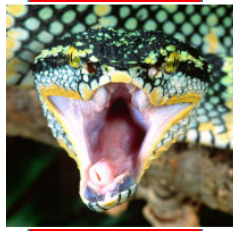
OpenCPI

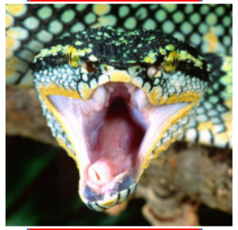
Intro to Platform Development



Summary of OpenCPI Development Roles

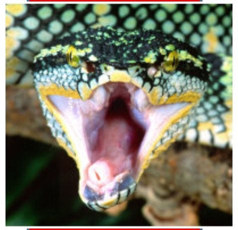
3 types of development with common Makefile, XML driven workflow

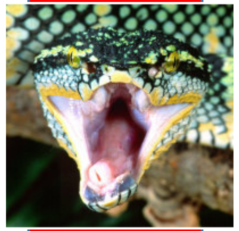
	Application Development	Component Development	Platform Development
Objective	• Create applications using components	• Create building blocks for applications	• Create infrastructure for running applications
Examples	• Tb_bias • FSK app	• Bias • FIR filter	• Zedboard • Matchstiq • Transceivers
Key functions	• Declare components and their connections and properties	• Process data and interface between othe components • Vendor agnostic (ideally)	• Provide interface to software and FPGA peripheral (devices workers)
Skills Required	• Familiarity with component library	• S/W: C, C++ • H/W: VHDL	• H/W: VHDL • Strong knowledge of platform architecture and interfaces
	Knowledge of OpenCPI build flow		



Overview

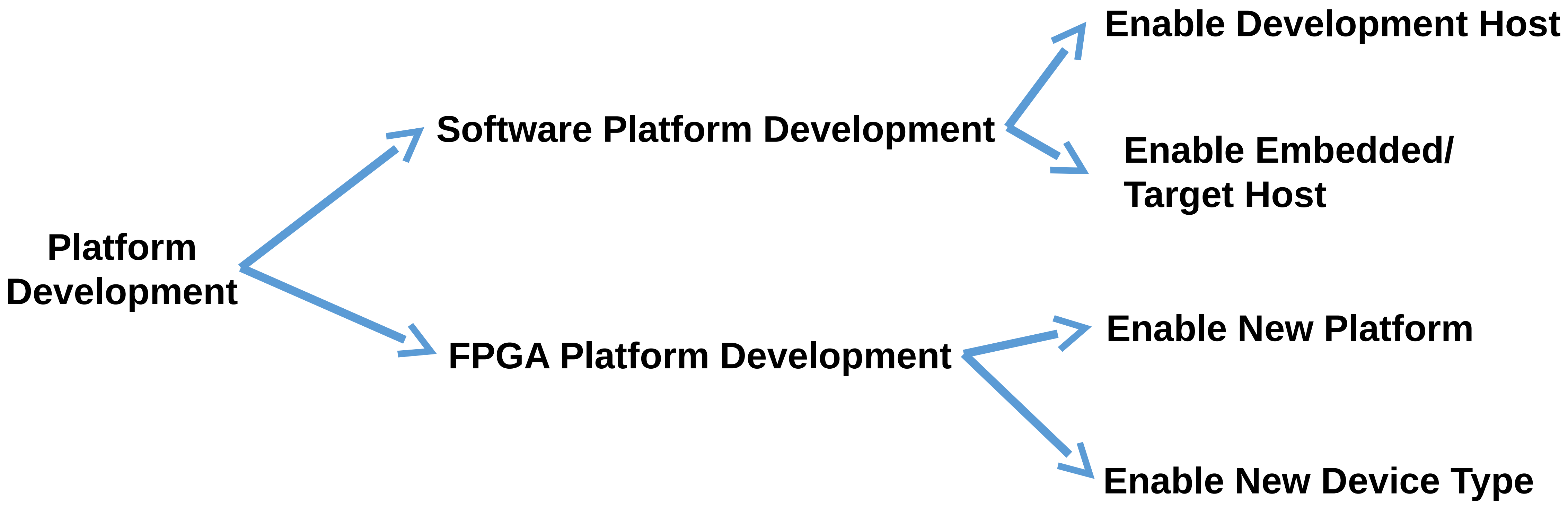
- Review of Terminology
- Systems vs Platforms ⇨ Platform Development
- Enabling a new System (What's the process?)
- Enabling Development Hosts (new Operating System)
- Enabling GPP (General Purpose Processor) Platforms
- Enabling FPGA (Field Programmable Gate Array) Platforms
  - Case Study: Epiq Matchstiq-Z1
  - Case Study: Device/Proxy Workers to support Lime MicroSystems Transceiver



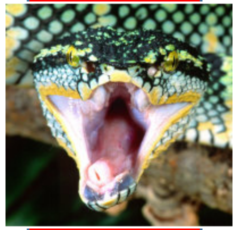


Types of Platform Development

enabling new platforms and devices for OpenCPI apps



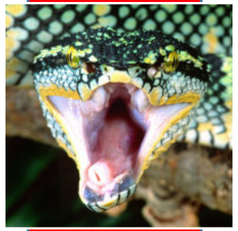
Review of Terminology



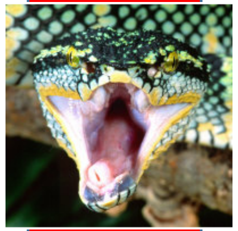
- Component – a “promise” of a function where properties and ports are defined
- Application – a set of connected components
- Platform – a processor and its directly-connected hardware
- System – Platforms connected by interconnects, used for running component-based applications
- Interconnects – signaling (control/data) paths that connect platforms
- Processor – an integrated circuit capable of executing a component-based application
- Application Worker – a component implementation that requires only abstracted data interfaces
- Container – execution environment on a processor that will execute workers
- Platform Configuration – a unique configuration of devices on a platform
- Control Plane – Worker “Life-Cycle” and “Property” access
- Data Plane – Container Data (DMA) Engine, “Ports” and “Protocols”
- Device Worker – an HDL component implementation that interfaces with I/O devices that are FPGA-external
- Platform Worker – Special type of Device Worker
- Assembly (sub-assembly) – a set of connected HDL Application Workers

Overview

- Review of Terminology
- **Systems vs Platforms ⇨ Platform Development**
- Enabling a new System (What’s the process?)
- Enabling Development Hosts (new Operating System)
- Enabling GPP (General Purpose Processor) Platforms
- Enabling FPGA (Field Programmable Gate Array) Platforms
  - Case Study: Epiq Matchstiq-Z1
  - Case Study: Device/Proxy Workers to support Lime MicroSystems Transceiver



Systems vs Platforms ⇨ Platform Development

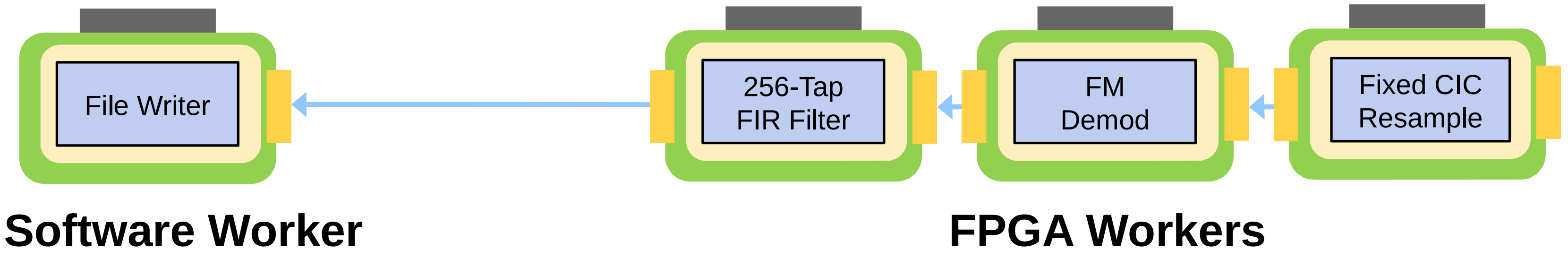


- **System** – a collection of processing elements (**Processor**) that can be used together as resources for running component-based applications.
- **Platform** – a **Processor** and its surrounding directly-connected hardware (memory and I/O devices).
- **Interconnect** – data paths that allow platforms to communicate with each other.
- Instead of enabling a “system”, OpenCPI focuses on enabling each “platform” and “interconnects” within a system.
- Hence **Platform Development** is enabling a <u>platform</u>, and enabling a <u>system</u> is enabling whatever <u>platforms</u> and <u>interconnects</u> are in the <u>system</u>.

