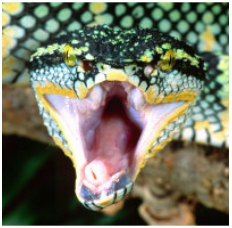


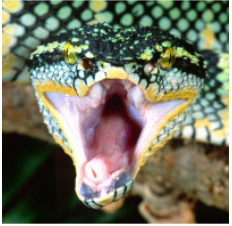
Lab 8: complex_mixer.hdl

HDL Application Worker
using 3rd Party Cores (Vivado IP)

Objective

- Design, Build and Test: **HDL** Application Worker
- Function: Complex Mixer (complex_mixer.hdl)
- OCS and OWD Properties (XML)
- Convert Data Port Interface: Interleaved to Parallel (OWD)
- Routes data/messages through the worker "pass-thru" (VHDL)
- Wraps Vivado IP (3rd party) cores (netlist, VHDL)
- Automated Unit Testing on multiple platforms (XML, Python)
 - Simulator: Xilinx XSIM
 - Hardware: Ettus E310 (Xilinx Zynq-7020)
- "Work-alike" to integrating a 3rd party LiquidDSP into an RCC Worker (Lab 4)





What's Provided

- **REUSE** "complex_mixer.test/" directory
 - Same as implemented in RCC Worker, 3rd Party LiquidDSP (Lab4)
 - Recall: One {component}.test/ per OCS
- Component Datasheet
 - /home/training/training_project/components/complex_mixer.test/doc/Complex_Mixer.pdf
- Worker VHDL with "commented" instructions
 - /home/training/provided/lab8/complex_mixer.vhd
- 3rd party core files: (Vivado IP output files)
 - complex_multiplier[_stub.vhd|.edf] and dds_compiler[_stub.vhd|.edf]
 - complex_multiplier_sim_net.vhd and dds_compiler_sim_net.vhd
 - /home/training/provided/lab8/vivado_ip/

App Worker Development Flow



- 1) Protocol (OPS): Use pre-existing or create new
- 2) Component (OCS): Use pre-existing or create new
- 3) Create new App Worker (Modify OWD, Makefile, and source HDL/RCC code)
- 4) Build the App Worker for target device(s)
- 5) Create Unit Test ({component}-test.xml, generate, verify and view scripts)
- 6) Build Unit Test
- 7) Run Unit Test

Step 1 – OPS: Use pre-existing or create new



- 1) Identify the OPS(s) declared by this component
 - Examine the "Component Ports" table in the Component Datasheet
- 2) Determine if OPS(s) exists
 - 1) Current project's component library?
/home/training/training_project/components/specs
 - 2) Other projects' components/specs/ directories within scope
Intersection of Project-registry and ProjectDependencies= in {my_project}/Project.mk
- 3) If NO to all questions \Rightarrow Create new OPS

ANSWER: REUSE! OPS XML file is available from framework

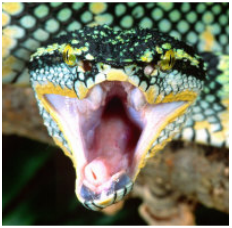
Step 2 – OCS: Use pre-existing or create new



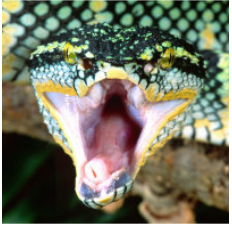
- 1) Review Component Spec Properties and Ports in Component Datasheet
 - Use Properties and Ports information to answer the following questions
- 2) Determine if OCS that satisfies the requirements exists
 - 1) Current project's component library?
/home/training/training_project/components/specs/
 - 2) Other projects' components/specs/ directories within scope
Intersection of Project-registry and ProjectDependencies= in {my_project}/Project.mk
- 3) If NO to all questions \Rightarrow Create new OCS

ANSWER: REUSE! OCS XML file is available from training_project

Step 3 - Create App Worker



- Via the IDE:
 - 1) **File** → **New** → **Other** → **ANGRYVIPER** → **OpenCPI Asset Wizard**
 - 2) Asset Type: **Worker**
 - 3) Worker Name: **complex_mixer**
 - 4) Add To Project: **ocpi.training**
 - 5) Model: **HDL**
 - 6) Programming Language: **VHDL**
 - 7) Add To Library: **components**
 - 8) Component: **complex_mixer (ocpi.training)**
 - 9) Finish
 - 10) Refresh the Project Explorer, then Refresh the OpenCPI Projects
 - 11) Use the Project Explorer to examine the auto-generated directories and files
 - components/{worker}.hdl/ - Worker directory with Author Model suffix (.hdl)
 - components/{worker}.hdl/Makefile - Includes a standard makefile fragment from the OCPI CDK
 - components/{worker}.hdl/{worker}.xml - OWD XML file
 - components/{worker}.hdl/{worker}.vhd - VHDL (architecture) skeleton file
 - components/{worker}.hdl/gen/ - OCPI worker build artifacts ({worker}-impl.vhd contains the Entity!)



