hwdbg

Debugging Hardware Like Software

Sina Karvandi
Spring 2024

“To err is human,
to debug is divine.”

hwdbg

HyperDbg's chip-level debugger

CIRCT Scala

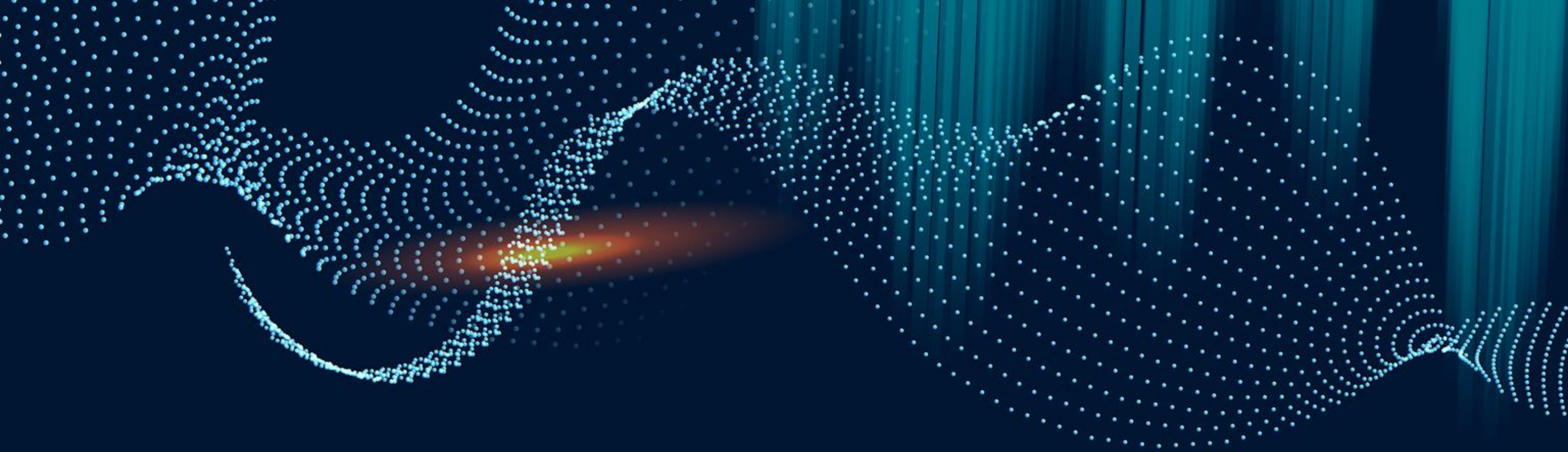
CHISEL

hwdbg is a new debugger chip generator based on chisel language (scala) & LLVM circt

hwdbg.hyperdbg.org/docs

Outline

Outline



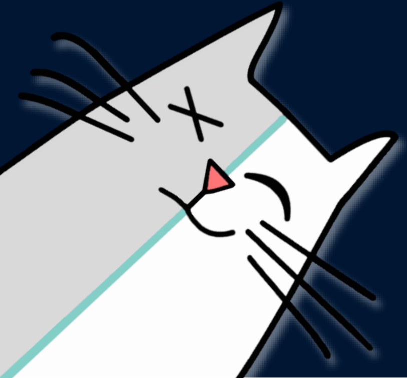
01

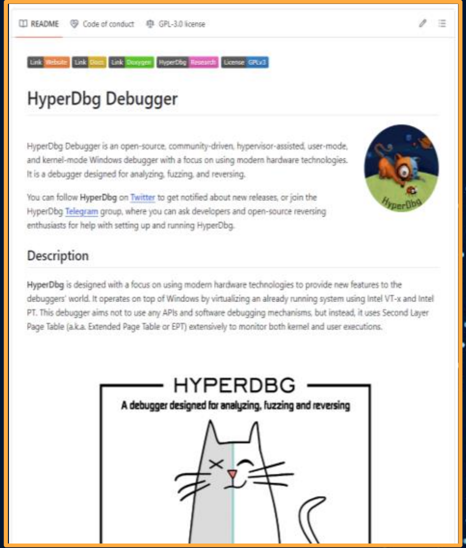
Introduction

The motivation behind creating a new debugger

What is HyperDbg?

HyperDbg is an open-source platform that brings debugging infrastructures like hypervisor-assisted, user-mode, and kernel-mode debuggers mainly with a focus on using modern hardware technologies for **analyzing**, **fuzzing**, and **reversing**.





README · Code of conduct · GPL-3.0 license

Link Website · Link Docs · Link Doxygen · HyperDbg Research · License GPLv3

HyperDbg Debugger

HyperDbg Debugger is an open-source, community-driven, hypervisor-assisted, user-mode, and kernel-mode Windows debugger with a focus on using modern hardware technologies. It is a debugger designed for analyzing, fuzzing, and reversing.

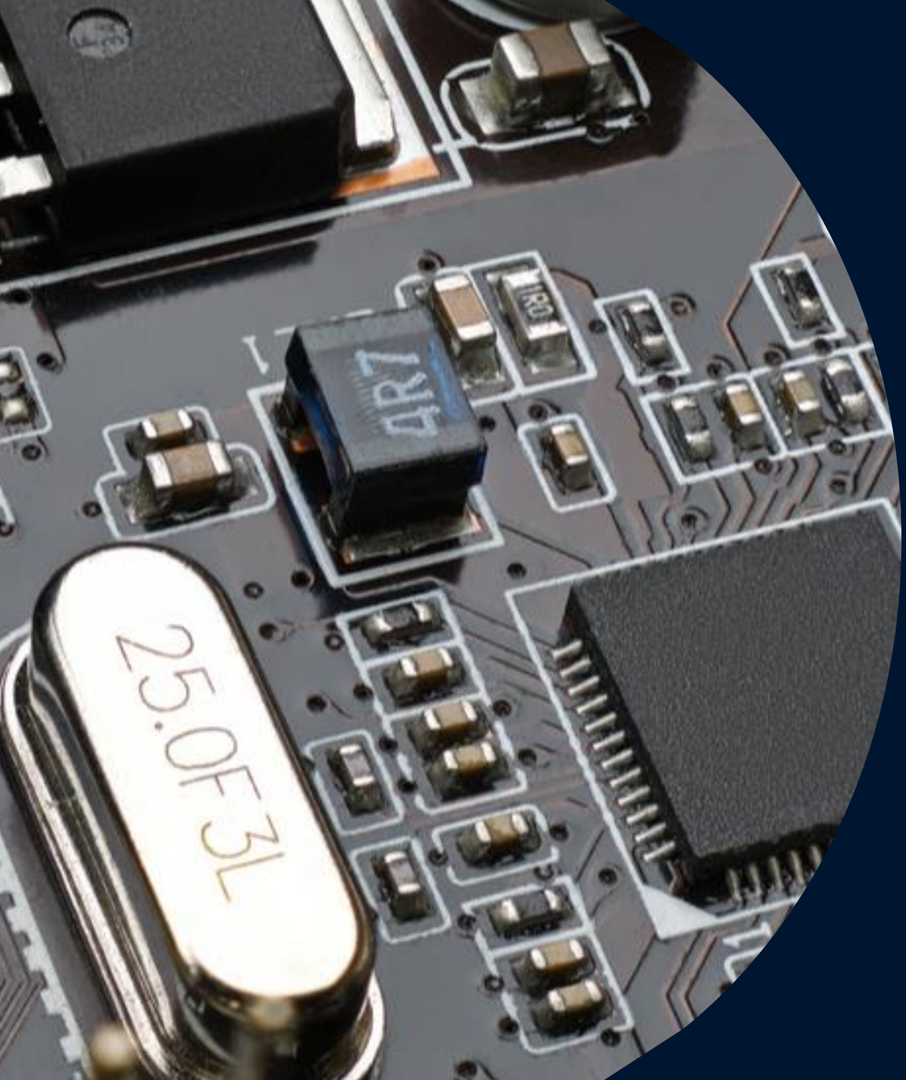
You can follow **HyperDbg** on Twitter to get notified about new releases, or join the HyperDbg Telegram group, where you can ask developers and open-source reversing enthusiasts for help with setting up and running HyperDbg.

