

Leveraging Open-Source Frameworks in Commercial FPGA Development

A Case Study with SpinalHDL

Krzysztof Czyż, PhD

CTO @ embeivity
krzysztof.czyz@embeivity.com

Mateusz Maciąg

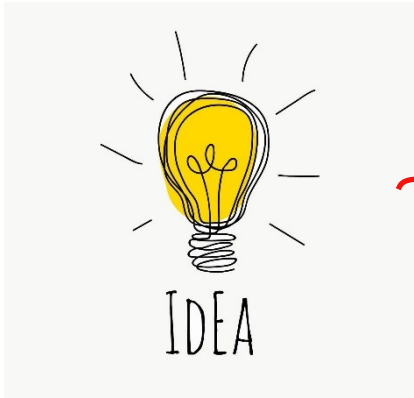
architect @ embeivity
mateusz.maciag@embeivity.com



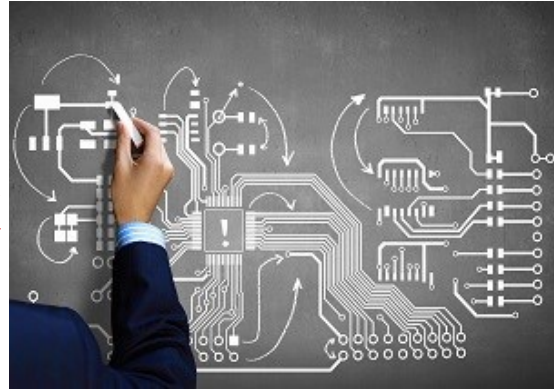
What's in it for you?

- How to improve FPGA design services?
- SpinalHDL - a tool enabling speed-up of gateway development.
- Vex/NaxRISCV - a highly configurable soft core.
- Cons and pros of using SpinalHDL in commercial projects.





Original idea

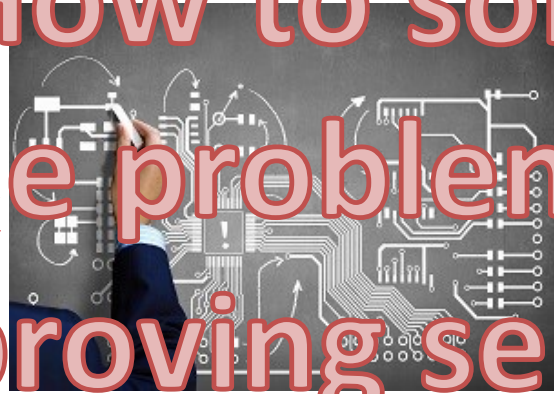


Design, prototyping and testing
(hardware, firmware, gateway)



Product

How to solve the problem of improving service?