Step 3 – Build Skeleton Code (WHY!?)

- Because...
 - Although not very exciting, this step proves the skeleton source code is build-able and the build engine is functional
- Build Generated Skeleton Code
 - 1) In the IDE, add the App Worker to the Project Operations panel
 - 2) Check the HDL Targets box and **highlight** "xsim" and "zynq", under "xilinx"
 - 3) Check "Assets" radio button
 - 4) Click "Build"
 - 5) Review the Console window messages to ensure this step is error free
 - 6) Refresh the Project Explorer panel and review the newly generated build artifacts in **target-xsim/** and **target-zynq/**

Step 3 – Add Properties to Worker



- Reference the "Worker Properties" table in the Component Datasheet
- Using IDE: OWD HDL Editor ("Design" tab), add properties:
 - data_select
 - Selects data to output when in bypass mode (input or NCO)
- Ensure all attributes and default values are set for each property/parameter

Step 3 – Configure Data Port for Parallel



- Convert data port interface from Interleaved to Parallel
 - Note: iqstream_protocol is default interleaved 16bit I/Q sample data
- Determine port configuration settings by examining the "Worker Interfaces" table in the Component Datasheet
- Edit OWD via the IDE: HDL App Worker OWD HDL Editor ("Design" tab)
 - 1) Highlight "Ports" from the "Outline"
 - 2) Click "Add a StreamInterface" and fill in the name "**in**" and set "DataWidth" to "**32**"
 - 3) Highlight "Ports" from the "Outline"
 - 4) Click "Add a StreamInterface" and fill in the name "**out**" and set "DataWidth" to "**32**"
- Expanding the data interface to 32 bit allows 16 bit I and 16 bit Q data to be transmitted simultaneously, i.e. parallel

Step 3 – Configure for Version 2 HDL API



- The v1.5 release of IDE does not provide a field for declaring Version 2 HDL API, so the XML must be manually modified

- 1) Edit OWD via the IDE: HDL App Worker OWD HDL Editor
- 2) Switch the tab from “Design” to “Source”
- 3) Add the top-level attribute Version=”2”
- 4) Add InsertEOM=”1” to the “out” port

```
<HdlWorker language='vhdl' spec='complex_mixer-spec' Version="2">  
  <Property Name="data_select" Type="bool" Default="false" Writable="true"></Property>  
  <StreamInterface Name="in" DataWidth="32"></StreamInterface>  
  <StreamInterface Name="in" DataWidth="32" InsertEOM="1"></StreamInterface>  
</HdlWorker>
```

Step 3 – Configure for Version 2 HDL API



- After the OWD has been configured for Version 2, the name of the architecture in the generated VHDL file must also be changed

- 1) Open the VHDL in a text editor
- 2) Change the name from “complex_mixer_worker” to “worker”

```
library IEEE; use IEEE.std_logic_1164.all; use ieee.numeric_std.all;
library ocpi; use ocpi.types.all; -- remove this to avoid all ocpi name collisions
architecture rtl of worker is
begin
    ctl_out.finished <= btrue; -- remove or change this line for worker to be finished when appropriate
    -- workers that are never "finished" need not drive this signal
    -- Skeleton assignments for agc_complex's volatile properties
end rtl;
```

Step 3 – Rebuild App Worker



- IDE: Build Worker with changes from OWD to update gen/*
 - 1) Refresh the OpenCPI Projects panel
 - 2) Add the App Worker to the Project Operations panel
 - 3) Check the HDL Targets box and **highlight** "xsim" and "zynq"
 - 4) Check "Assets" Radio Button
 - 5) Click "Build"
 - 6) Review the Console window messages to ensure this step is error free
 - 7) Examine gen/{worker}-impl.vhd to observe that the new OWD properties and ports modifications are now available

Step 3 – Copy 3rd Party Cores into Worker/



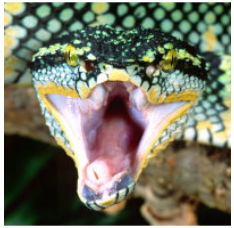
- Copy provided 3rd party cores directory into {worker}.hdl directory
 - 1) \$ `cd /home/training/training_project/components/complex_mixer.hdl/`
 - 2) \$ `cp -r /home/training/provided/lab8/vivado_ip/ .`
- Modify worker Makefile to reference the cores' VHDL files
 - Add the follow **PRIOR** to the “include...” statement:

```
SourceFiles=./vivado_ip/complex_multiplier_sim_net.vhd ./vivado_ip/dds_compiler_sim_net.vhd
```

```
include $(OCPI_CDK_DIR)/include/worker.mk
```

NOTE: Included source file are ONLY valid for targeting SIMULATORS

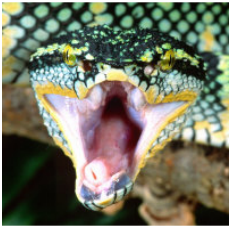
Step 3 - Create App Worker (cont)



