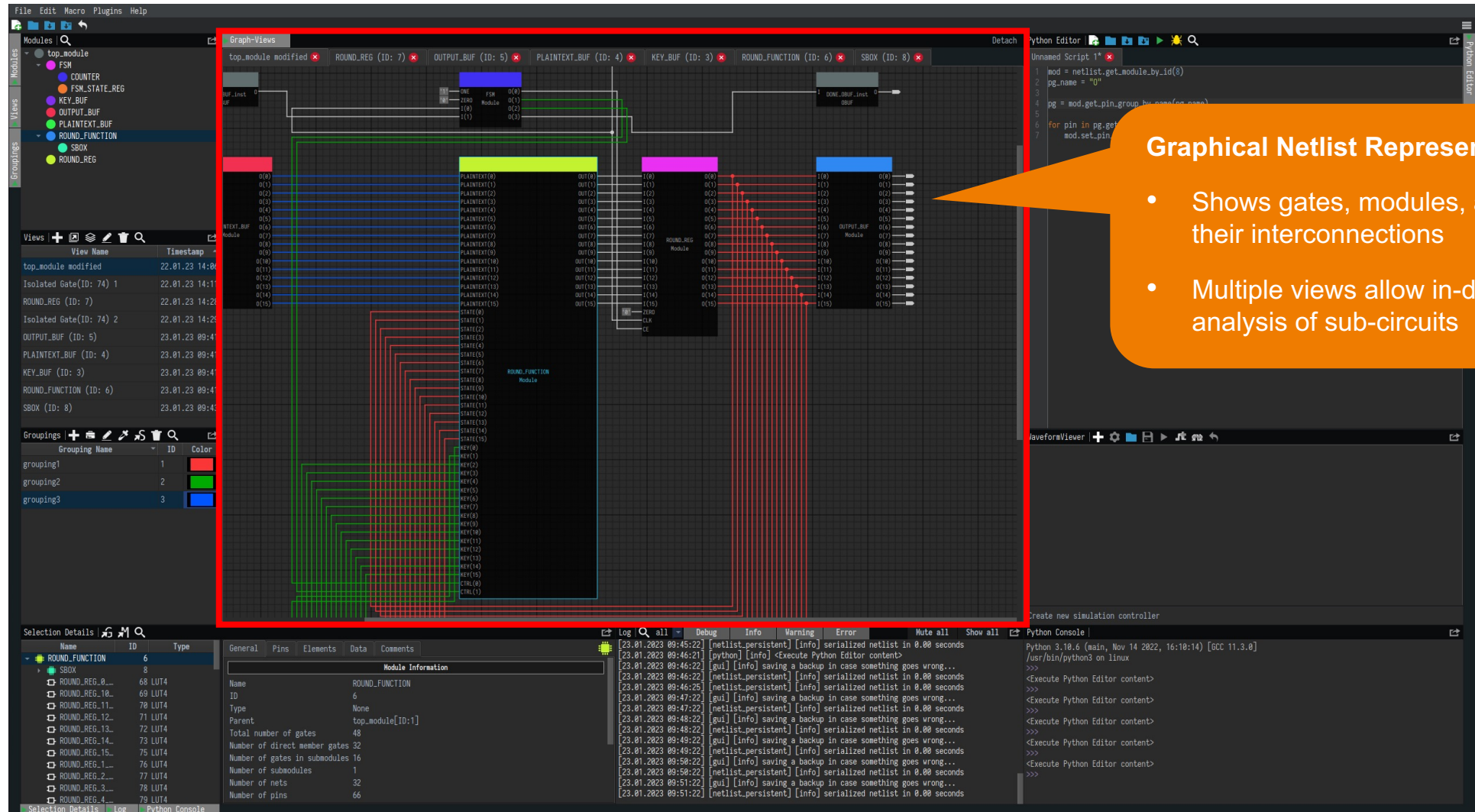
**Log:**

```
[23.01.2023 09:45:22] [netlist-persistent] [info] serialized netlist in 0.00 seconds
[23.01.2023 09:46:21] [python] [info] <Execute Python Editor content>
[23.01.2023 09:46:22] [gui] [info] saving a backup in case something goes wrong...
[23.01.2023 09:46:25] [netlist-persistent] [info] serialized netlist in 0.00 seconds
[23.01.2023 09:47:22] [gui] [info] saving a backup in case something goes wrong...
[23.01.2023 09:47:22] [netlist-persistent] [info] serialized netlist in 0.00 seconds
[23.01.2023 09:48:22] [gui] [info] saving a backup in case something goes wrong...
[23.01.2023 09:48:22] [netlist-persistent] [info] serialized netlist in 0.00 seconds
[23.01.2023 09:49:22] [gui] [info] saving a backup in case something goes wrong...
[23.01.2023 09:49:22] [netlist-persistent] [info] serialized netlist in 0.00 seconds
[23.01.2023 09:50:22] [gui] [info] saving a backup in case something goes wrong...
[23.01.2023 09:50:22] [netlist-persistent] [info] serialized netlist in 0.00 seconds
[23.01.2023 09:51:22] [gui] [info] saving a backup in case something goes wrong...
[23.01.2023 09:51:22] [netlist-persistent] [info] serialized netlist in 0.00 seconds
```

A modular netlist reverse engineering framework written in modern C++.