Description

HyperDbg is designed with a focus on using modern hardware technologies to provide new features to the debuggers' world. It operates on top of Windows by virtualizing an already running system using Intel VT-x and Intel PT. This debugger aims not to use any APIs and software debugging mechanisms, but instead, it uses Second Layer Page Table (a.k.a. Extended Page Table or EPT) extensively to monitor both kernel and user executions.

HYPERDBG

A debugger designed for analyzing, fuzzing and reversing



Main Motivation

The key intuition behind **hwdbg** is making security researchers and hardware engineers able to run **custom action** in the smallest (shortest) event in a digital circuit which is the changes (rising-edge & falling-edge) within the **clock cycle**.

Key Contributions

hwdbg comes with these contributions.

- Introduction of a Unified Debugging Framework
- Open-source and Customizable Tool
- Integration with FPGAs
- Robust Debugging Experience
- Enhancement of Reverse Engineering and Chip Fuzzing Capabilities
- Comprehensive Scripting Support

01

Software

The software side is implemented in C/C++ for sending debugging commands and parsing scripts.

02

Hardware

The hardware side will be written in chisel programming language as well as Verilog and SystemVerilog.

03

Platform

The testing platform is AMD Xilinx FPGAs.

Key Features

An open-source logic analyzer

Signal Analyzer

Automatic testing and fuzzing of chips by software generated testcases

Testing & Fuzzing

Software Abstractions

Simulate software debugging concepts in hardware

Signal Manipulator

Manipulating digital signals by changing the state of wires

Debugging Modes

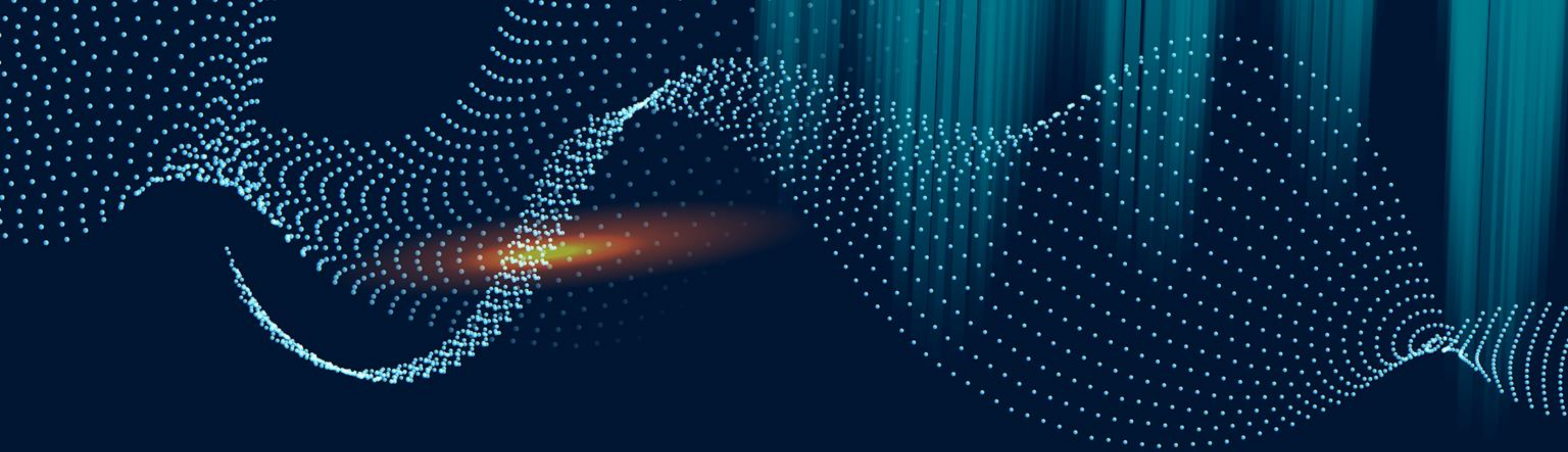
Passive debugging

In this debugging mode, **hwdbg** acts as an introspection tool, and monitors different signals to the target debugging module.

Active debugging

In this debugging mode, **hwdbg** is able to both monitor and manipulate signals (Input/Output). These signal modifications are configured by the user's custom scripts.

Outline

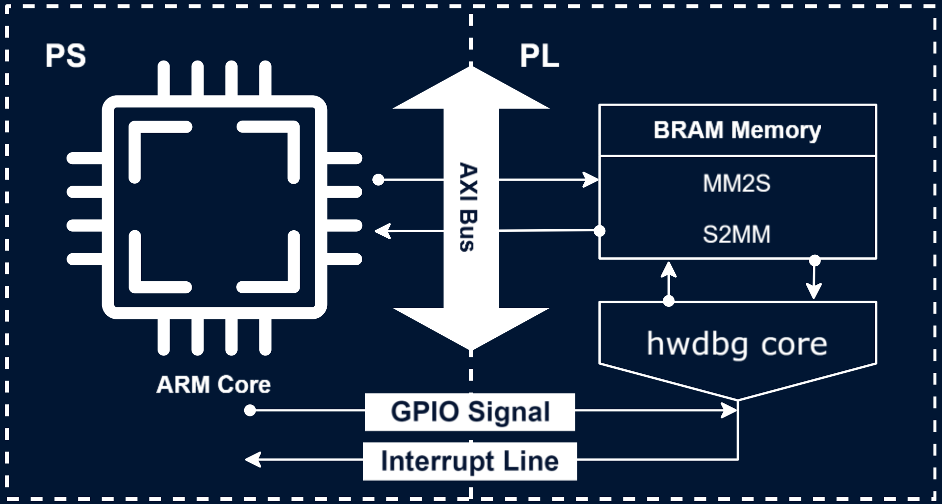


02 Design

The proposed debugger chip generator design

PS <> PL Communication

hwdbg communicates from PS to PL by using a shared BlockRAM over the AXI bus, as well as an interrupt line from PL to PS and a shared GPIO line from PS to PL.



Communication Modules

Sending module is responsible for preparing mandatory headers and write to BRAM as well as interrupting PS.

—Sending Module

Once share PS <> PL line is high, receiving verifies BRAM data headers and notifies interpreter.

—Receiving Module

Because only one BRAM port is shared with PL, synchronization is needed to avoid data corruption in case of simultaneous writes.