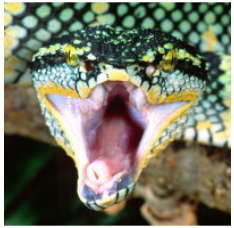
- The skeleton .vhd file is an empty architecture
 - The entity is generated VHDL based on OCS and OWD, and located in the gen/{worker}-impl.vhd
- Replace skeleton .vhd file with *provided* .vhd
 - Contains instructions to modify the VHDL source code
 - 1) \$ **cd /home/training/training_project/components/complex_mixer.hdl/**
 - 2) \$ **cp -f /home/training/provided/lab8/complex_mixer.vhd .**
- Open the VHDL in a text editor. Starting at the top, modify the source code per the commented instructions (look for “TODO”). A high-level summary of required modifications is provided:
 - **Make general signal and data port assignments**
 - **Wire signals to the 3rd Party IP Cores instantiations**

Step 4 - Build the App Worker

- Build HDL App Worker for **ONLY** XSIM Targets
 - 1) In the IDE, add the App Worker to the Project Operations panel
 - 2) Check the HDL Targets box and **highlight** "xsim"
 - 3) Check "Assets" Radio Button
 - 4) Click "Build"
 - 5) Review the Console window messages and address any errors



Step 4 – Review Build Logs and Artifacts



- End of build log should resemble the following, if free of errors:

```
Configuration:
build ocpi.training.components.complex_mixer.hdl HDL: xsim RCC: centos7 xilinx13_4

[ocpidev -d /home/training/training_project build worker complex_mixer.hdl -l components --hdl-target xsim]

make: Entering directory `/home/training/training_project/components/complex_mixer.hdl'
make[1]: Entering directory `/home/training/training_project/components'
make[1]: Leaving directory `/home/training/training_project/components'
Generating the VHDL constants file for config 0: target-xsim/generics.vhd
Generating the Verilog constants file for config 0: target-xsim/generics.vh
Building the complex_mixer worker for xsim (target-xsim/complex_mixer) 0:(complex_mixer_ocpi_max_opcode_out=0
complex_mixer_ocpi_max_bytes_in=8192 complex_mixer_ocpi_version=2 complex_mixer_ocpi_max_bytes_out=8192
complex_mixer_ocpi_endian=little complex_mixer_ocpi_max_latency_out=256 complex_mixer_ocpi_max_opcode_in=0
complex_mixer_ocpi_debug=false)
Tool "xsim" for target "xsim" succeeded. 0:13.93 at 11:43:57
Creating link to export worker binary: ../lib/hdl/xsim/complex_mixer -> target-xsim/complex_mixer
make: Leaving directory `/home/training/training_project/components/complex_mixer.hdl'
Updating project metadata...
== > Command completed. Rval = 0
```

- Observe new artifacts {worker}.hdl/: "target-xsim/"
 - This directory contains output files from their respective FPGA vendor tools (in this case, simply Xilinx Vivado) in addition to a framework auto-generated file, generics.vhd
 - "generics.vhd" contains parameter configuration settings of the HDL worker for the particular "target-"

Step 5(a) - 7(a) Unit Test - Simulation

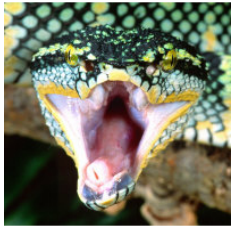


- Employ the framework's Unit Test Suite to generate:
 - OAS (OpenCPI Application Specification) XML file(s)
 - Used by the framework for running the executable on a simulation platform
 - In this case, the target simulation platform is Xilinx XSIM Simulation Server
 - OHAD (OpenCPI HDL Assembly Description) XML file(s)
 - Used by the framework to build an executable for the target simulation platform
 - In this case, the target simulation platform is Xilinx XSIM (xsim)
 - Input test data file(s) based on user provided scripts
 - Various scripts to manage the execution of the applications onto the target platform(s)

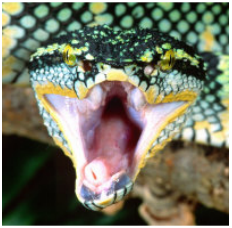
Step 5(a) - Create Unit Test

- **REUSE** from "Work-alike" 3rd party LiquidDSP (Lab4)
- Located in "complex_mixer.test/" directory
- Changes will be made to add the OWD property (next page)

```
<Tests UseHDLFileIo='true'>  
  <Input Port='in' Script='generate.py 100 12.5 32767 16384'></Input>  
  <Output Port='out' Script='verify.py 100 16384' View='view.sh'></Output>  
  <Property Name='phs_inc' Values='8192'></Property>  
  <Property Name='enable' Values='0,1'></Property>  
</Tests>
```



Step 5(a) – Edit test XML

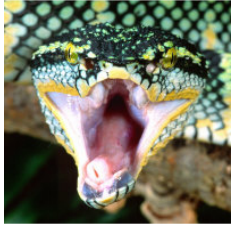


- Edit the unit test description file *complex_mixer-test.xml* to include HDL Worker specific property (OWD), as shown below, in bold:

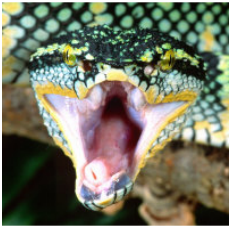
```
<Tests UseHDLFileIo='true'>
  <Input Port='in' Script='generate.py 100 12.5 32767 16384'></Input>
  <Output Port='out' Script='verify.py 100 16384' View='view.sh'></Output>
  <Property Name='phs_inc' Values='8192'></Property>
  <Property Name='enable' Values='0,1'></Property></Property>
  <Property Name="data_select" Values="0,1"/></Property>
</Tests>
```

- When "bypass" is enabled, "data_select" routes either the input data or NCO output to the output data port

Step 6(a) - Build Unit Test (Xilinx XSIM)



- Build Unit Test Suite for target simulation platform
 - 1) In the IDE, add the Unit Test to the Project Operations panel
 - 2) Highlight "xsim" the HDL Platforms panel (HDL Targets box unchecked)
 - 3) Check "Tests" radio button
 - 4) Click "gen + build"
 - 5) Review the Console window messages and address any errors
- Observe new artifacts in {OCS}.test/gen/
 - cases.txt – "Human-readable" file listing various test configurations
 - cases.xml – Used by framework to execute tests
 - cases.xml.deps – List of dependent files
 - applications/ - OAS files and scripts used by framework to execute applications
 - assemblies/ - Used by framework to build bitstreams



Step 6(a) – Review Build Artifacts (Xilinx XSIM)

- Observe new artifacts in `complex_mixer.test/gen/assemblies/complex_mixer_0_frw/`
 - **complex_mixer_0_frw.xml** – generated assembly xml (OHAD)
 - `gen/` - artifacts generated/used by framework
 - `lib/` - artifacts generated/used by framework
 - `target-xsim/` - artifacts generated/used by framework and FPGA tools
 - `container-complex_mixer_0_frw_xsim_base/`
 - `gen/` - artifacts generated/used by framework
 - `target_xsim/`
 - artifacts generated/used by framework and output files from FPGA tools
 - Execution file for execution onto a Simulation platform

Step 7(a) - Run Unit Test (Xilinx XSIM)



- Via IDE: Run Unit Test Suite for target simulation platform

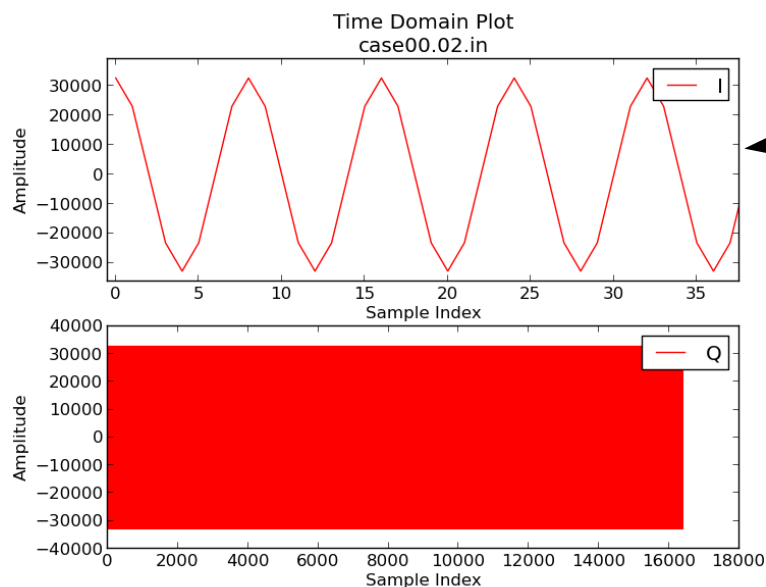
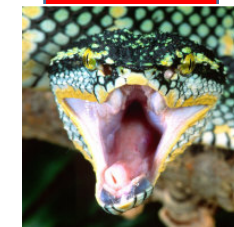
- 1) In the IDE, add the Unit Test (.test) to the Project Operations panel
- 2) Highlight "xsim" the HDL Platforms panel (HDL Targets box unchecked)
- 3) Check "keep simulations"
- 4) Check "run view script"
- 5) Click "prep+run+verify" to Run Tests
Simulation takes approximately 1 minute to complete.
- 6) Review the Console window messages and address any errors
Completion of each test case is reported in Console with a "PASSED".