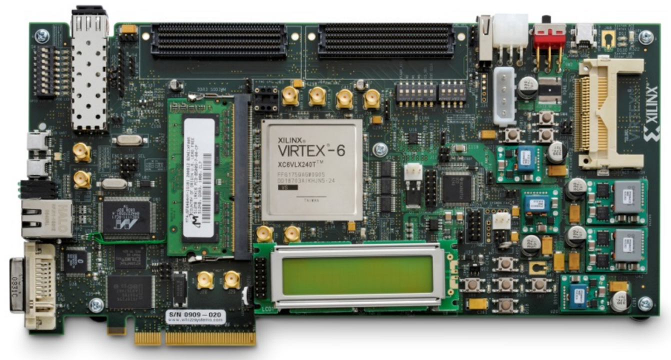
What is an OpenCPI FPGA Platform?

- An **OpenCPI FPGA Platform** is the FPGA, its surrounding infrastructure and attached devices.
- *Enabling the platform* allows it to be used for an OpenCPI application.
- Put another way…

What do I need to run an OpenCPI app…

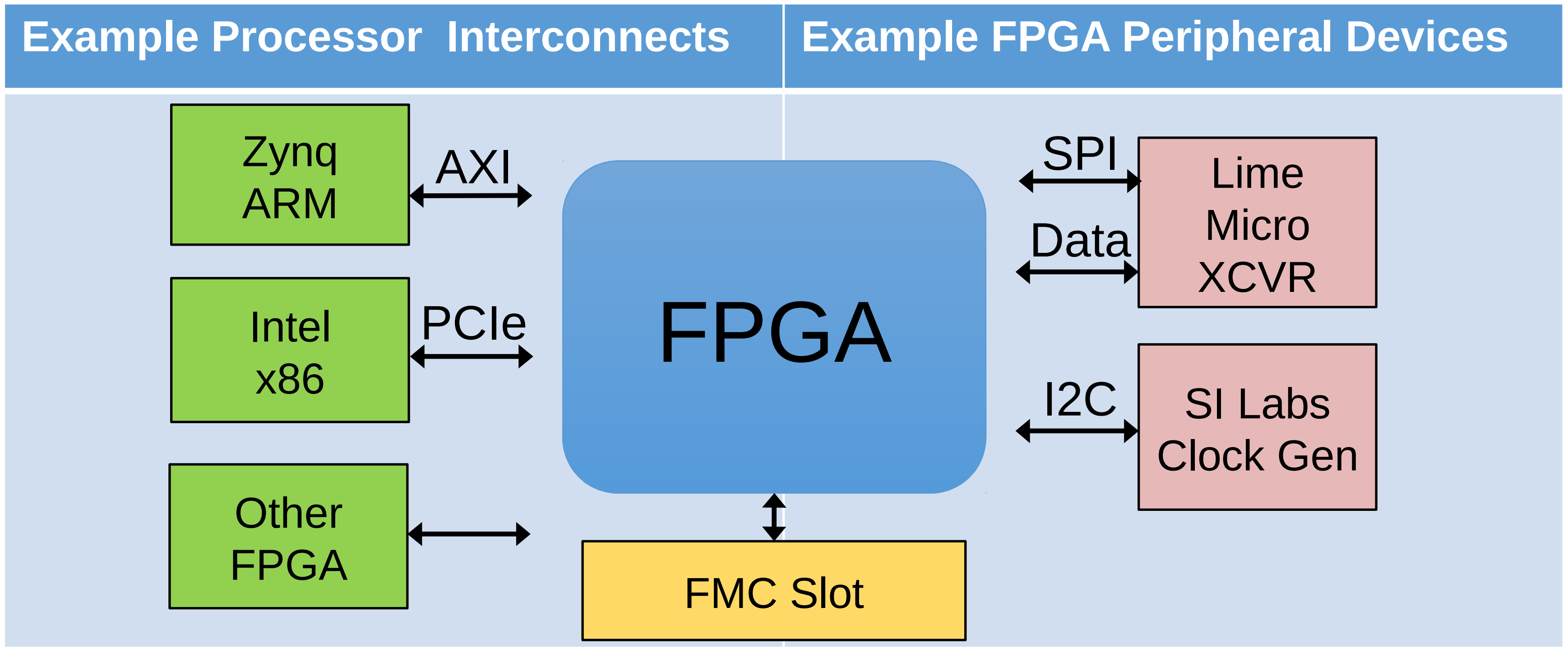


on *my* hardware?

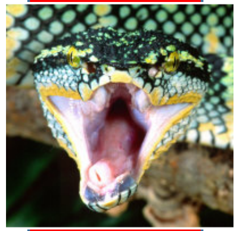


Review: Pieces of OpenCPI Platforms

1. ***The FPGA:*** a place where HDL application workers may execute
2. ***Interconnects:*** off-chip connections to processor(s) or other FPGA platforms
3. ***Devices:*** peripherals attached to the FPGA, useful to applications
4. ***Slots:*** hardware physical interfaces where ***cards*** (with devices) are plugged in



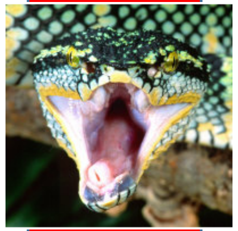
Inside an OpenCPI Platform



Platforms consist of a **Processor** with **Interconnects** allowing it to communicate with other platforms and two additional elements:

- **Devices**
  - Hardware elements locally attached to the processor
  - Allow source or sink of data flowing between components to enter or exit a system
- **Slots** (& Cards)
  - A means to support add-on **cards**
  - **Cards** – When plugged into a platform's slot, a card declares the use of additional devices

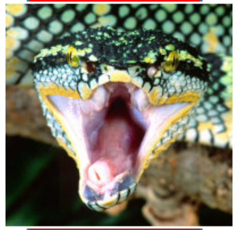
Examples of Supported Platforms



- Common CPU, Intel or AMD x86 on a motherboard
  - If cards are plugged into slots on the PC's motherboard, and those cards have **processors** (*e.g.* GPP, FPGA), then those cards can act as additional **platforms** in that **system** (*e.g.* Xilinx Virtex6 and Altera Stratix4 PCIe-based development boards)
  - Multi-core GPPs are single "processors" since they generally run a single operating system and act as a single resource that can run multiple threads concurrently

- ZedBoard from Digilent, based on Xilinx Zynq device
  - "System-On Chip" (SoC)
  - Two **processing** (AKA **platforms**) elements: Dual-core ARM and FPGA
  - **Interconnects**: AXI-4 interfaces (Master: GP 0/1, Slave: HP 0-3)
  - **Devices**: memory, pushbuttons, etc
  - **Slots**: FMC-LPC

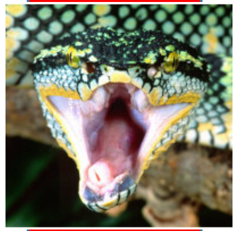
Development and Execution



Every platform must be enabled for:

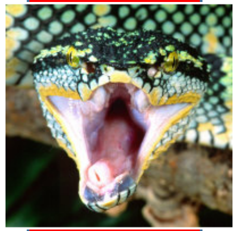
- Development - the process of producing “executable binaries”
  - Requires integrating compiler tool chains into OpenCPI’s build framework
  - Typically requires scripts or wrappers to enable tool operation within OpenCPI
  - Does not preclude or require GUI-based IDEs in the development process
  - Cross-compilers are leveraged for supporting embedded systems
- Execution – the process of running those binaries on the available platforms in a system

Overview



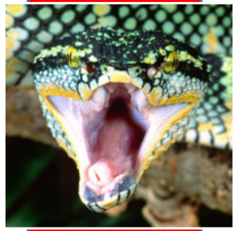
- Review of Terminology
- Systems vs Platforms ⇨ Platform Development
- **Enabling a new System (What’s the process?)**
- Enabling Development Hosts (new Operating System)
- Enabling GPP (General Purpose Processor) Platforms
- Enabling FPGA (Field Programmable Gate Array) Platforms
  - Case Study: Epiq Matchstiq-Z1
  - Case Study: Device/Proxy Workers to support Lime MicroSystems Transceiver

Enabling New Systems



- Enabling **systems** implies enabling **platforms** and **interconnects** in the system, as well as enabling **processors** and **devices** on the platforms and devices on any **cards** used in the system.
- What are the steps?
  - System Inventory – Which elements are relevant?
  - Assessment – What’s the current level of support?
  - Additional System-Level information
  - Experiments to fill in the knowledge gaps