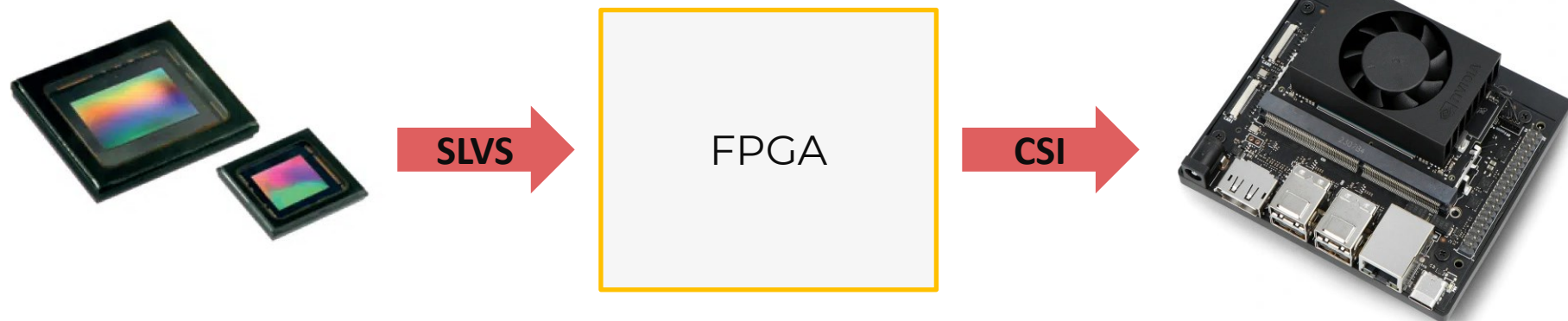


Original idea

Design, prototyping and testing
(hardware, firmware, software)

Product

find the right tools



The SLVS-to-CSI bridge

How fast could you implement it?

- high-level hardware description language
- hosted on the top of Scala
- focus on RTL description
- interoperable with existing tools
it generates VHDL/Verilog files (as an output netlist)
it can integrate VHDL/Verilog IP as blackbox
- open source, started in December 2014
by Charles Papon

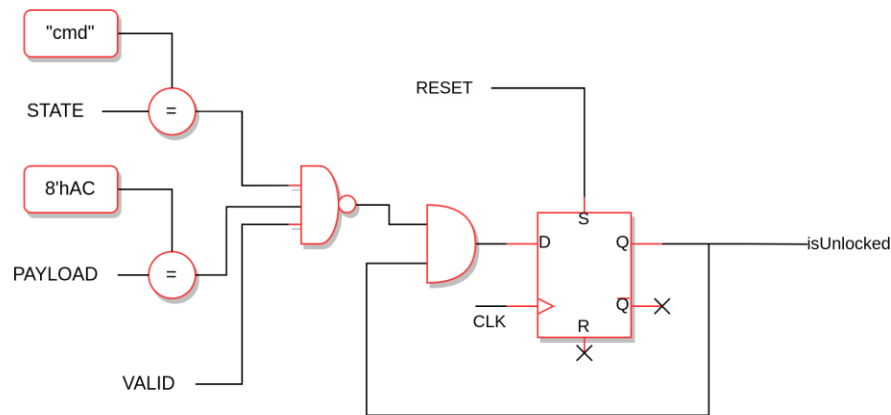


SpinalHDL

Registers and state machines

```
val isUnlocked = Reg(Bool) init True
```

```
val cmd: State = new State {  
  whenIsActive {  
    when(spiDev.io.rx.valid) {  
      when(  
        // 0xAC - lock bootloader  
        spiDev.io.rx.payload === B"10101100"  
      ) {  
        isUnlocked := False  
        goto(trap)  
      }.elsewhen(...) {  
        ...  
      }.otherwise(goto(trap))  
    }  
  }  
}
```



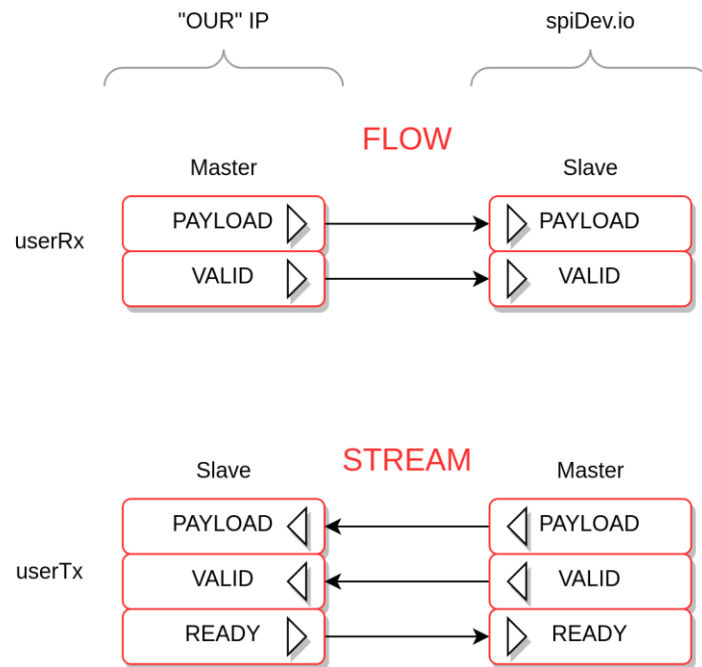
† simplified graphical representation of generated VHDL and/or Verilog code