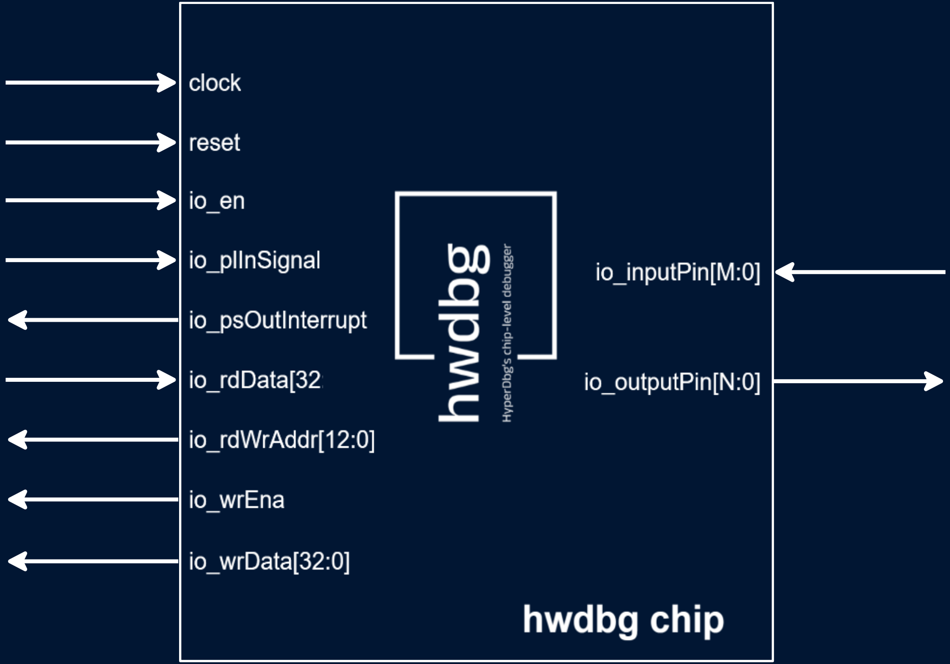
—Send/Receive Synchronization Module

Once a valid packet is received, the interpreting module configures the chips and sends the response to the debugger.

—Interpreting Module

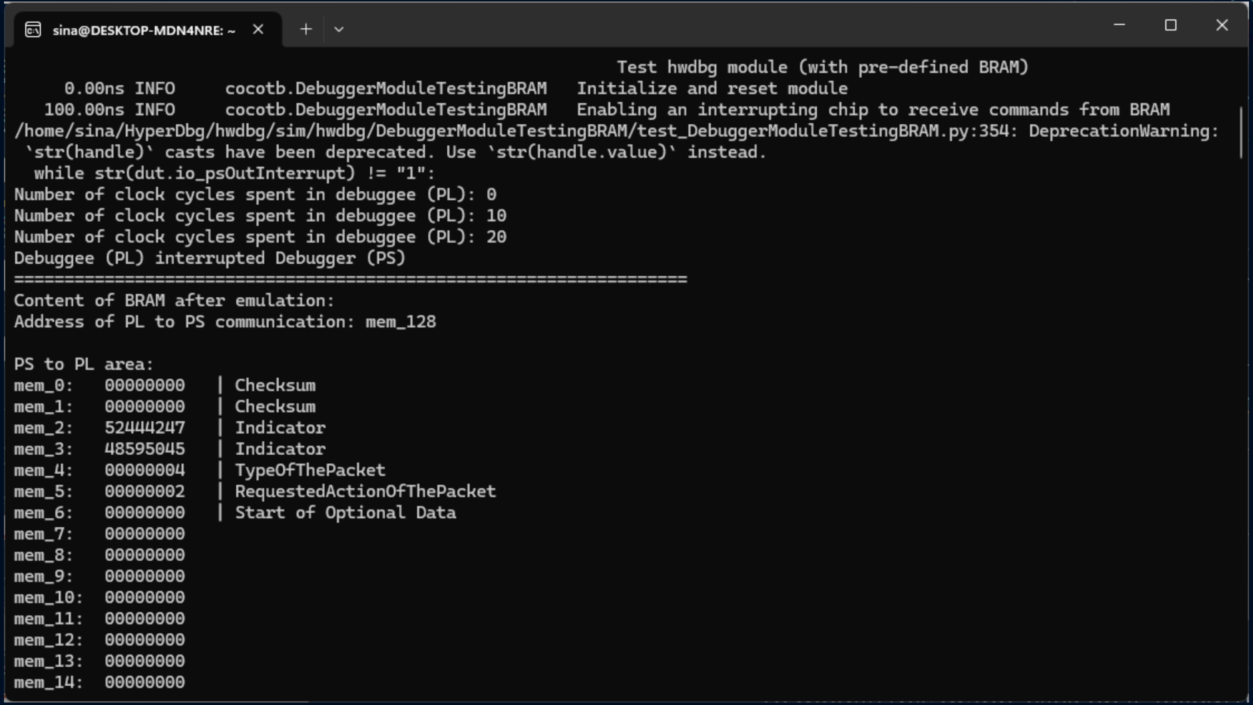
Input/Output Ports

This picture illustrates the input/output pins of the **hwdbg** design.



BRAM Simulator

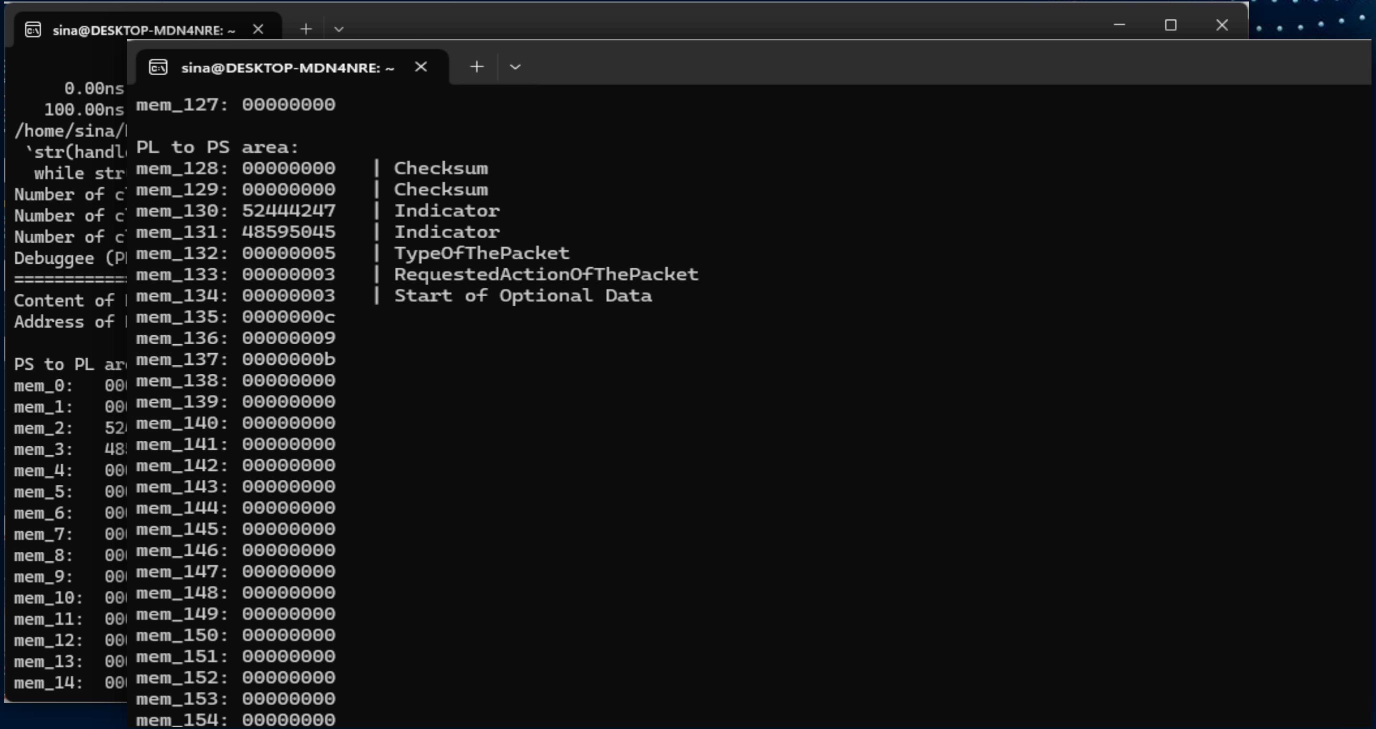
Here is the BRAM simulator (PS to PL area).



```
                                                              Test hwdbg module (with pre-defined BRAM)
     0.00ns INFO     cocotb.DebuggerModuleTestingBRAM   Initialize and reset module
   100.00ns INFO     cocotb.DebuggerModuleTestingBRAM   Enabling an interrupting chip to receive commands from BRAM
/home/sina/HyperDbg/hwdbg/sim/hwdbg/DebuggerModuleTestingBRAM/test_DebuggerModuleTestingBRAM.py:354: DeprecationWarning:
 `str(handle)` casts have been deprecated. Use `str(handle.value)` instead.
  while str(dut.io_psOutInterrupt) != "1":
Number of clock cycles spent in debuggee (PL): 0
Number of clock cycles spent in debuggee (PL): 10
Number of clock cycles spent in debuggee (PL): 20
Debuggee (PL) interrupted Debugger (PS)
===================================================================
Content of BRAM after emulation:
Address of PL to PS communication: mem_128

PS to PL area:
mem_0:   00000000   | Checksum
mem_1:   00000000   | Checksum
mem_2:   52444247   | Indicator
mem_3:   48595045   | Indicator
mem_4:   00000004   | TypeOfThePacket
mem_5:   00000002   | RequestedActionOfThePacket
mem_6:   00000000   | Start of Optional Data
mem_7:   00000000
mem_8:   00000000
mem_9:   00000000
mem_10:  00000000
mem_11:  00000000
mem_12:  00000000
mem_13:  00000000
mem_14:  00000000
```