- Via Command-line terminal:

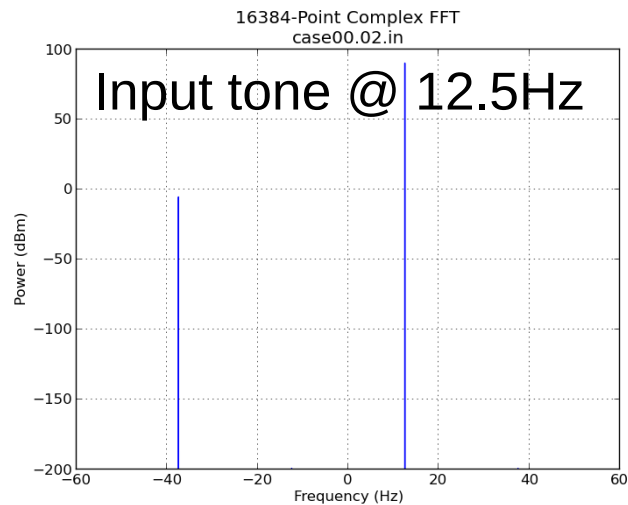
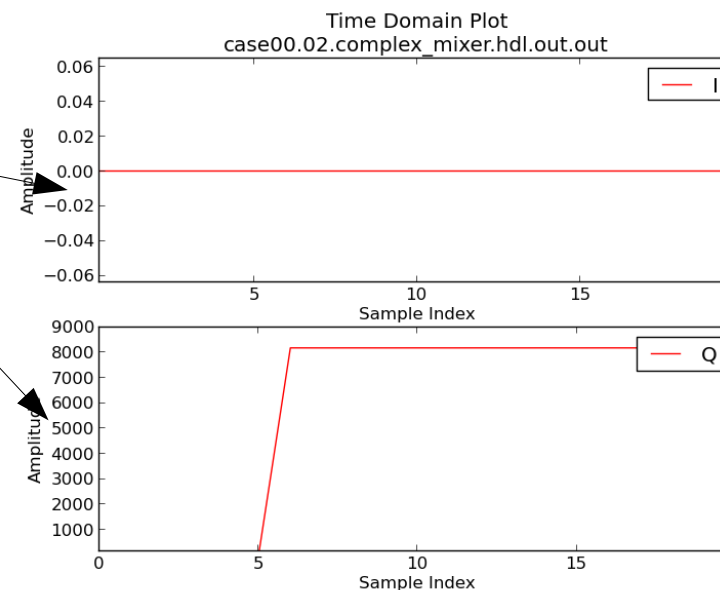
- In a terminal window, execute within the {component}.test/

```
$ ocpidev run --mode prep_run_verify --only-platform xsim --keep-simulations --view
```

Step 7(a) – Unit Test I/O Plots (Xilinx XSIM)

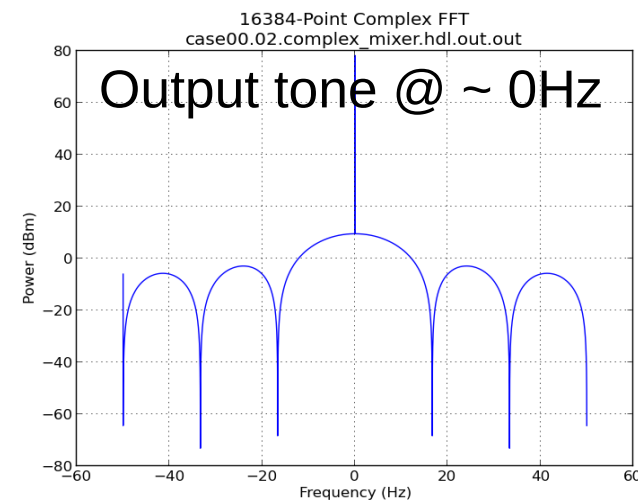


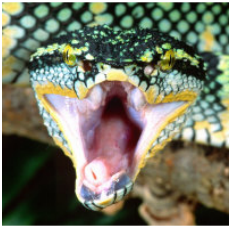
Note:
Zoomed-In



$\text{phs_inc} = 8192$

$\text{Fnc0} = 100\text{MHz} \times \text{phs_inc} / 65536$
 $\text{Fnc0} = -12.5 \text{ MHz}$