Flows and streams

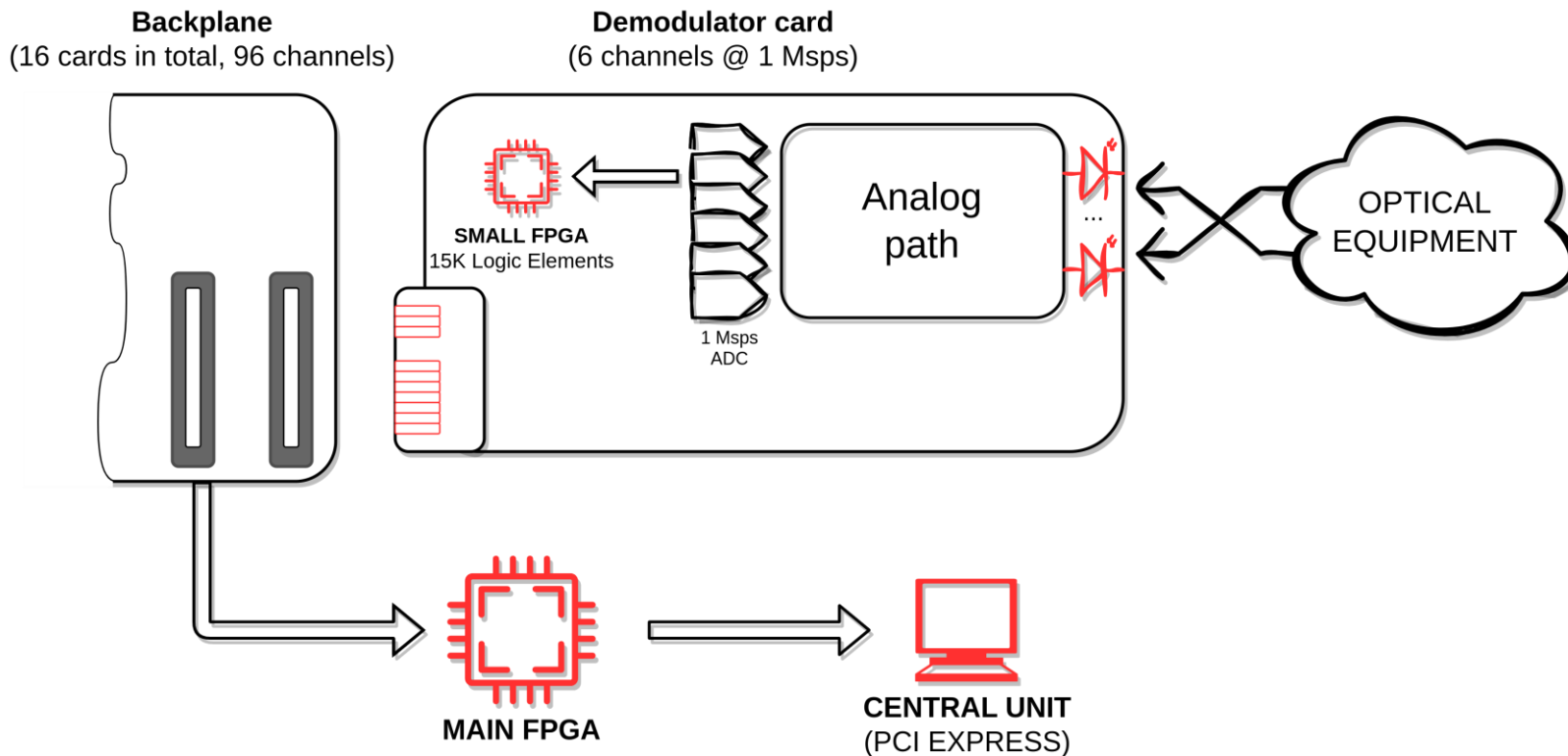
```
val io = new Bundle() {  
  val userRx = master Flow (Bits(spiWordBits))  
  val userTx = slave Stream (Bits(spiWordBits))  
}
```

```
io.userRx.setIdle()  
val user: State = new State {  
  whenIsActive {  
    when(spiDev.io.rx.valid) {  
      io.userRx << spiDev.io.rx  
    }  
  }  
}  
...  
...
```

```
...  
io.userTx  
  .throwWhen(spiTxDiscarding)  
  .continueWhen(fsm.isActive(fsm.user)) >>  
spiDev.io.tx  
spiDev.io.txBusEnable := spiMisoAllowed  
...
```