# HAL – THE HARDWARE ANALYZER



## Graphical Netlist Representation:

- Shows gates, modules, and their interconnections
- Multiple views allow in-depth analysis of sub-circuits





# HAL – THE HARDWARE ANALYZER

## Modules:

- Collections of functionally related gates
- Re-introduce hierarchy to netlist using manual and automated approaches

The screenshot displays the HAL software interface, which is used for hardware analysis. The interface is divided into several panels:

- Modules Panel (Left):** A tree view showing the hierarchy of modules. The 'ROUND\_FUNCTION' module is selected, and its sub-modules are listed: 'FSM', 'FSM\_STATE\_REG', 'KEY\_BUF', 'OUTPUT\_BUF', 'PLAINTEXT\_BUF', 'ROUND\_FUNCTION', 'SBOX', and 'ROUND\_REG'. The 'ROUND\_FUNCTION' module is highlighted in blue.
- Views Panel (Bottom Left):** A table showing the list of modules and their timestamps. The 'ROUND\_FUNCTION' module is selected.
- Groupings Panel (Bottom Left):** A table showing the groupings of modules. The 'ROUND\_FUNCTION' module is selected.
- Netlist Diagram (Center):** A visual representation of the hardware netlist, showing the connections between modules and gates. The 'ROUND\_FUNCTION' module is highlighted in blue.
- Python Editor (Right):** A text editor showing a Python script that interacts with the HAL API. The script defines a module, gets a pin group, and iterates over pins to set their names.
- Python Console (Bottom Right):** A console window showing the output of the Python script, including log messages and error messages.



# HAL – THE HARDWARE ANALYZER

The screenshot displays the HAL software interface. The main window shows a netlist diagram with various modules and their connections. The left sidebar contains a 'Modules' list and a 'Views' panel. The bottom-left panel, titled 'Selection Details', is highlighted with a red box and contains the following information:

Name	ID	Type
ROUND_FUNCTION	6	
SBOX	8	
ROUND_REG_0...	68	LUT4
ROUND_REG_10...	69	LUT4
ROUND_REG_11...	70	LUT4
ROUND_REG_12...	71	LUT4
ROUND_REG_13...	72	LUT4
ROUND_REG_14...	73	LUT4
ROUND_REG_15...	75	LUT4
ROUND_REG_1...	76	LUT4
ROUND_REG_2...	77	LUT4
ROUND_REG_3...	78	LUT4
ROUND_REG_4...	79	LUT4

The 'Module Information' panel for the selected 'ROUND\_FUNCTION' module shows the following details:

Property	Value
Name	ROUND_FUNCTION
ID	6
Type	None
Parent	top_module[ID:1]
Total number of gates	48
Number of direct number gates	32
Number of gates in submodules	16
Number of submodules	1
Number of nats	32
Number of pins	66

An orange callout box points to the 'Selection Details' panel with the text: 'Gate/Module/Net Details: Shows details on current selection. Name, type, pins, Boolean functions, ...'



# HAL – THE HARDWARE ANALYZER

The screenshot displays the HAL software interface. On the left, a 'Modules' panel lists components like FSF, FSF\_STATE\_REG, KEY\_BUF, OUTPUT\_BUF, PLAINTEXT\_BUF, ROUND\_FUNCTION, SBOX, and ROUND\_REG. Below it, a 'Views' panel shows a list of views with timestamps. The main area is a netlist diagram with various modules and connections. On the right, a 'Python Editor' window is open, showing a script for interacting with the netlist. An orange callout box highlights the 'Python Scripting' section, which lists two key features: a powerful Python API to interact with the netlist and the ability to prototype detection algorithms and quick experiments. At the bottom, a 'Python Console' window shows the execution of the script, with logs indicating successful serialization and execution of the Python code.