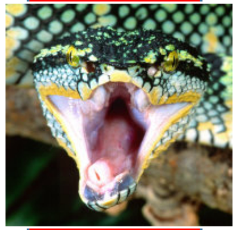
System Inventory



Identify hardware elements relevant to component-based applications:

- Processors AKA “Containers”
  - Elements capable of executing a worker
  - CPU (Intel AMD x86_64, Zynq-ARM), FPGA (Zynq-PL, Virtex6, Stratix4)
- Interconnects
  - Physical paths which connect Processors (PCIe, Ethernet)
- Devices
  - Devices connected to Processors
  - Examples: SPI, I2C, ADC, DAC, custom, memory, temperature, GPS

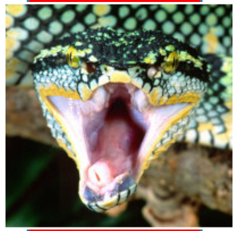
Assessment – Current Level of Support



For each:

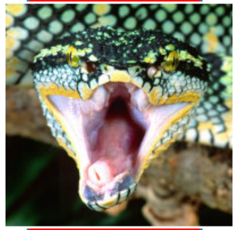
- Processor
  - What is the current level of support by the framework?: Is it already present, a variant, a new processor, or a new class of processor?
  - What is the current level of support by the tool chain?: Is it already present, does it require an update/variant, or is it missing?
- Interconnect – for each processor type that is attached to it
  - What is the current level of support by the framework?: Is it already present, does it require enhancements, or is it missing for the processor type?
- Device
  - What is the current level of support by the framework?: Is it already present, does it require enhancements, is it missing but similar to supported devices, or missing?

Gather/Create Technical Data Package



- Datasheets, ICDs, schematics, Connectivity (pinouts), signal timing, and memory maps
- Description of functions for each element
- Description of power sequences, boot sequences, initialization, reconfiguration, calibration, operation
- Event driven actions
  - *E.g.* temperature thresholds cause a throttling of the processors
- Timing
  - Oscillators drive synchronization (SW/HW) circuits, control and data clocks, etc.
  - Synchronization with GPS or reference clocks (1PPS, 10MHz distro)
  - Special calibration sequences
- Non-volatile media, removable media
- How are reprogrammable device programmed? (CPU, boot flash, JTAG)
- Debug ports
- Are there any “white wires” or other hardware errata?

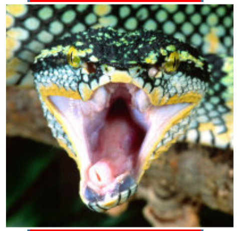
Experiments to Fill in Knowledge Gaps



- Reasons for performing experiments:
  - Uncertainties and gaps in technical documentation
  - Information is unavailable due to proprietary restrictions
  - Verify functional or performance capabilities of the system that are missing or questionable from the gathered information
  - Reverse engineer missing ICD aspects (assuming no legal impediments)

These efforts establish the final information to plan and budget for enabling a new system.

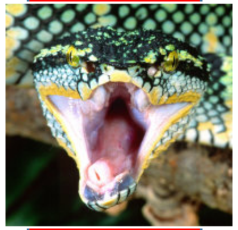
Now What?



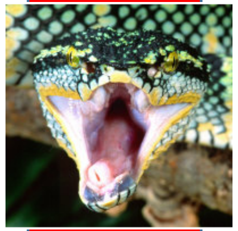
- Planning and Specification – Define requirements and tasking
- Technical Development – Execute tasks
- Verification – Testing
- “Contribute” – Back to OpenCPI repository to benefit others

Overview

- Review of Terminology
- Systems vs Platforms ⇨ Platform Development
- Enabling a new System (What’s the process?)
- **Enabling Development Hosts (new Operating System)**
- Enabling GPP (General Purpose Processor) Platforms
- Enabling FPGA (Field Programmable Gate Array) Platforms
  - Case Study: Epiq Matchstiq-Z1
  - Case Study: Device/Proxy Workers to support Lime MicroSystems Transceiver



Enabling Development Hosts



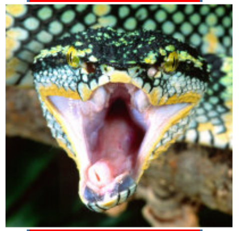
All steps shall be recorded for reproducibility!

1. Install OS with suitable options and packages
2. Install native development tools to drive OpenCPI's build system
    - make
    - git
    - C/C++ compiler tool chain
    - python
3. Retrieve current source code for OpenCPI

THE NEXT STEPS ARE THE SAME AS THOSE FOR SUPPORTING ANY RUNTIME PLATFORM AND ARE EXPLAINED IN THE NEXT SECTION "Enable GPP Platforms".

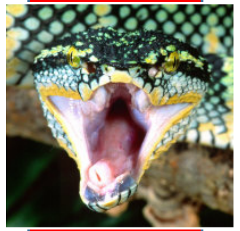
4. Establish a build environment with appropriate options
5. Build the core OpenCPI software targeting the development host
6. Build/Test OpenCPI component libraries and example applications

Overview



- Review of Terminology
- Systems vs Platforms ⇨ Platform Development
- Enabling a new System (What’s the process?)
- Enabling Development Hosts (new Operating System)
- **Enabling GPP (General Purpose Processor) Platforms**
- Enabling FPGA (Field Programmable Gate Array) Platforms
  - Case Study: Epiq Matchstiq-Z1
  - Case Study: Device/Proxy Workers to support Lime MicroSystems Transceiver

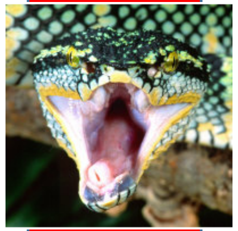
Enabling GPP Platforms



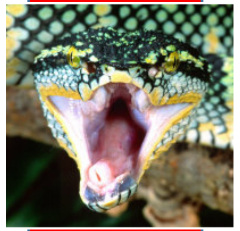
Two aspects:

- Development/Cross development
  - Building component and application binaries for the target platform
- Execution
  - Building the core OpenCPI runtime libraries and runtime command line utilities for the target platform

Development/Cross Development



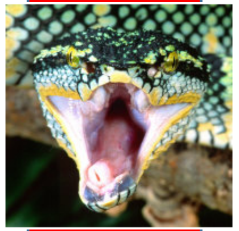
- Whether the GPP is native to the development host, or on an embedded system, the tool chain must be integrated into the OpenCPI build process
  - Cross-compilers are used for embedded systems
- Three ways development tools for GPP platforms are established:
  - Development host’s globally installed default tools, *e.g.* CentOS7, MacOS
  - A unique prerequisite installation in OpenCPI’s own installation tree (*e.g.* specific compiler dependency)
  - A “side-effect” of another tool installation, ex. Xilinx Zynq SOC (FPGA + dual core ARM)



Execution for GPP Platforms

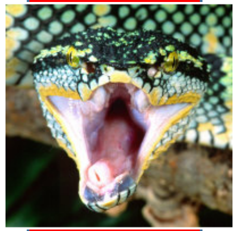
- Requires building the framework using the appropriate [cross-]compiler for the target GPP platform
  - Command line tools
  - Libraries
  - Device drivers
- Several runtime libraries have conditional compilation depending on the system or CPU being targeted
  - Significant Linux dependencies