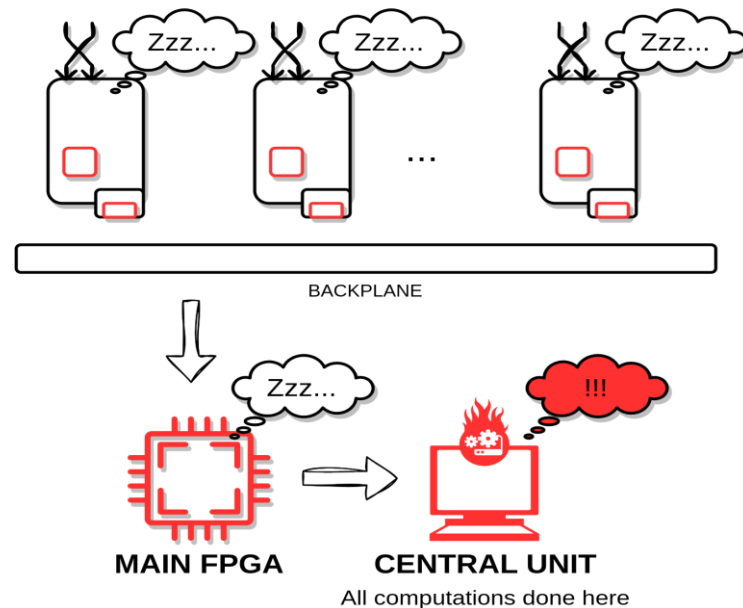


The redesign of the optical interrogator



Situation overview:

- FPGAs used so far for data transfer only (< 5% usage)
- all computation overhead lies on the central unit
- hundreds of units manufactured
- client decision: offload expensive computations to FPGAs on the cards