BRAM Simulator (cont.)

Here is the BRAM simulator (PL to PS area).

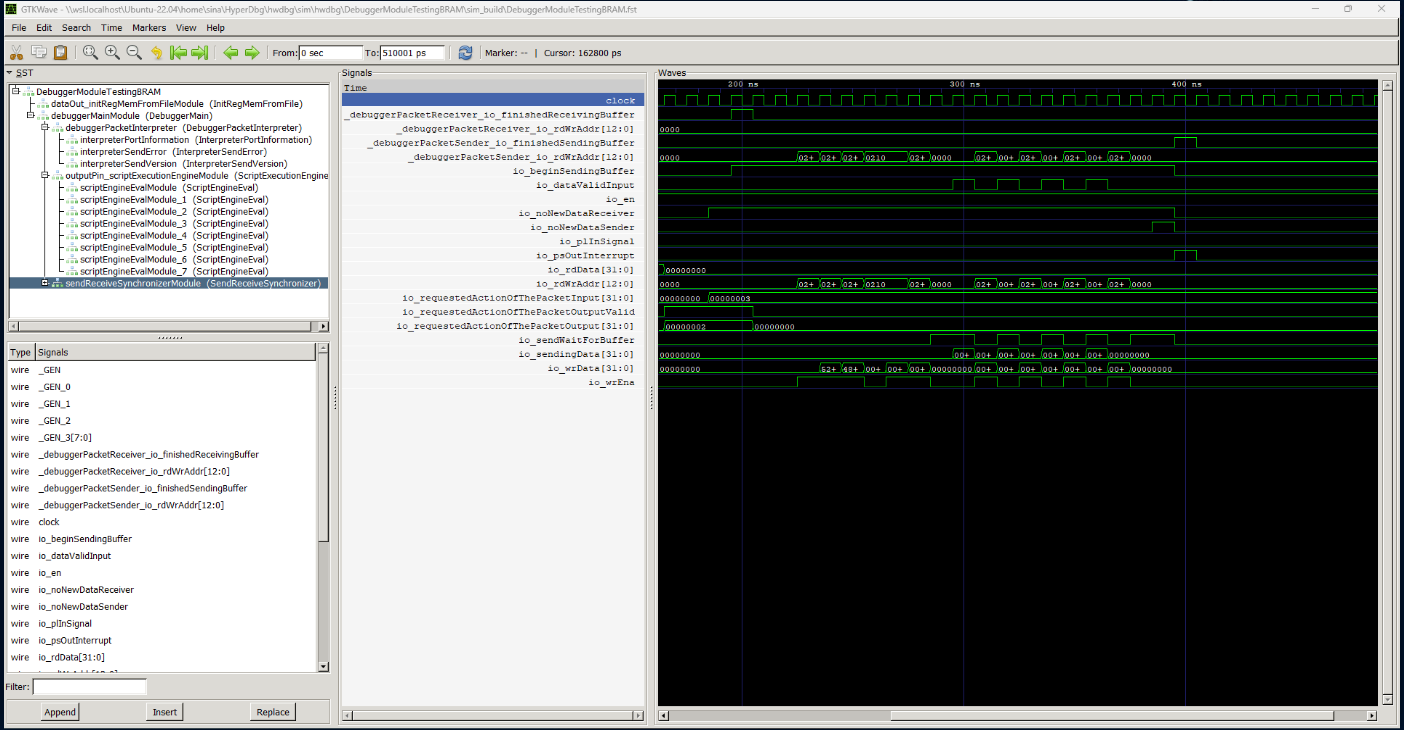


```
   0.00ns
 100.00ns
/home/sina/
 `str(handl
  while str
Number of c
Number of c
Number of c
Debuggee (P
============
Content of
Address of

PS to PL ar
mem_0:    00
mem_1:    00
mem_2:    52
mem_3:    48
mem_4:    00
mem_5:    00
mem_6:    00
mem_7:    00
mem_8:    00
mem_9:    00
mem_10:   00
mem_11:   00
mem_12:   00
mem_13:   00
mem_14:   00
```

```
mem_127: 00000000

PL to PS area:
mem_128: 00000000    | Checksum
mem_129: 00000000    | Checksum
mem_130: 52444247    | Indicator
mem_131: 48595045    | Indicator
mem_132: 00000005    | TypeOfThePacket
mem_133: 00000003    | RequestedActionOfThePacket
mem_134: 00000003    | Start of Optional Data
mem_135: 0000000c
mem_136: 00000009
mem_137: 0000000b
mem_138: 00000000
mem_139: 00000000
mem_140: 00000000
mem_141: 00000000
mem_142: 00000000
mem_143: 00000000
mem_144: 00000000
mem_145: 00000000
mem_146: 00000000
mem_147: 00000000
mem_148: 00000000
mem_149: 00000000
mem_150: 00000000
mem_151: 00000000
mem_152: 00000000
mem_153: 00000000
mem_154: 00000000
```

BRAM Simulator Waves

Here is the BRAM simulator wave results for **hwdbg**.