Overview



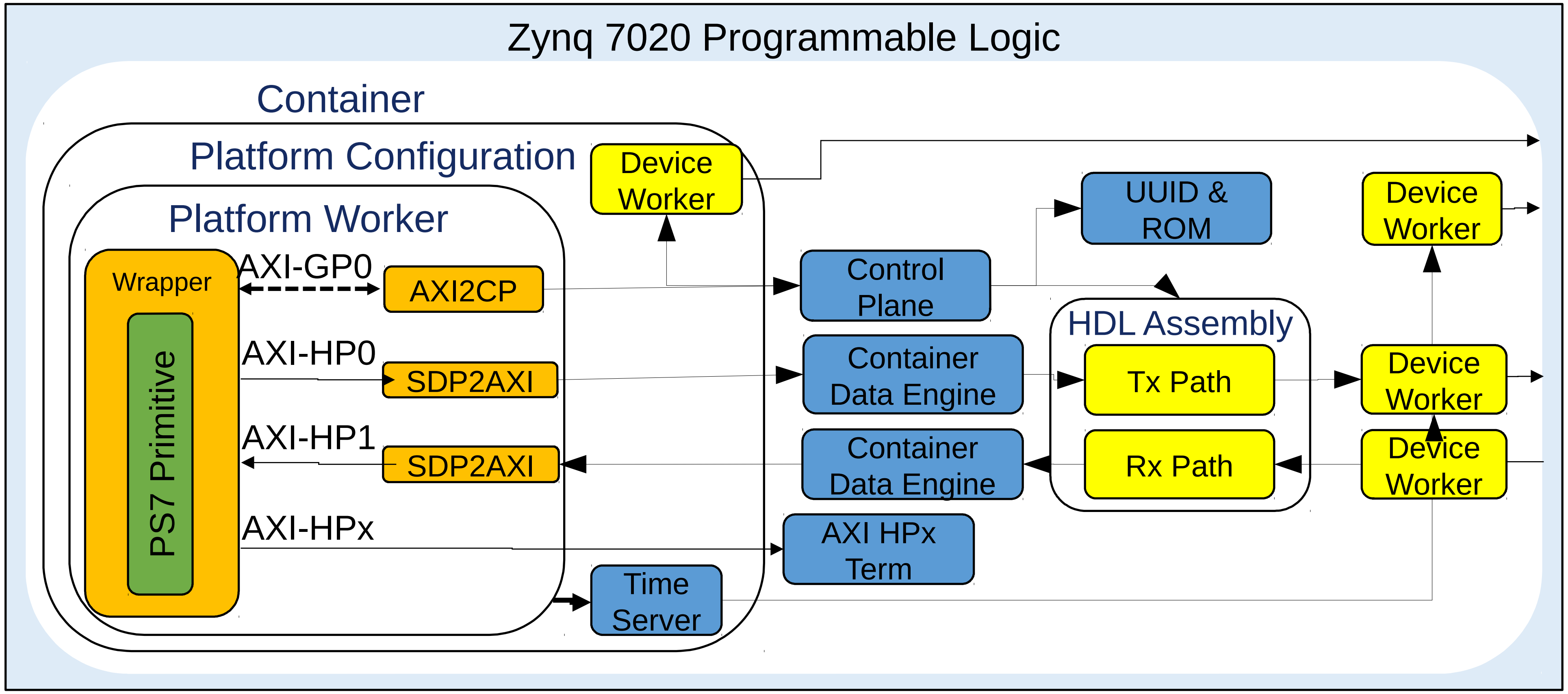
- Review of Terminology
- Systems vs Platforms ⇨ Platform Development
- Enabling a new System (What’s the process?)
- Enabling Development Hosts (new Operating System)
- Enabling GPP (General Purpose Processor) Platforms
- **Enabling FPGA (Field Programmable Gate Array) Platforms**
  - Case Study: Epiq Matchstiq-Z1 SDR
  - Case Study: Device/Proxy Workers to support Lime MicroSystems Transceiver

Enabling FPGA Platforms



- Two aspects enable an FPGA Platform
- Need to understand “What is an OpenCPI...”
  - ...FPGA Platform?
  - ...FPGA Platform Infrastructure?
  - ...Platform Worker?
  - ...Device Worker?
  - ...Slot?
  - ...Platform Configuration?
  - ...Container/Bitstream?
- Case Study:
  - Epiq Matchstiq-Z1 SDR
  - Device/Proxy Workers to support Lime MicroSystems Transceiver

Block Diagram of a Zynq-based Container



🟧 **Generic Infrastructure included with OpenCPI**

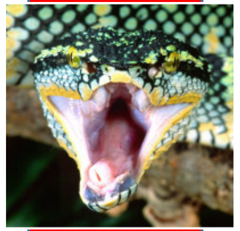
🟦 Generic Infrastructure included with OpenCPI – automatically instanced and connected, as needed, by the code-generation tool