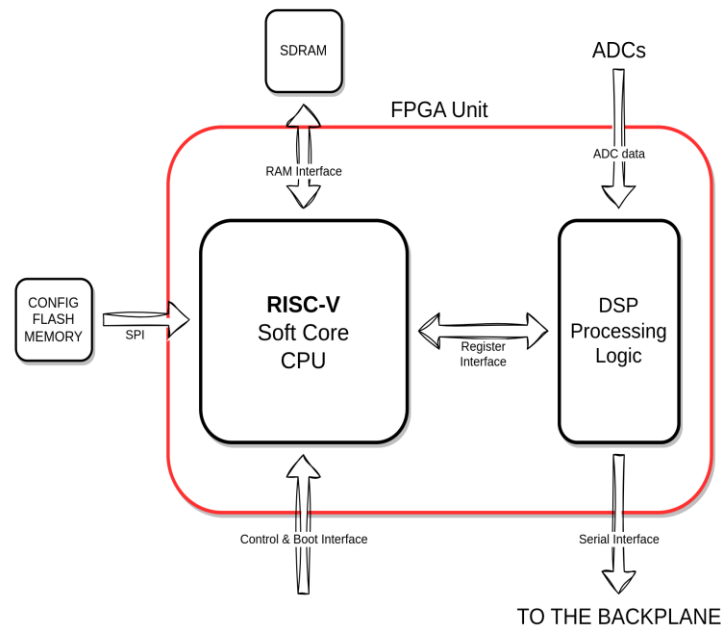


Signal processing algorithm, due to its nature, requires a soft-core CPU

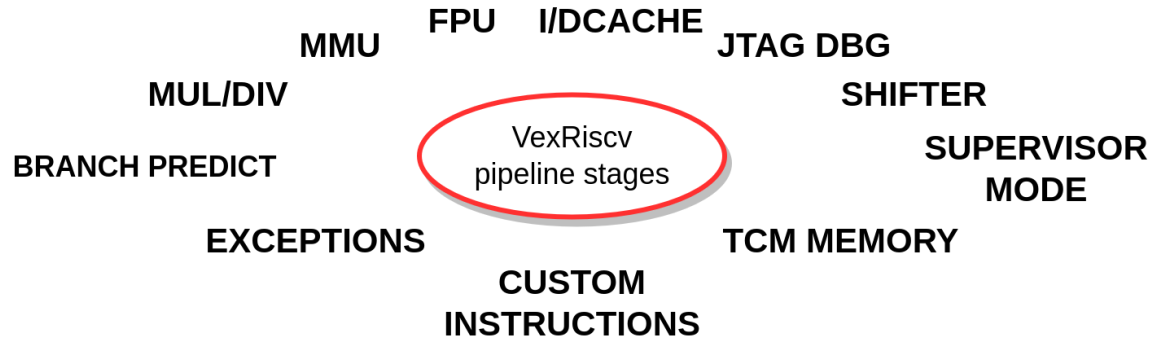
Due to very limited resources, *highly-configurable, ready for fine-tuning implementation is needed*

Chosen solution

VexRiscv – SpinalHDL implementation of the RISC-V architecture.



- RV32IM*A*F*D*C* instruction set, pipeline from 2 to 5+ stages
- "Plugin" based design:



- Interface agnostic (AXI4, Avalon, Wishbone)
- Tested with Linux, Zephyr OS, FreeRTOS

The entire CPU configuration is done in a single place

Each plugin provides many options to finely tune the implementation

Examples:

- Set non-cacheable address range
- Instruction decoding details
- ALU implementation details
- Use full barrel shift vs. simple

```
BootManager.scala RiskyConfig.scala ×
src > main > scala > vexriscv > custom > RiskyConfig.scala
32 object RiskyConfig {
34   def default = {
35     val config = RiskyConfig(
73       cpuPlugins = ArrayBuffer(
112         ...),
113         new StaticMemoryTranslatorPlugin(
114           ...ioRange = _(31 downto 28) == 0xf
115         ...),
116         new DecoderSimplePlugin(
117           ...catchIllegalInstruction = true
118         ...),
119         new RegFilePlugin(
120           ...regFileReadyKind = plugin.SYNC,
121           ...zeroBoot = false
122         ...),
123         new IntAluPlugin,
124         new SrcPlugin(
125           ...separatedAddSub = false,
126           ...executeInsertion = true
127         ...),
128         new FullBarrelShifterPlugin,
129         new MulPlugin,
130         new DivPlugin,
```