**Python Scripting:**

- Powerful Python API to interact with the netlist
- Allows prototyping of detection algorithms and quick experiments

```
1 mod = netlist.get_module_by_id(6)
2 pg_name = "Q"
3
4 pg = mod.get_pin_group_by_name(pg_name)
5
6 for pin in pg.get_pins():
7     mod.set_pin_name(pin, pg_name + "(" + str(pg.get_index(pin)) + ")")
```





# HAL – THE HARDWARE ANALYZER

**Netlist Simulation:**

- Integrated waveform viewer with deep integration into the graph view
- More on this in the next talk

The screenshot shows the HAL software interface. The main window displays a netlist simulation graph with various modules and their connections. A red box highlights the 'WaveformViewer' window, which is currently empty. An orange speech bubble points to the waveform viewer with the text 'Netlist Simulation: Integrated waveform viewer with deep integration into the graph view' and 'More on this in the next talk'. The bottom of the interface shows a log window with debug messages and a Python console.

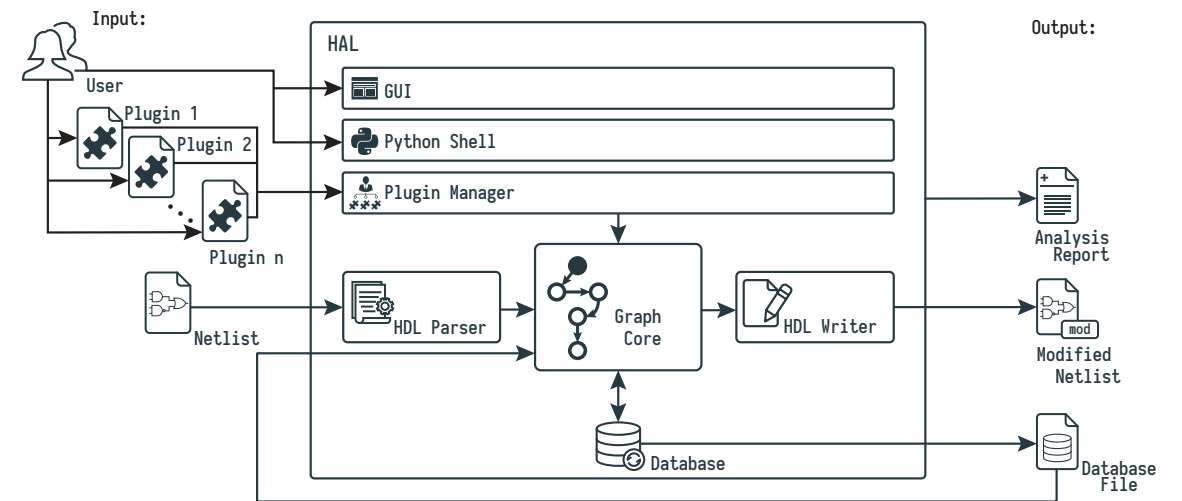




# UNDER THE HOOD – CORE

## Provides Functionalities for Recurring Tasks

- Creation and management of HAL projects
- Netlist and gate library parsing and writing
- Netlist interaction and manipulation
  - Interpretation of the netlist as a graph
  - Allows easy traversal of the netlist
  - Enables execution of graph algorithms
- Boolean function handling and evaluation
- Module creation and management





# UNDER THE HOOD – PLUGIN SYSTEM

## C++ API allows to customize and extend HAL functionality

- The HAL core can be extended according to your requirements using C++ plugins
- Allows you to keep plugins private while still running an up-to-date HAL
- Enables interacting with the netlist, running automated detection algorithms, ...
- (Experimental) GUI support for plugin functionality
  - Add your plugin functions to context menus and the toolbar
- **Already available:** Dataflow analysis (DANA), Simulator, Pre-Processing, Parsers & Writers, ...



# UNDER THE HOOD – BOOLEAN FUNCTIONS

## Out-of-the-Box Support for Multi-Bit Boolean Functions

- Allows for functional analysis of word-level combinational sub-circuits
- Utility functions to get Boolean functions of sub-graphs
- Interpretation of LUT configurations and translation to Boolean functions
- Support for
  - Arithmetic operations: AND, OR, XOR, NOT, ADD, SUB, MUL ...
  - Comparison operations: EQ, ULT, SLT, ULE, SLE
  - Slice, Concat, zero extension, sign extension



# UNDER THE HOOD – SMT SOLVING

## Native Support for SMT Solving

- Allow to check functional equivalence of Boolean functions
  - **Example:** verify whether a multi-bit function implements an addition
- HAL API enables interfacing with arbitrary SMT solvers
  - Supported by default: z3, Boolector, Bitwuzla
- No need for conversion of Boolean functions, **HAL takes care**





# WHAT HAL CAN AND CANNOT DO FOR YOU



## Framework to provide functionality to analyze netlists

- Parsing of netlists and gate libraries
- Core features provides functions to interact with netlist
- Boolean function support (including SMT/SAT integration)
- Simulation support
- Plugin system to automate certain tasks



## HAL is not a one-click solution

- HAL provides functions to analyze netlist, but does not do the analysis for you



Any Questions?