GTKWave - \\wsl.localhost\Ubuntu-22.04\home\sina\HyperDbg\hwdbg\sim\hwdbg\DebuggerModuleTestingBRAM\sim_build\DebuggerModuleTestingBRAM.fst

From: 0 sec To: 510001 ps Marker: -- | Cursor: 162800 ps

SST

```
DebuggerModuleTestingBRAM
  dataOut_initRegMemFromFileModule  (InitRegMemFromFile)
  debuggerMainModule  (DebuggerMain)
    debuggerPacketInterpreter  (DebuggerPacketInterpreter)
      interpreterPortInformation  (InterpreterPortInformation)
      interpreterSendError  (InterpreterSendError)
      interpreterSendVersion  (InterpreterSendVersion)
    outputPin_scriptExecutionEngineModule  (ScriptExecutionEngine
      scriptEngineEvalModule  (ScriptEngineEval)
      scriptEngineEvalModule_1  (ScriptEngineEval)
      scriptEngineEvalModule_2  (ScriptEngineEval)
      scriptEngineEvalModule_3  (ScriptEngineEval)
      scriptEngineEvalModule_4  (ScriptEngineEval)
      scriptEngineEvalModule_5  (ScriptEngineEval)
      scriptEngineEvalModule_6  (ScriptEngineEval)
      scriptEngineEvalModule_7  (ScriptEngineEval)
    sendReceiveSynchronizerModule  (SendReceiveSynchronizer)
```

Type	Signals
wire	_GEN
wire	_GEN_0
wire	_GEN_1
wire	_GEN_2
wire	_GEN_3[7:0]
wire	_debuggerPacketReceiver_io_finishedReceivingBuffer
wire	_debuggerPacketReceiver_io_rdWrAddr[12:0]
wire	_debuggerPacketSender_io_finishedSendingBuffer
wire	_debuggerPacketSender_io_rdWrAddr[12:0]
wire	clock
wire	io_beginSendingBuffer
wire	io_dataValidInput
wire	io_en
wire	io_noNewDataReceiver
wire	io_noNewDataSender
wire	io_plInSignal
wire	io_psOutInterrupt
wire	io_rdData[31:0]

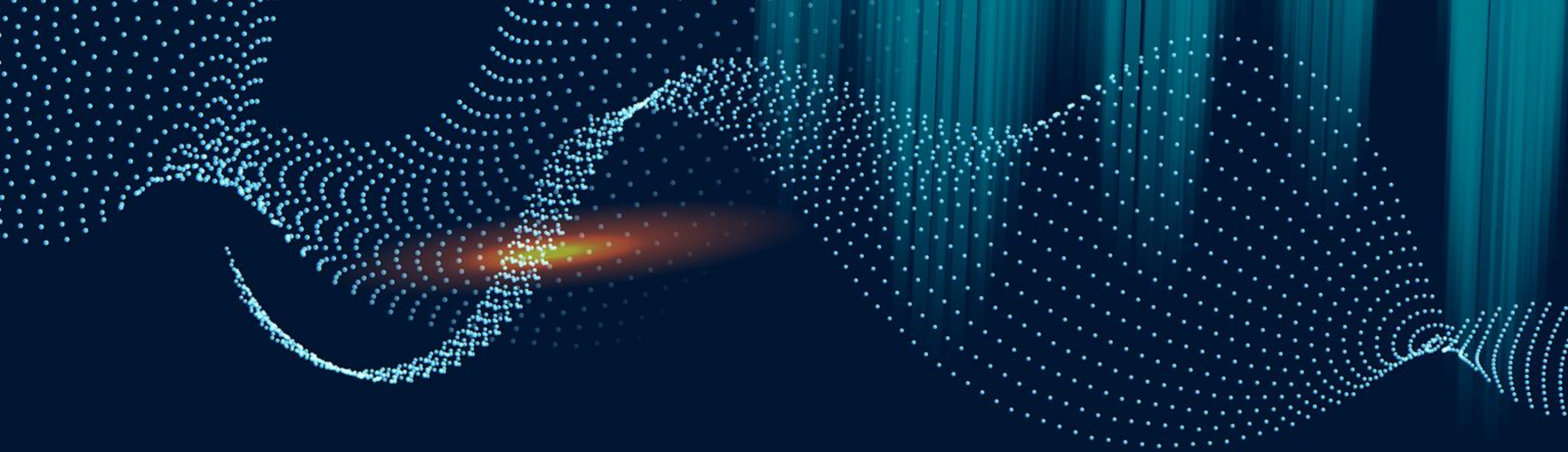
Filter:

Signals

```
Time
clock
_debuggerPacketReceiver_io_finishedReceivingBuffer
_debuggerPacketReceiver_io_rdWrAddr[12:0]
_debuggerPacketSender_io_finishedSendingBuffer
_debuggerPacketSender_io_rdWrAddr[12:0]
io_beginSendingBuffer
io_dataValidInput
io_en
io_noNewDataReceiver
io_noNewDataSender
io_plInSignal
io_psOutInterrupt
io_rdData[31:0]
io_rdWrAddr[12:0]
io_requestedActionOfThePacketInput[31:0]
io_requestedActionOfThePacketOutputValid
io_requestedActionOfThePacketOutput[31:0]
io_sendWaitForBuffer
io_sendingData[31:0]
io_wrData[31:0]
io_wrEna
```

Waves

Outline

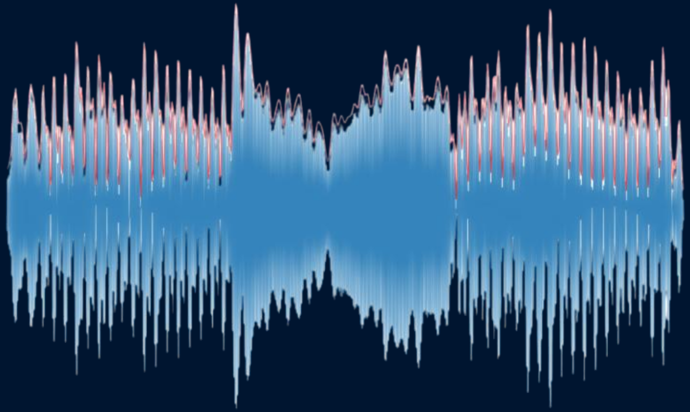


03 Sensors

The debugging sensor modules embedded in the **hwdbg** debugger

Side-channel Attacks in Microchips/IP Cores

- A side-channel attack in chips and IP cores involves exploiting indirect information leakage, such as power consumption or electromagnetic emissions, to extract sensitive data.
- We use sensors like Ring Oscillators (RO) for detecting these side channels because they can monitor variations in hardware characteristics, which may indicate the presence of side-channel leakage or attacks.



PUF

Physical Unclonable Function

- A PUF is a physical entity utilized in microchips and other electronic devices to generate unique, device-specific cryptographic keys based on the uncontrollable physical variations that occur during the manufacturing process.
- These unique differences, such as variations in silicon pathways, ensure that each PUF is unique and make it theoretically impossible to duplicate or clone.

Debugging Sensors

Ring Oscillator (RO)

Measures variations in signal propagation delay to detect changes in environmental conditions or circuit behavior.

IDELAYE2/3

Provides adjustable signal delay capabilities for timing adjustments and alignment in high-speed circuits.

Time-to-Digital Converter (TDC)

Converts the time interval between two events into a digital value for precise timing measurements.

Debugging Sensors (cont.)

Voltage Sensor

Monitors and measures the voltage levels within a circuit to ensure stable and correct operation.