- **Peripheral** attachment
- **Bus pipelining fine-tuning**
(*scary Scala operators*)
- **Bus connections**

```
axiCrossbar.addSlaves(  
  ram.io.axi → (0x80000000L, onChipRamSize),  
  sdramCtrl.io.axi → (0x40000000L, sdramLayout.capacity),  
  apbBridge0.io.axi → (0xf0000000L, 1 MB),  
  apbBridge1.io.axi → (0xf8000000L, (1 << apb1BusConfig.  
)  
)
```

```
axiCrossbar.addSlaves(  
  ram.io.axi → (0x80000000L, onChipRamSize),  
  sdramCtrl.io.axi → (0x40000000L, sdramLayout.capacity),  
  apbBridge0.io.axi → (0xf0000000L, 1 MB),  
  apbBridge1.io.axi → (0xf8000000L, (1 << apb1BusConfig.addressWi  
)  
)
```

```
axiCrossbar.addPipelining(sdramCtrl.io.axi)((crossbar, ctrl) ⇒ {  
  crossbar.sharedCmd.halfPipe() >> ctrl.sharedCmd  
  crossbar.writeData >/→ ctrl.writeData  
  crossbar.writeRsp << ctrl.writeRsp  
  crossbar.readRsp << ctrl.readRsp  
})
```

Problem: custom booting scheme requires telling the CPU to not execute any code *until the firmware is loaded*.

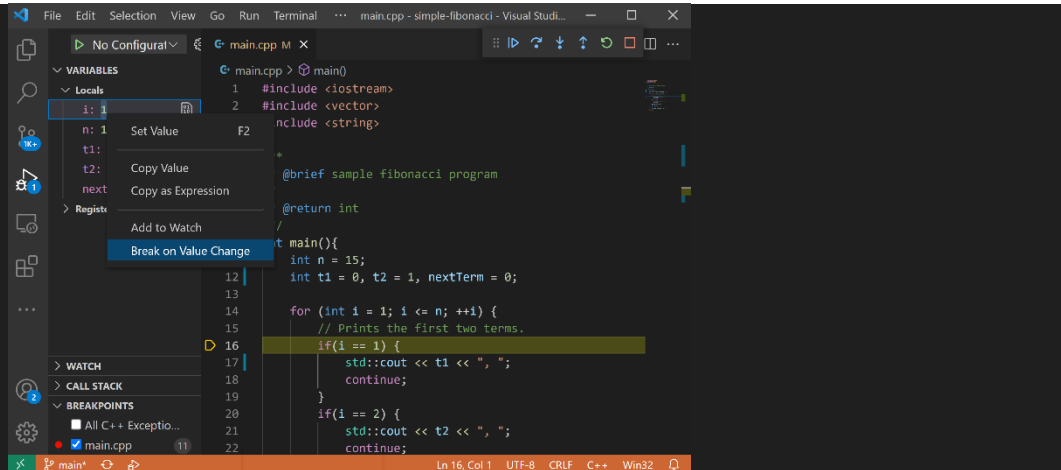
Solution:
BootHoldOnPlugin

```
21 |
22 | class BootHoldOnPlugin(holdAddress : BigInt = 0x40000000L)
23 | ... extends Plugin[VexRiscv] { /* ... */
24 |
25 | ... override def setup(pipeline: VexRiscv): Unit = {
26 | ...   io = slave(BootHoldIo()).setName("boot")
27 | ...   val pcManagerService = pipeline.service(classOf[JumpService])
28 |
29 | ...   forceJump = pcManagerService.createJumpInterface(
30 | ...     | ... pipeline.decode, 1000)
31 | ... }
32 |
33 | ... override def build(pipeline: VexRiscv): Unit = {
34 | ...   forceJump.valid := RegNext(io.softReset)
35 | ...   forceJump.payload := holdAddress
36 | ... }
37 | }
38 |
```

Simulation capabilities

Need to simulate gateway and firmware...