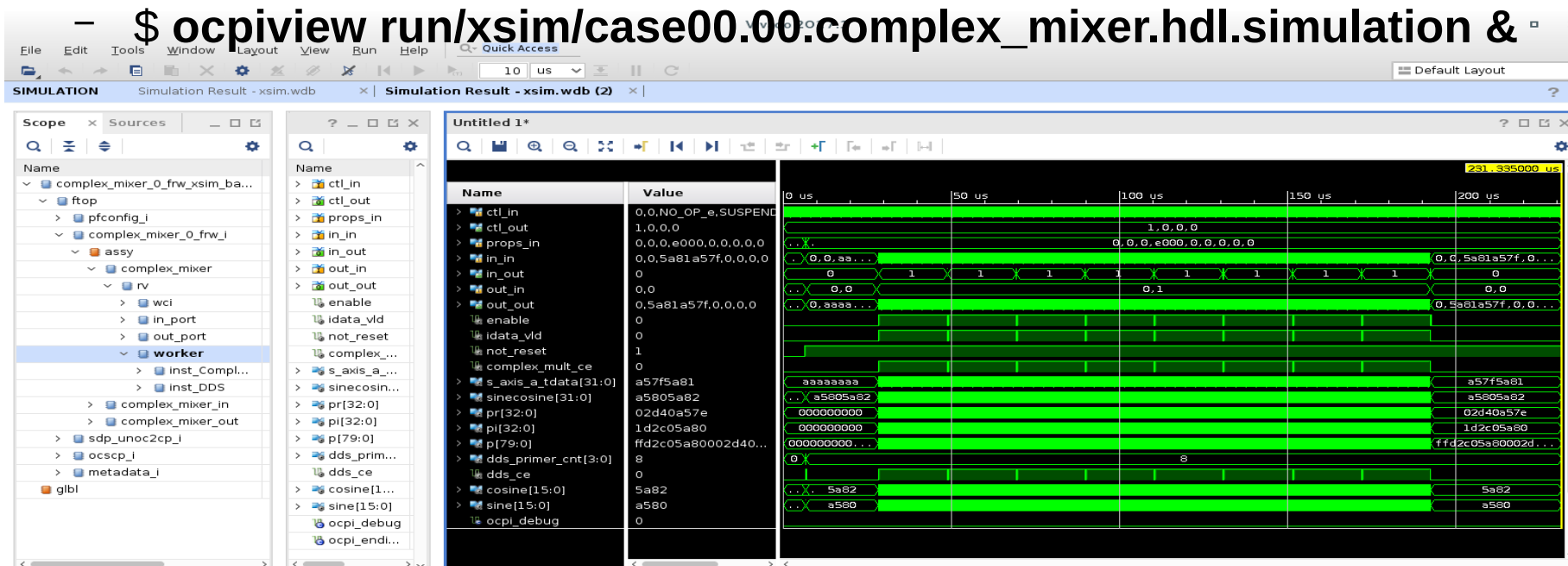




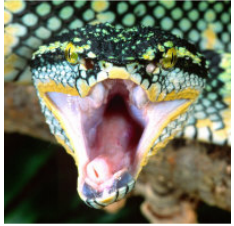
Step 7(a) – View Simulation Waveforms(Xilinx XSIM)

- Must have checked "keep simulations" or ran "make run OnlyPlatforms=xsim **KeepSimulations=1**"
- In a terminal window, browse to complex_mixer.test/ and execute

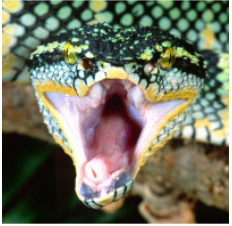
– `$ ocpiview run/xsim/case00.00.complex_mixer.hdl.simulation &`



Step 5(b) – 7(b) - Unit Test Ettus E310



- Employ the framework's Unit Test Suite to generate:
 - OAS (OpenCPI Application Specification) XML file(s)
 - Used by the framework for running the bitstream on hardware platform
 - In this case, the target hardware platform is Ettus E310
 - OHAD (OpenCPI HDL Assembly Description) XML file(s)
 - Used by the framework to build an bitstream for the target hardware platform
 - In this case, the target hardware platform is Ettus E310 (e3xx)
 - Input test data file(s) based on user provided scripts
 - Various scripts to manage the execution of the applications onto the target platform(s)



Backtrack to Step 3 - Create new App Worker

- Modify worker Makefile to reference the cores' VHDL files
 - **PRIOR** to the “include...” statement
 - **CHANGE** the **SourceFiles** and **Cores** variables to the following:

SourceFiles=../vivado_ip/complex_multiplier_stub.vhd ../vivado_ip/dds_compiler_stub.vhd

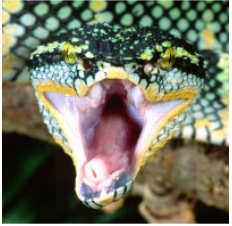
Cores=../vivado_ip/complex_multiplier.edf ../vivado_ip/dds_compiler.edf

include \$(OCPI_CDK_DIR)/include/worker.mk

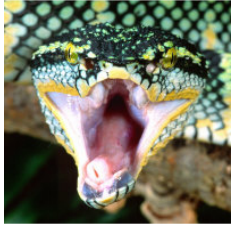
NOTE: Included source file are ONLY valid for targeting HARDWARE

Backtrack to Step 4 - Build App Worker

- Build HDL App Worker for **ONLY** Zynq Targets
 - 1) In the IDE, add the App Worker to the Project Operations panel
 - 2) Check the HDL Targets box and **highlight** "zynq"
 - 3) Check "Assets" Radio Button
 - 4) Click "Build"
 - 5) Review the Console window messages and address any errors



Backtrack to Step 4 – Review Logs/Artifacts



- End of build log should resemble the following, if free of errors:

Configuration:

build ocpi.training.components.complex_mixer.hdl HDL: e3xx RCC: centos7

```
[ocpidev -d /home/training/training_project build worker complex_mixer.hdl -l components --hdl-platform e3xx]
```

```
make: Entering directory `/home/training/training_project/components/complex_mixer.hdl'
```

```
make[1]: Entering directory `/home/training/training_project/components'
```

```
make[1]: Leaving directory `/home/training/training_project/components'
```

```
Building worker core "complex_mixer" for target "zynq" 0:(complex_mixer_ocpi_max_opcode_out=0 complex_mixer_ocpi_max_bytes_in=8192  
complex_mixer_ocpi_version=2 complex_mixer_ocpi_max_bytes_out=8192 complex_mixer_ocpi_endian=little complex_mixer_ocpi_max_latency_out=256  
complex_mixer_ocpi_max_opcode_in=0 complex_mixer_ocpi_debug=false) target-zynq/complex_mixer.edf
```

```
Tool "vivado" for target "zynq" succeeded. 0:37.32 at 12:26:19
```

```
Creating link to export worker binary: ../lib/hdl/zynq/complex_mixer.edf -> target-zynq/complex_mixer.edf
```

```
Creating link from ../lib/hdl/zynq/complex_mixer.cores -> target-zynq/complex_mixer.cores to expose the list of core dependencies for worker complex_mixer.
```

```
make: Leaving directory `/home/training/training_project/components/complex_mixer.hdl'
```

```
Updating project metadata...
```

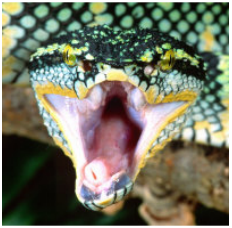
```
== > Command completed. Rval = 0
```

- Observe new artifacts {worker}.hdl/: "target-zynq/"
 - This directory contains output files from their respective FPGA vendor tools (in this case, simply Xilinx Vivado) in addition to a framework auto-generated file, generics.vhd
 - "generics.vhd" contains parameter configuration settings of the HDL worker for the particular "target-"

Step 5(b) - Create Unit Test

- **REUSE** form "Work-alike" 3rd Party Libraries (Lab 4)!
- **REUSE** from Simulation portion of this lab!
- Located in "complex_mixer.test/" directory
- No change required to Test XML

```
<Tests UseHDLFileIo='true'>  
  <Input Port='in' Script='generate.py 100 12.5 32767 16384'></Input>  
  <Output Port='out' Script='verify.py 100 16384' View='view.sh'></Output>  
  <Property Name='phs_inc' Values='8192'></Property>  
  <Property Name='enable' Values='0,1'></Property></Property>  
  <Property Name="data_select" Values="0,1"/></Property>  
</Tests>
```



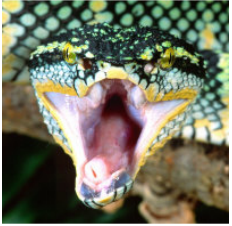
Step 6(b) - Build Unit Test (e3xx)

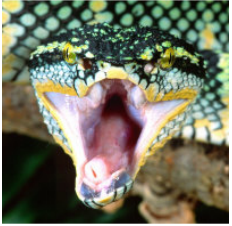


- Build Unit Test Suite for target hardware platform
 - 1) In the IDE, add the Unit Test to the Project Operations panel
 - 2) Highlight "e3xx" the HDL Platforms panel (HDL Targets box unchecked)
 - 3) Check "Tests" radio button
 - 4) Click "gen+build" to build tests
 - 5) Review the Console window messages and address any errors
- NOTE: The build process takes 5-10 mins to complete.

Step 6(b) – Review Build Logs (e3xx)

- Observe the console window to make sure the GUI completed successfully.