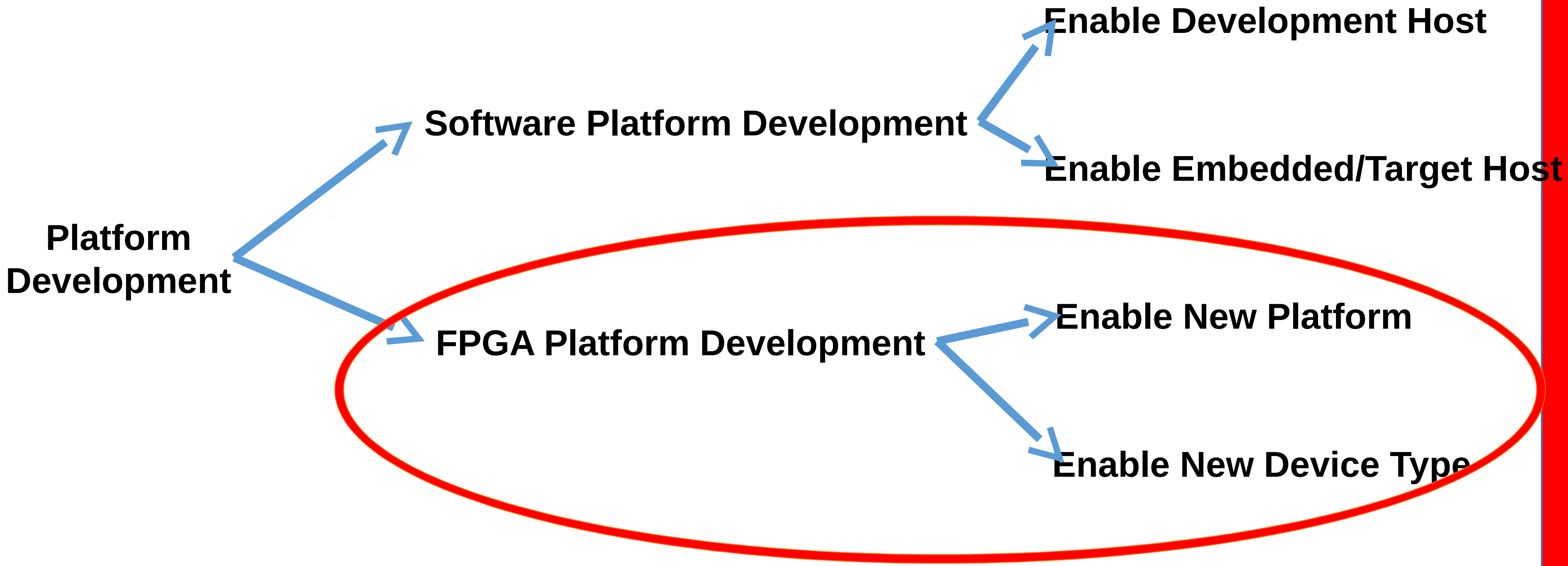
🟩 Vendor supplied hard IP block

🟨 Generic included with OpenCPI or to be developed

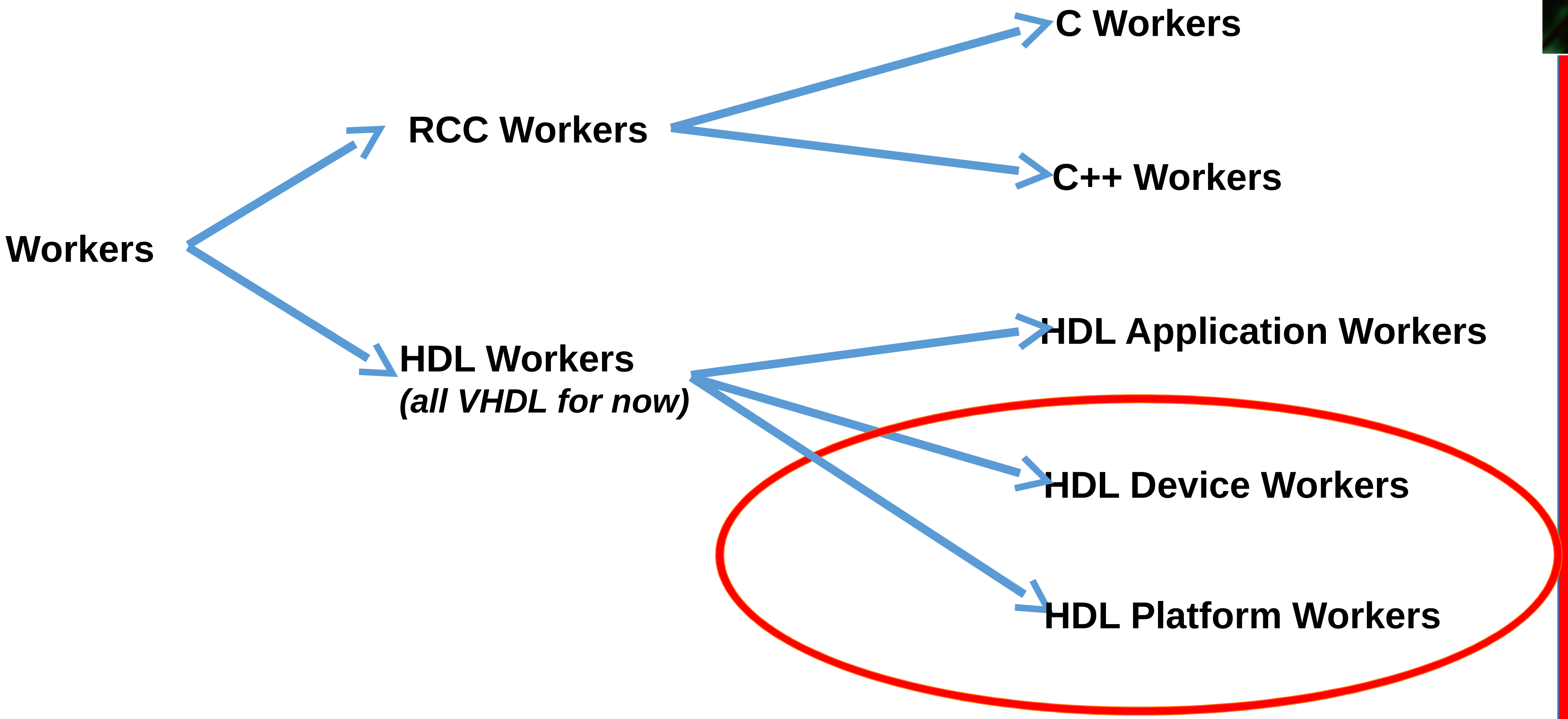
Types of OpenCPI Platform Development

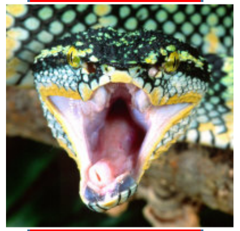
enabling new platforms and devices for OpenCPI apps



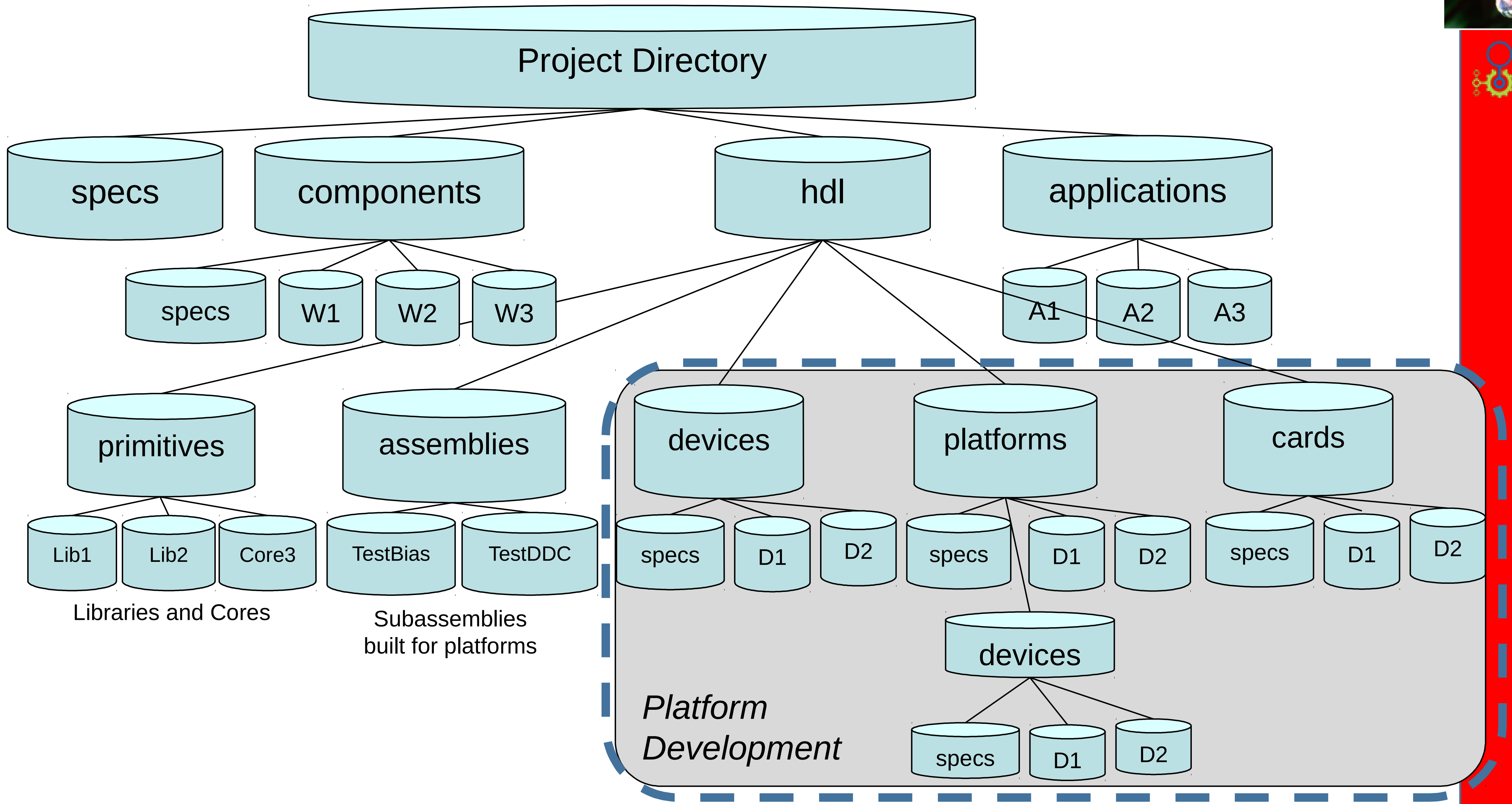


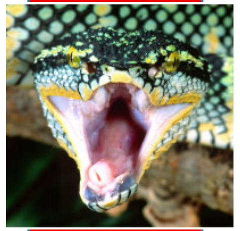
Types of OpenCPI Workers

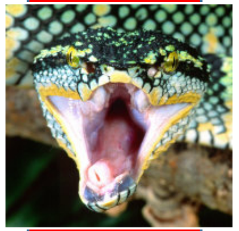




Project Directory Layout





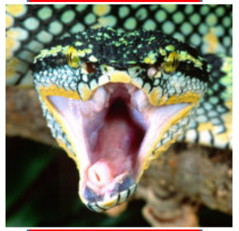


Enabling FPGA Platforms

Similar to enabling a GPP, there are two aspects to enable an FPGA:

- Development environment
  - Installation and integration of tool chain to target (build) the FPGA on the platform and core OpenCPI HDL code
- Execution
  - Writing specific new HDL code that supports the particulars of the hardware attached to the FPGA (except for simulator)
  - Verifying reference test applications on the platform

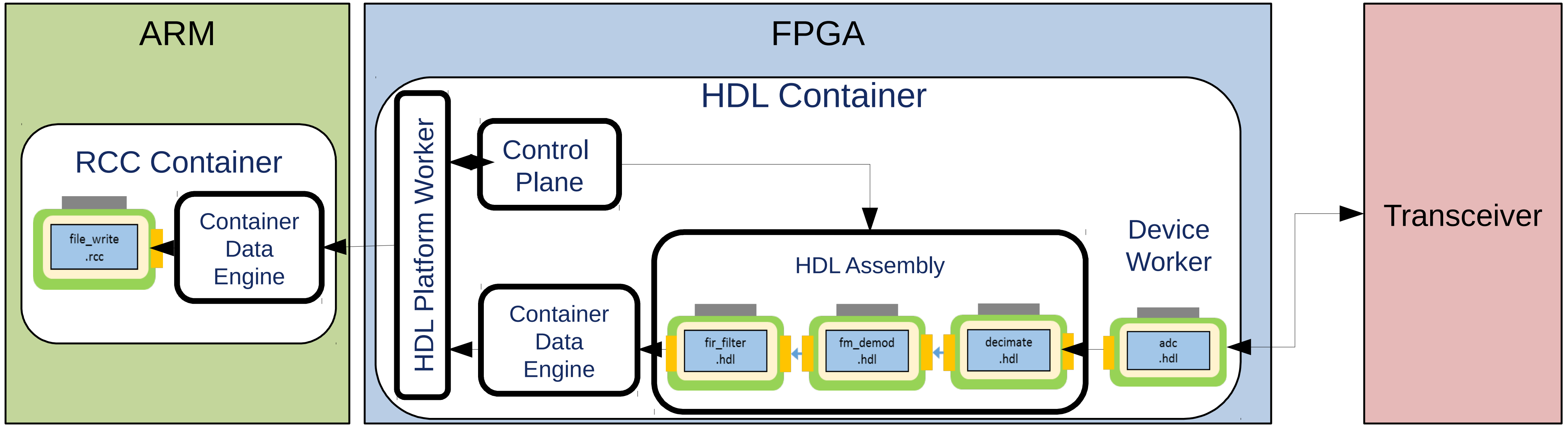
What is an “OpenCPI FPGA Platform”?