... just launch Verilator simulation and connect to your RiscV core with OpenOCD and GDB.



```
main.cpp - simple-fibonacci - Visual Studio...
File Edit Selection View Go Run Terminal ... main.cpp - simple-fibonacci - Visual Studio...
No Configural... main.cpp M X
VARIABLES
  Locals
    i: 1
    n: 1 Set Value F2
    t1:
    t2: Copy Value
    next Copy as Expression
  Registers
    Add to Watch
    Break on Value Change
  WATCH
  CALL STACK
  BREAKPOINTS
    All C++ Exceptio...
    main.cpp (1)
main.cpp
1 #include <iostream>
2 #include <vector>
3 #include <string>
4
5 @brief sample fibonacci program
6 @return int
7
8 int main(){
9     int n = 15;
10    int t1 = 0, t2 = 1, nextTerm = 0;
11
12    for (int i = 1; i <= n; ++i) {
13        // Prints the first two terms.
14        if(i == 1) {
15            std::cout << t1 << ", ";
16            continue;
17        }
18        if(i == 2) {
19            std::cout << t2 << ", ";
20            continue;
21        }
22    }
```

```
BOOT
SDRAM : MODE REGISTER DEFINITION CAS=3 burstLength=0
CONNECTED
SDRAM : MODE REGISTER DEFINITION CAS=3 burstLength=0
SDRAM : MODE REGISTER DEFINITION CAS=3 burstLength=0

Dhrystone Benchmark, Version 2.1 (Language: C)

Program compiled without 'register' attribute

Please give the number of runs through the benchmark: Execution starts, 200 runs through Dhryst
Final values of the variables used in the benchmark:

Int_Glob: 5
  should be: 5 Bool_Glob: 1
  should be: 1 Ch_1_Glob: A
  should be: A Ch_2_Glob: B
  should be: B Arr_1_Glob[8]: 7
  should be: 7 Arr_2_Glob[8][7]: 210
  should be: Number_Of_Runs + 10 Ptr_Glob->
Ptr_Comp: 1073765508
  should be: (implementation-dependent)
Discr: 0
  should be: 0
Enum_Comp: 2
  should be: 2
Int_Comp: 17
  should be: 17
Str_Comp: DHRYSTONE PROGRAM, SOME STRING
  should be: DHRYSTONE PROGRAM, SOME STRING Next_Ptr_Glob->
Ptr_Comp: 1073765508
  should be: (implementation-dependent), same as above
Discr: 0
  should be: 0
Enum_Comp: 1
  should be: 1
Int_Comp: 18
  should be: 18
Str_Comp: DHRYSTONE PROGRAM, SOME STRING
  should be: DHRYSTONE PROGRAM, SOME STRING Int_1_Loc: 5
  should be: 5 Int_2_Loc: 13
  should be: 13 Int_3_Loc: 7
  should be: 7 Enum_Loc: 1
  should be: 1 Str_1_Loc: DHRYSTONE PROGRAM, 1'ST STRING
  should be: DHRYSTONE PROGRAM, 1'ST STRING Str_2_Loc: DHRYSTONE PROGRAM, 2'ND STRING
  should be: DHRYSTONE PROGRAM, 2'ND STRING

Clock cycles=97424
DMIPS per Mhz: 1.16
```


- **Documentation:** needs much more use case examples, can't avoid looking into library implementation (there is a Workshop GitHub repository though).
- The SpinalHDL engine is mature and quite robust, **but library components may experience serious problems** (found a buggy SPI slave peripheral).
- Conventions above the syntax – Scala freedom to create and overload literally **any operator** makes understanding the code a bit harder.
- Some of the high-level abstractions make learning curve a bit steeper.

- **Efficient way of describing hardware:** no need to deal with implementation details. **Time-boost you gain could be impressive.**
- **There is no logic overhead in the generated code.**
- SpinalHDL is interoperable with VHDL and Verilog.
- Simulation using Verilator **enables simulation not only your design, but also testing of firmware running in simulated design.**
- Large SpinalHDL standard library.
- Open-source tool with licensing scheme enabling usage in commercial applications.
- Responsiveness of SpinalHDL's creator, Charles Papon.

Thank you !

contact@embevity.com
www.embevity.com



Linux



Adaptive
Computing
Partner

SELECT