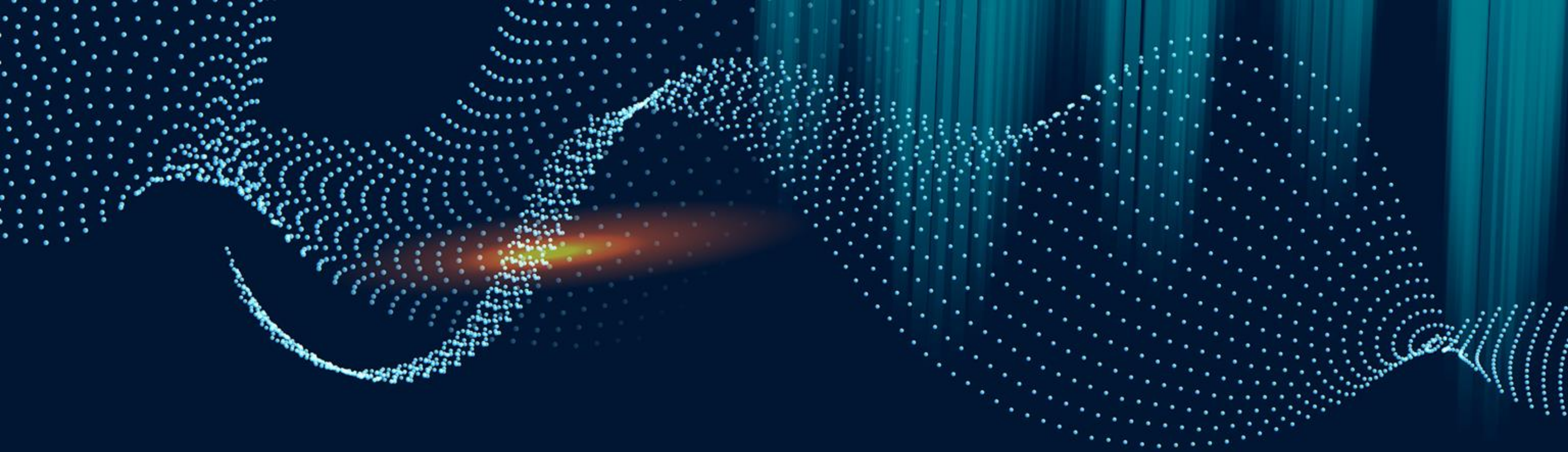
Frequency & Clock Sensors

Monitors the frequency and stability of clock signals to ensure they remain within the expected range and performance criteria.

Temperature Sensor

Measures the temperature within the chip to monitor thermal conditions and prevent overheating.

Outline



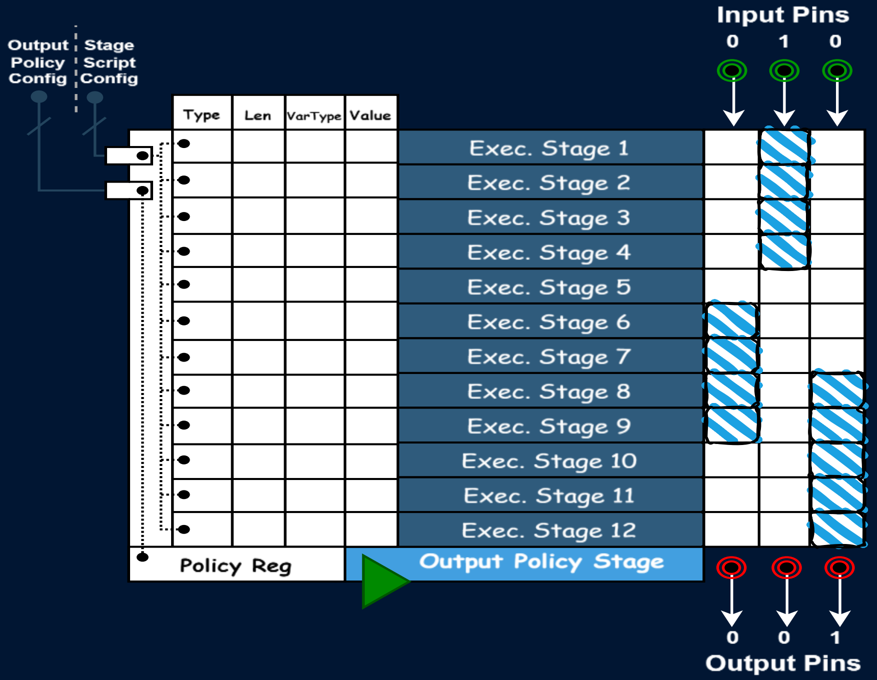
04

Scripting Language

HyperDbg's *dslang* variation for hardware debugging

Script Execution Stages

This picture demonstrates the execution of different IR scripts in each stage.



Ports & Pins

Registers	Pseudo-registers
@hw_pin0	$hw_clk, or $hw_clock: 1 or 0
@hw_pin1	$hw_counter and, $hw_clock_edge_counter
@hw_pinX	$hw_clock_frequency
@hw_port0	$hw_stage: Stage number of script
@hw_port1	
@hw_portX	

Supported Statements

Memory Modification

Modifying the BRAM memory

External Argument

Arguments as $arg0, $arg1, etc.

Register and Variable Assignment

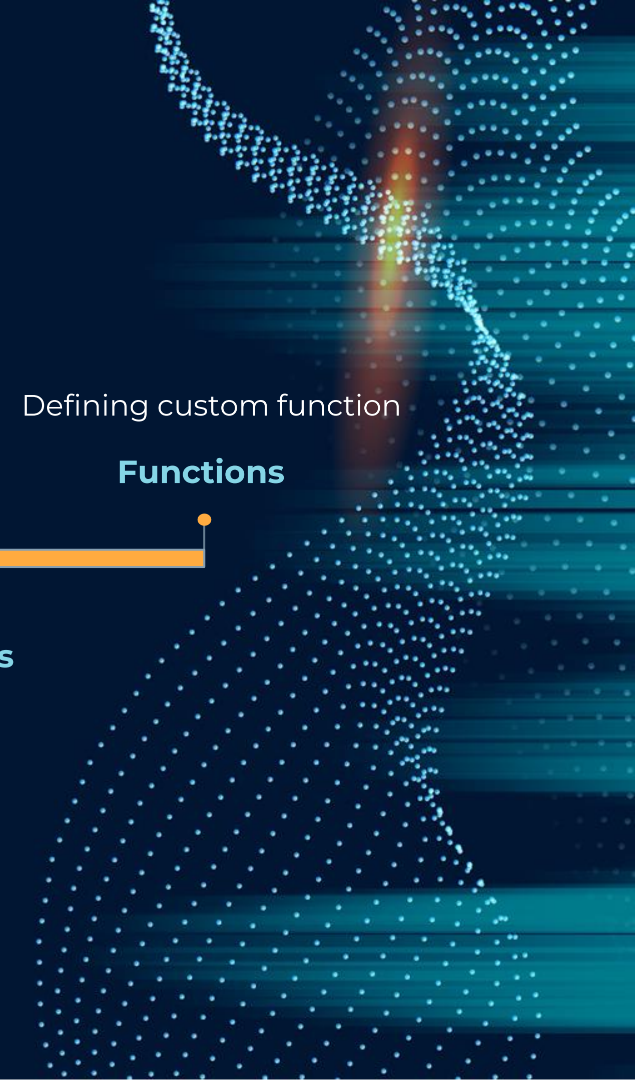
Changing pins and ports (registers) and assign to variables

Conditional Statements

if, else, elsif statements

Functions

Defining custom function



Supported Operators

Precedence	Operators
Parentheses	( )
Unary Operators	-, +, ~, &, *
Arithmetic Operators	*, /, %
Arithmetic Operators	+, -
Shift Operators	>>, <<
Comparison Operators	>=, <=, >, ==, !=
Bitwise AND Operator	&
Bitwise XOR Operator	^
Logical AND Operator	&&
Logical OR Operator	||