- A single FPGA on some board that has the infrastructure to serve as an OpenCPI Container to a component-based application
  - May have Devices and Slots
  - Ex. ZedBoard (Zynq 7020), Matchstiq-Z1 (Zynq 7020), Xilinx ML605, Altera Stratix4
- If a board has multiple FPGAs, then *each one* is an OpenCPI FPGA platform
  - Provided that enough resources remain after the OpenCPI infrastructure is in-place and the proper interconnects are present
  - Note: At this time, there are no working examples of platforms with multiple FPGAs
- An FPGA simulator is an FPGA platform
  - Where HDL components may execute, with all the same infrastructure as physical FPGA platforms (“co-simulation”) or a “bare-bones” infrastructure environment

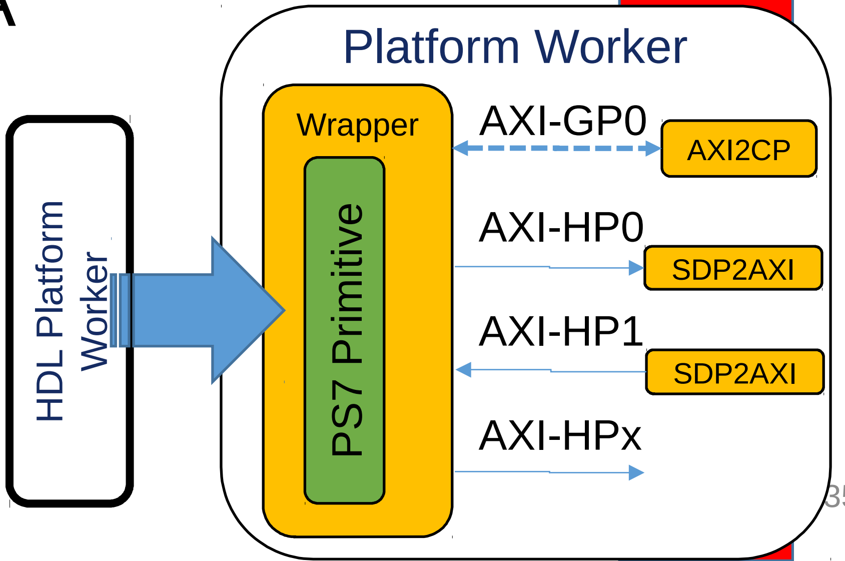
What is “OpenCPI FPGA Platform Infrastructure”?

- Typically comprised of the following OSS IP HDL modules
  - Platform Worker – Adapter for interfacing with Interconnects and internals
  - Control Plane – Local bus to interface with a workers control port
  - Data Plane – Pathways for workers to exchange data, and where the Container Data Engine(s) are used to pass data on/off the FPGA

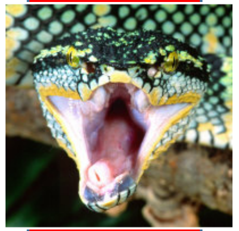


HDL Platform Worker

- A specific type of HDL Device Worker which describes additional information regarding hardware aspects of the platform
  - System-time, control, data
- Provides infrastructure to adapt external Interconnects to the internal control/data interfaces of the FPGA
- Implements Platform Spec (OCS)
  - {coreproject}/specs/platform-spec.xml
- Defined by XML and VHDL



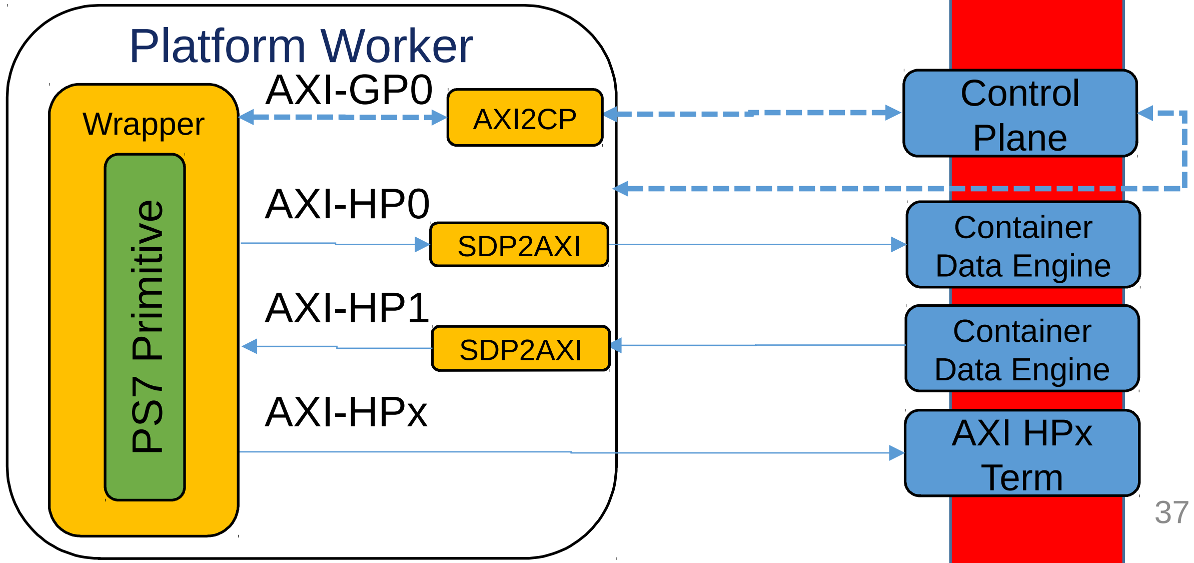
HDL Platform Worker - OWD



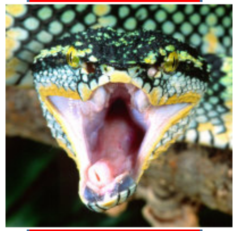
- Like all workers, the Platform Worker has an OWD XML file (and HDL) which defines requirements for the code-generation tool <HdlPlatform>
  - Configure platform <specproperty>, *e.g.* platform name
  - Define properties <property>
  - Declares signals (similar to Device Worker!) <signal>
  - Declares capabilities that are unique to Platform Workers
    - Master of <metadata>, <timebase>, control plane <cpmaster>, data plane <sdp>
    - Declare and parameterize frequency with which to operate time service module
    - Declares all possible <device> workers that are available in support of the platform
      - Does not mean that they will be built into bitstream, but required for the build engine
    - Defines Slots(s) <slot>
    - Signal re-mapping: platform signals to slot signals (and possibly device signals)
    - Defines Dev-Signals which are signals connected between device workers <devsignals>
- Unlike App/Device Workers, Platform Workers do not support DataInterfaces

HDL Platform Worker - VHDL

- Instances modules to adapt external Interconnects to internal control/data mechanisms
- System-level: reset and clocking (control plane and time server)
- PCIe-based platforms instance a vendor PCIe IP block (ex. Xilinx Block Plus Endpoint) module and an OpenCPI IP uNOC (micro-Network On A Chip) and adapters are used to convert the PCIe to the Control Plane and Container Data Engines
- Zynq-based platforms instance a processor module and OpenCPI IP adapters to convert AXI to the Control Plane and Container Data Engines infrastructure modules

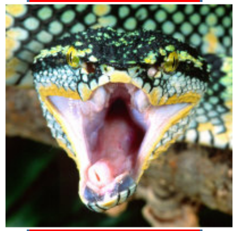


Control Plane

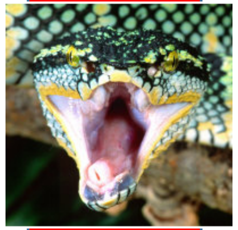


- Platform independent HDL module for reading and writing properties for all three types of HDL workers: App, Device, Platform
- Instanced in the auto-generated Container VHDL
- Scales according to the number of workers in the Container
- System-level provider of reset and clock for all workers (adapters)
  - Platform Worker sources reset and clock to Control Plane, then Control Plane sources to workers' control interfaces
- {coreproject}/hdl/primitives/platform/ocscp_rv.vhd