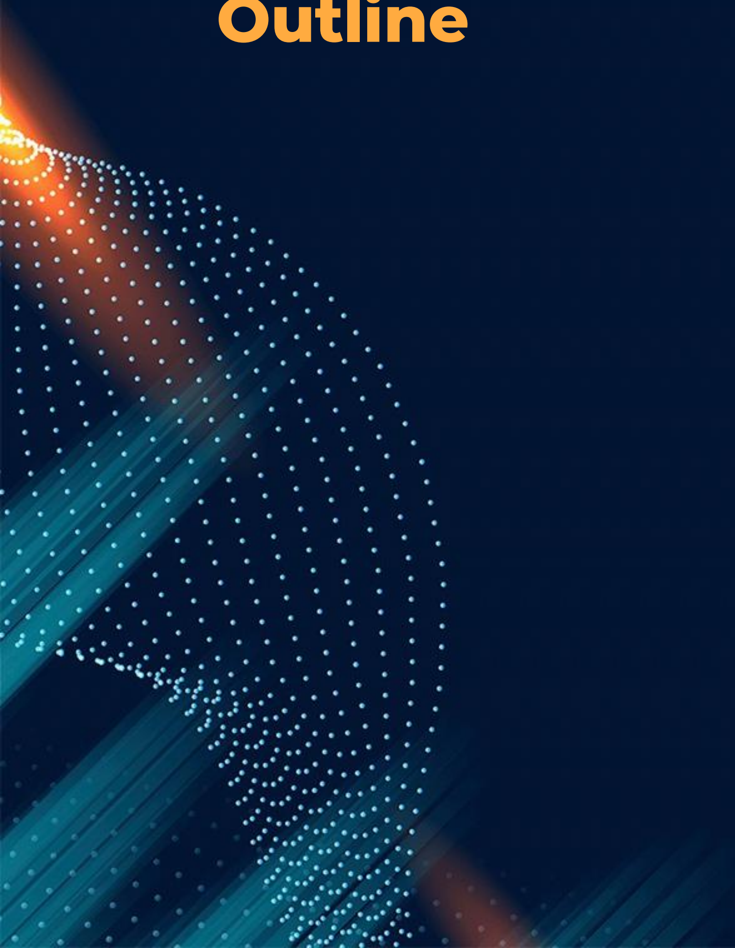
Example Script 1

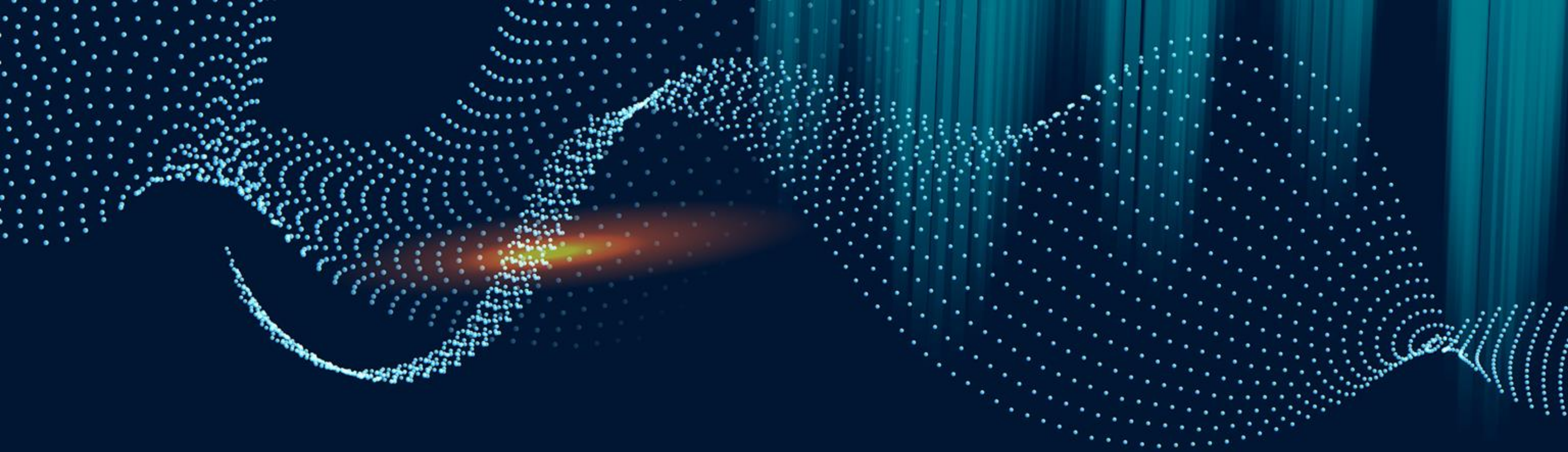
```
? {

  if (@hw_port5 == 55) {
    printf("@hw_port5 is equal to 0x55\n");
  }
  elsif(@hw_port5 == 66) {
    printf("@hw_port5 is equal to 0x66\n");
  }
  elsif(@hw_port5 == 77) {
    printf("@hw_port5 is equal to 0x77\n");
  } else {
    printf("@hw_port5 is not equal to 0x55, 0x66,
           0x77. It is equal to %llx\n", @hw_port5);
  }
}
```

Example Script 2

```
? {
  int my_func1(int var1) {
    result = var1 + 1;
    printf("my_func1 %d\n", result);
    return result;
  }
  int my_func2(int var1) {
    result = var1 * 2;
    printf("my_func2 %d\n", result);
    return result;
  }

  int my_func3(int var1) {
    result = my_func1(var1) + my_func2(var1);
    printf("my_func3 %d\n", result);
    return result;
  }
  printf("%d\n", my_func3(2));
}
```

Outline





05

D.F.R.

Debugging, Fuzzing, Reversing

Key (must-have) Software Debugger Features

- Reading/Writing Memory
- Inspecting/Modifying Registers Values
- Stepping Through Instructions
- Putting Breakpoints and Pausing The Debuggee

Debugging Hardware Like Software

Logic Analyzer

Reading/Writing Memory

Inspecting/Modifying Registers Values

Signal Inspection and Modification

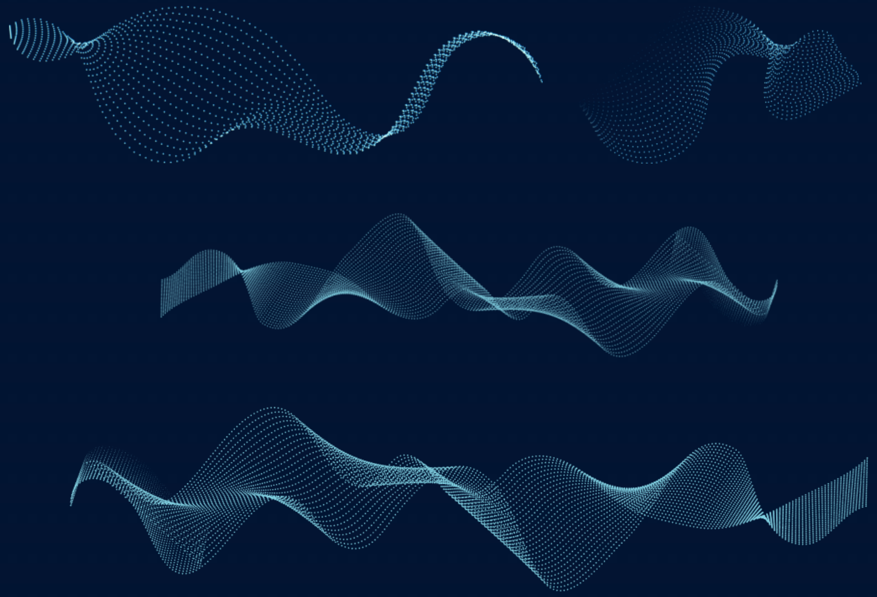
Emulation of Instruction Stepping by Controlling Clock Signal

Stepping Through Instructions

Putting Breakpoints and Pausing The Debuggee

Cutting The Signals Based on Conditions

Software Stepping in Hardware



- Once we can control the clock signal, we are able to see and modify I/O ports at each cycle.
- The next rising-edge clock will be inserted once the introspection/modification is done.
- Not possible in chips with internal clock sources.

Anomaly Definition

Anomalies In Expected Functionalities

These occur when the device fails to compute the correct output due to specific inputs, detectable through emulation.