Data Plane



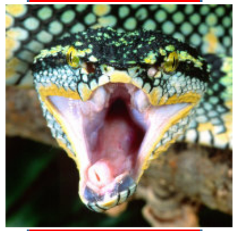
- Portable framework modules for moving data between workers and to/from containers
  - Container Data (DMA) Engine per stream
  - uNOC/SDP for multiplexing streams (maybe control plane) into an interconnect channel
  - Platforms worker's adaptation of these “interconnect access channels” to the system interconnect



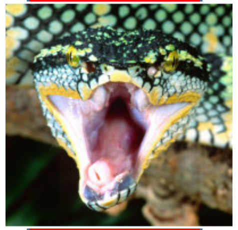
Slots (and Cards)

- Defined by XML (no HDL code)
- Characterized by
  - Physical connectors
  - Electrical signaling and direction
  - Pin and signal name assignments
- Common types (hdl/cards/spec/)
  - FMC (FPGA Mezzanine Cards, as defined by VITA 57 standard)
    - fmc_lpc.xml, fmc_hpc.xml
  - HSMC (High Speed Mezzanine Cards, as defined by Altera)
    - hsmc.xml, hsmc_alst4.xml

Cards (and Slots)



- Defined by XML (no HDL code)
- Instances additional Device Worker(s) that may be plugged into a Slot on various platforms
- Therefore devices may be directly attached to the pins of the platform FPGA, or they may exist on a plug-in card, that, when plugged into a slot, become attached to the platform FPGA
- Common types (hdl/cards/spec/)
  - Lime MicroSystems/Zipper
    - lime_zipper_hsmc.xml lime_zipper_fmc_hpc.xml, lime_zipper_fmc_lpc.xml
  - Analog Devices FMCOMMS3_EBZ
    - fmcomms3.xml



Platform Configuration

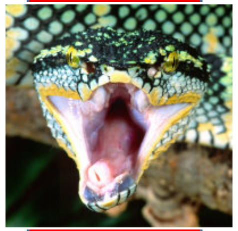
- XML that defines a platform with a particular set of devices
  - Top-level element <HdlConfig> and child <device> elements
- For devices mentioned in the Platform Worker’s OWD, the device element simply has a “name“ attribute indicating which of the platform’s devices that will be instanced in the platform configuration
- Can also specify devices that are on cards plugged into one of the platform’s slots using the “card“ attribute or “slot“ attribute when there are multiple cards plugged in

```
<HdlConfig>
        <device name=‘lime_dac’ />
        <device name=‘lime_adc’ card=‘lime-zipper-fmc’/>
</HdlConfig>
```

Container

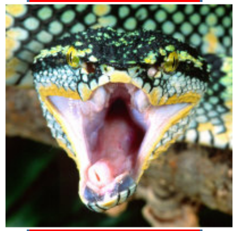
- Execution environment on some platform that will execute workers
- In HDL, the container is the complete design for an entire FPGA, including workers and infrastructure. Described by XML. Typically built inside of an HDL Assembly directory.
  - Container (BitStream) = Platform Configuration + Assembly + Device Worker(s)
- In RCC, the container loads, executes, controls, and moves data to/from RCC workers

```
<HdlContainer Platform=‘matchstiq_z1/matchstiq_z1_rx_tx'/>
	<Connection External='in' Device=‘lime_dac’ Port=‘in’/>
	<Connection External=‘out_to_dac' Device=‘lime_dac’ Port=‘in’/>
	<Connection External=‘in_from_adc' Device=‘lime_adc’ Port=‘out’/>
	<Connection External=‘out’ Interconnect=‘zynq’/>
</HdlContainer>
```

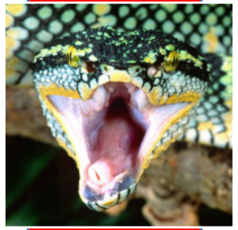


UUID and ROM

- Auto-generated VHDL which contains information used at execution and early in the process of platform enablement
- UUID
  - Unique identifier of the bitstream used by the framework during execution to confirm that the desired bitstream is loaded
- ROM
  - Compressed XML that describes the bitstream
  - Used for debug or as apart of an platform inspection process

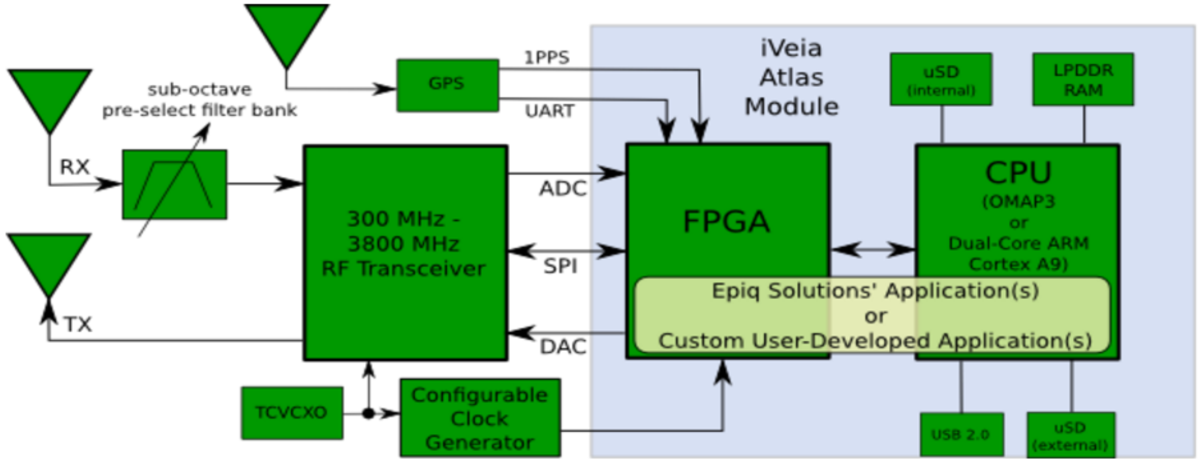


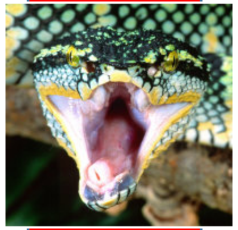
HDL Platform as It Relates to a Project



- A directory which contains files that implement an HDL Platform
  - Platform Worker’s OWD <HdlPlatform> and VHDL (arch)
  - Platform Configuration XML files <HdlConfig>
  - Device Workers that are unique to the platform (platform/devices/)
  - File specifying vendor-tool build-time options (.ut)
  - OpenCPI metadata files: Makefile, .mk
  - Vendor FPGA constraints files are managed at this level
    - Xilinx *.xdc, Altera *.qsf