Anomalies Based On Sensors Indications

Sensors can indicate anomalies through abnormal state changes, signaling potential internal issues in the chip.

Fuzzing & Fault-Injection

Cut Singal

Remove signal at different random stages

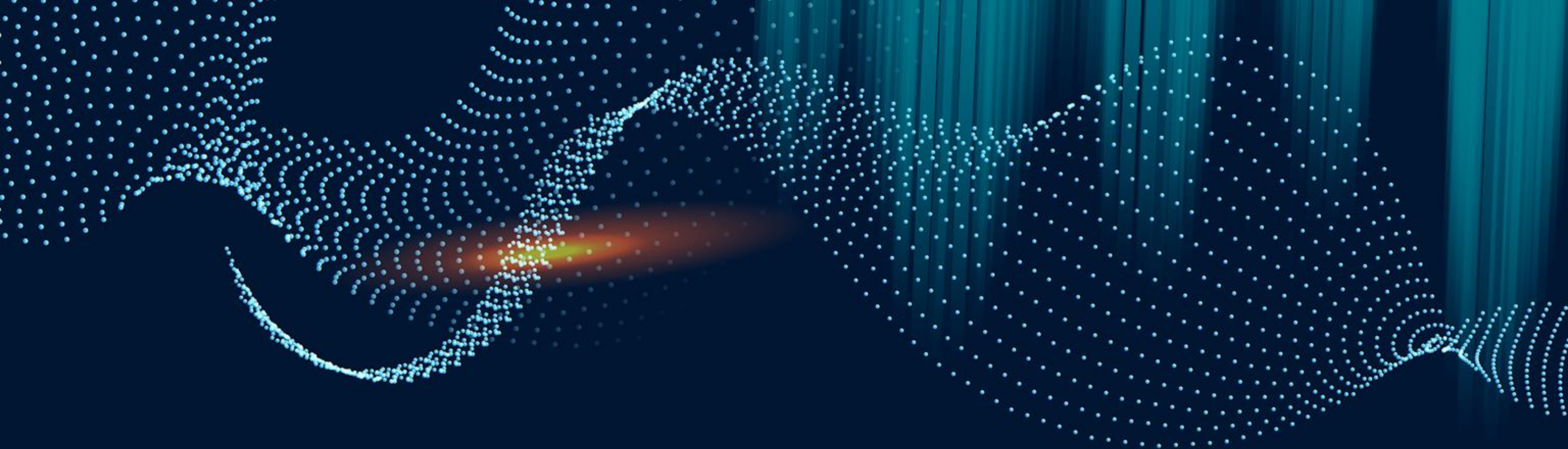
Fuzzing

Providing random software generated data and observe the output

Change Clock domain

Using PWM with different duty cycles

Outline



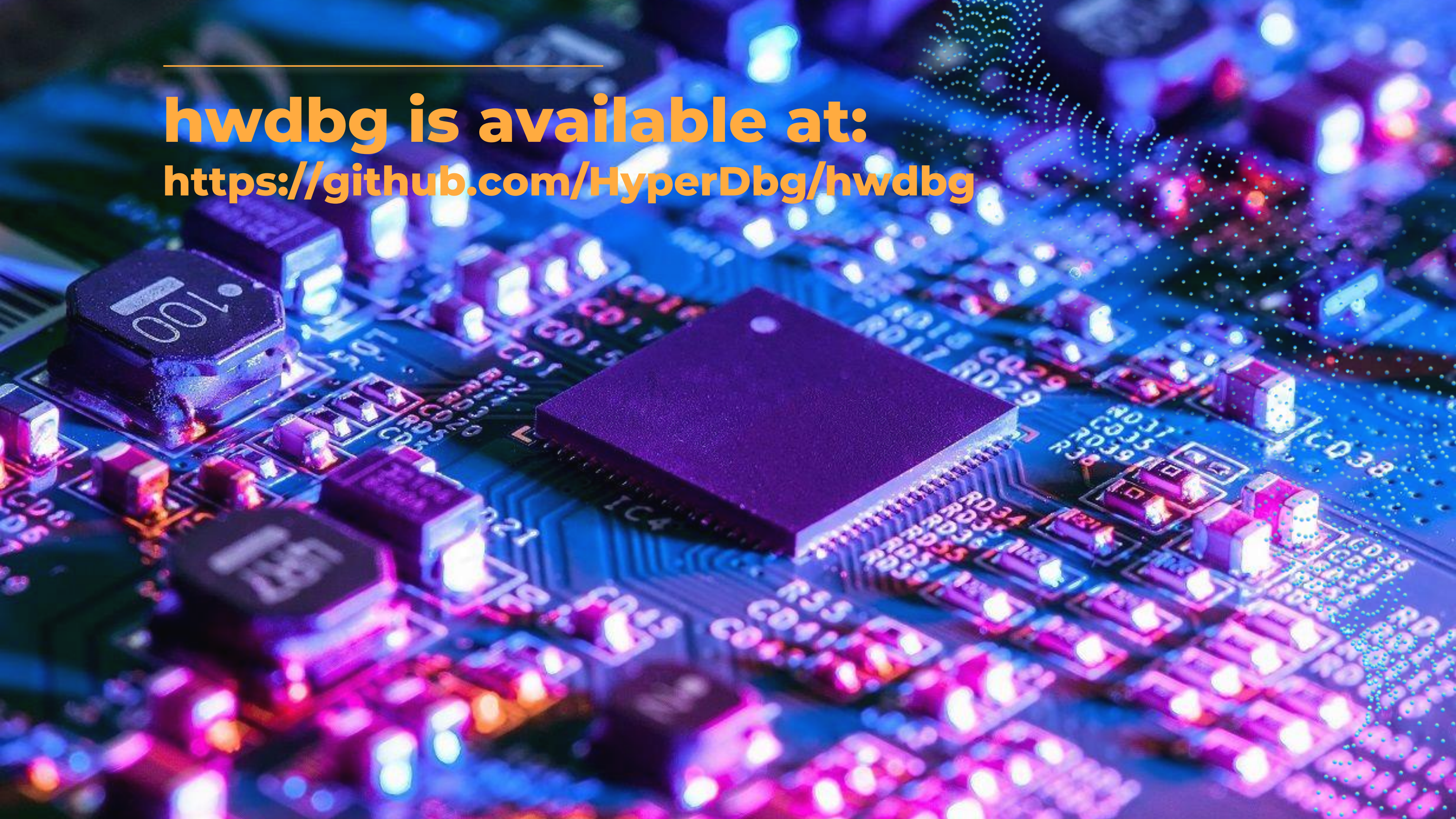
06

Evaluation

Evaluating hwdbg debugger chip generator

dslang vs TSM

	hwdbg (*dslang*)	Xilinx ILA (TSM)
Can act as a logic analyzer	Yes	Yes
Speed comparison	Slower	Faster
Can trigger an event (e.g., a breakpoint)	Yes	Yes
Has debugging sensors	Yes	No
Can modify signals on the fly (configurable)	Yes	No
Can interpret complex computations	Yes	No
Supports on-the-fly configuration	Yes	No
Notify the debugger without triggering event	Yes	No
Supports software stepping	Yes	No
Simplicity of state machine scripting	Simpler, software-like scripting	More complex
Can perform stepping fault-injection and signal manipulation	Yes	No
Can perform active debugging (producing fuzzing signals)	Yes	No



hwdbg is available at:

https://github.com/HyperDbg/hwdbg

Questions

Any questions?

Thank you!

Do you have any questions?

Mail:	sina@hyperdbg.org
Twitter:	@Intel80x86
Blog:	rayanfam.com

or

https://github.com/orgs/HyperDbg/discussions