Case Study:
Epiq Matchstiq-Z1



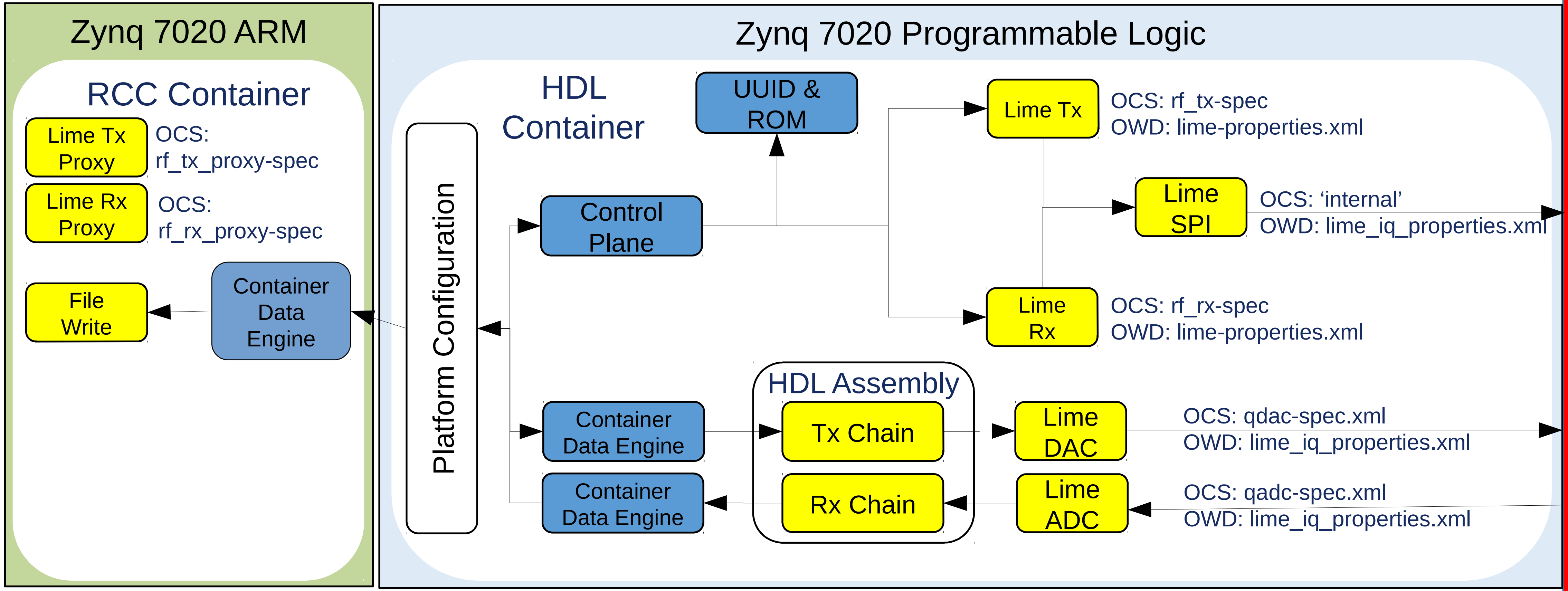


- Operating System : PetaLinux

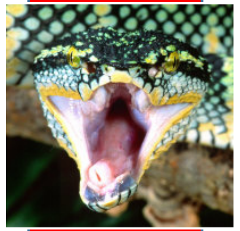
- Processors
  - Xilinx Zynq 7020 (Dual-core ARM processor and FPGA)
- Interconnects
  - AXI (PS ⇔ PL)
- Devices
  - Lime Microsystems Transceiver: Tx, Rx, ADC, DAC, SPI
  - I2C Bus: Temperature sensor, RF switch, RF step attenuator, clock synthesizer
  - GPS receiver: UART

Use Case: Device/Proxy Workers

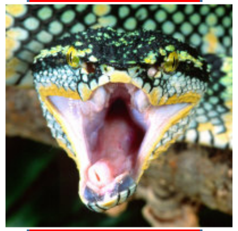
- To support Lime MicroSystems Transceiver



Backup

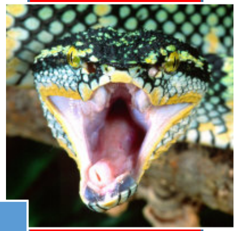


Create an HDL Platform



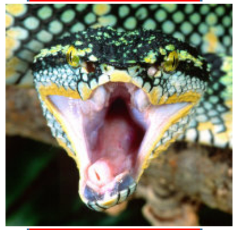
- Create an HDL Platform from the top-level of project
  - $ ocpidev create hdl platform my_platform
- hdl/platforms/my_platform/
  - my_platform.xml – OWD for the platform worker, which describes additional information regarding hardware aspects of the platform
  - my_platform.mk – Set make various variables, i.e. Exact FPGA device part number or OS target for embedded platforms
  - Makefile – Has an ‘include’ directive which indicates this is a platform
    - Include $(OCPI_CDK_DIR)/include/hdl/hdl-platform.mk

HDL Platform Directory - Makefile



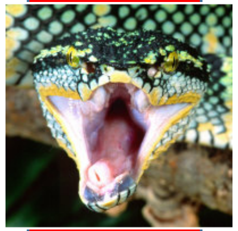
Variable Name	Override/ Augment Platform Library Makefile?	Description
SourceFiles	N	A list of additional source files for this worker (VHDL or Verilog)
Libraries	Y	A list of primitive libraries built elsewhere. If a name has no slashes, it will follow the HDL Search Path rules
Configurations	Y	A list of space-separated platform configuration XML files
ExportFiles	Y	A list of space-separated files to include as symbolic-links in the top-level *exports/*

HDL Platform Directory



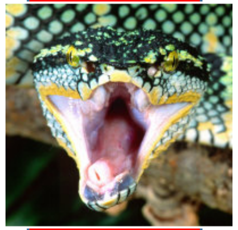
- To generate the skeleton platform worker VHDL file, must perform an initial build, from the hdl/platforms/my_platform/
  - $ make
    - <platform>.vhd – VHDL architecture file
- Other files to be added by developer
  - <platform>.ut – Xilinx specific build-time global options
  - <platform>.sdc – Altera specific build-time global options
  - <platform>.{xdc|qsf} – Vendor specific FPGA constraints file

Breakdown of Platform Worker Shell VHDL



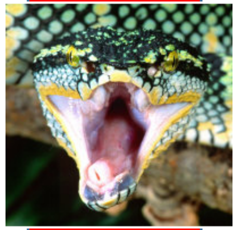
- Auto-generated VHDL based on the Platform Worker OWD XML file
- hdl/platforms/<platform>/gen/<platform>-impl.vhd (auto-gen entity/arch)
  - <platform> (entity/arch of the <platform>-impl.vhd. Wraps the <platform>_rv entity to expand the control interface records into signals which is required for subsequent code-generation processes when building for the Assembly)
    - <platform>_wci (auto-gen entity/arch)
    - <platform>_rv (auto-gen entity/arch “Record VHDL”)
      - <platform>_wci (auto-gen instance)
      - <platform>_worker (auto-gen entity)
        - <Your VHDL is written here!>.vhd (auto-gen arch shell for worker)
  - <platform>_worker_defs (auto-gen package that defines records for props_in, props_out, ctl_in, ctl_out, and memory map property)
    - DataInterfaces are not supported!

Breakdown of App/Dev Worker Shell VHDL



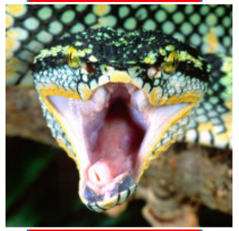
- <worker>/gen/<worker>-impl.vhd (auto-gen entity/arch)
  - <worker> (entity/arch of the <worker>-impl.vhd. Wraps the <worker>_rv entity to expand the control, in and out interfaces records into signals which is required for subsequent code-generation processes when building for the Assembly)
    - <worker>_wci (auto-gen entity/arch)
    - <worker>_rv (auto-gen entity/arch "Record VHDL")
      - <worker>_wci (auto-gen instance)
      - <worker>_wsi.slave (auto-gen instance)
      - <worker>_wsi.master (auto-gen instance)
      - <worker>_worker (auto-gen entity)
        - <Your VHDL is written here!>.vhd (auto-gen arch shell for worker)
  - <worker>_worker_defs (auto-gen package that defines records for props_in, props_out, ctl_in, ctl_out, in_in, out_out, and memory map property)

Hierarchy of the FPGA



- Container (Bitstream) = Platform Configuration + Assembly + Device Worker(s)
  - Auto or User-defined XML and auto-generated VHDL
- Platform Configuration(s) = Platform Worker + Time Server + Device Worker(s)
  - User-defined XML and auto-generated VHDL
- Platform Worker = IP Infrastructure HDL code + User-generated + Primitives + Slots
  - User-defined XML and auto-generated and user-generated VHDL
- Assembly = Application Worker(s)
  - User-defined XML and auto-generated VERILOG!
- Application and Device Workers = User-generated + Primitives
  - Auto or User-defined XML and auto-generated and user-generated VHDL
- Primitives
    - User-generated (ideally generic) VHDL, vendor tool generated, 3rd party, IP cores

Breakdown of the FPGA HDL



- Container (Bitstream) = Platform Configuration + Assembly + Device Worker(s)
  - Auto-generated -impl.vhd (entity) and -assy.vhd (arch)
- Platform Configuration(s) = Platform Worker + Time Server + Device Worker(s)
  - Auto-generated -impl.vhd (entity) and -assy.vhd (arch)
- Platform Worker = IP Infrastructure HDL code + User-generated + Primitives + Slots
  - Auto-generated -impl.vhd (entity) and -assy.vhd (arch) shell, with User-generated arch content
- Assembly = Application Worker(s)
  - Auto-generated -impl.vhd (entity) and -assy.v (arch) VERILOG!
- Application and Device Workers = User-generated + Primitives
  - Auto-generated -impl.vhd (entity) and <worker_name>.vhd (arch) shell, with User-generated arch content
- Primitives
  - User-generated (ideally generic) VHDL, vendor tool generated, 3rd party, IP cores