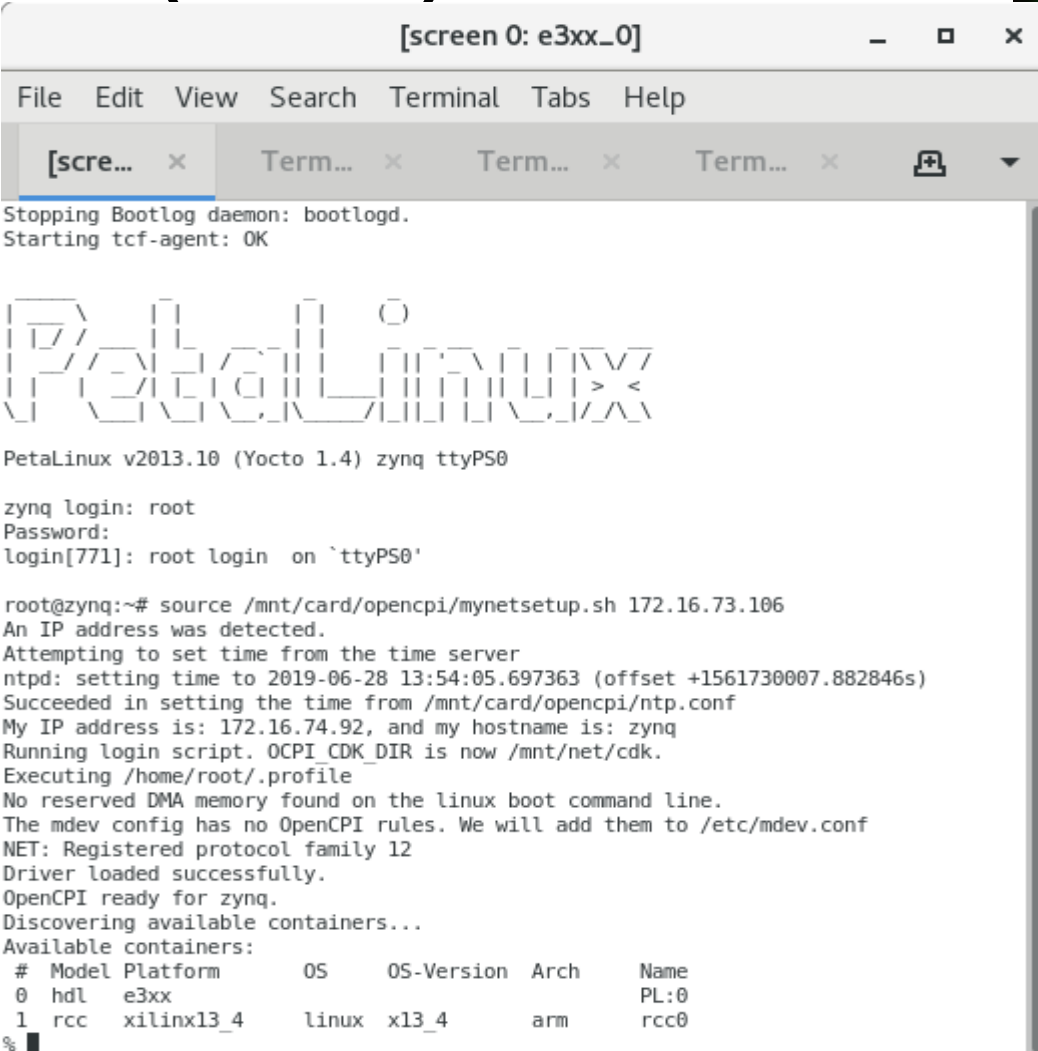
Step 6(b) – Review Build Artifacts (e3xx)

- Observe new artifacts in `complex_mixer.test/gen/assemblies/complex_mixer_0/`
 - **complex_mixer_0.xml** – generated assembly xml (OHAD)
 - `gen/` - artifacts generated/used by framework
 - `lib/` - artifacts generated/used by framework
 - `target-zynq/` - artifacts generated/used by framework and FPGA tools
 - `container-complex_mixer_0_e3xx_base/`
 - `gen/` - artifacts generated/used by framework
 - `target_zynq/`
 - artifacts generated/used by framework and output files from FPGA tools
 - Bitstream file for execution onto a hardware platform

Step 7(b) – Run Unit Test (e3xx)

- Setup deployment platform
 1. Connect to serial port via USB on rear of Ettus E310 on Host
 - "screen /dev/e3xx_0 115200"
 2. Boot and login into Petalinux on E310
 - User/Password = root:root
 3. Verify Host and E310 have valid IP addresses
 - For training, they should both be on the same subnet
 4. Run setup script on E310
 - "source /mnt/card/opencpi/mynetsetup.sh <Host ip address>"

More detail on this process can be found in the **E3xx Getting Started Guide** document

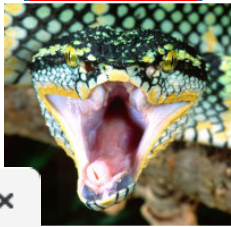


```
[screen 0: e3xx_0]
File Edit View Search Terminal Tabs Help
[scre... x Term... x Term... x Term... x +
Stopping Bootlog daemon: bootlogd.
Starting tcf-agent: OK

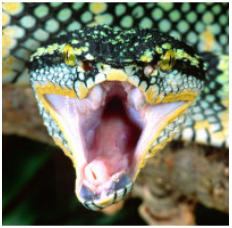
PetaLinux
PetaLinux v2013.10 (Yocto 1.4) zynq ttyPS0

zynq login: root
Password:
login[771]: root login on `ttyPS0'

root@zynq:~# source /mnt/card/opencpi/mynetsetup.sh 172.16.73.106
An IP address was detected.
Attempting to set time from the time server
ntpd: setting time to 2019-06-28 13:54:05.697363 (offset +1561730007.882846s)
Succeeded in setting the time from /mnt/card/opencpi/ntp.conf
My IP address is: 172.16.74.92, and my hostname is: zynq
Running login script. OCPI_CDK_DIR is now /mnt/net/cdk.
Executing /home/root/.profile
No reserved DMA memory found on the linux boot command line.
The mdev config has no OpenCPI rules. We will add them to /etc/mdev.conf
NET: Registered protocol family 12
Driver loaded successfully.
OpenCPI ready for zynq.
Discovering available containers...
Available containers:
# Model Platform OS OS-Version Arch Name
0 hdl e3xx linux x13_4 arm PL:0
1 rcc xilinx13_4 linux x13_4 arm rcc0
% █
```



Step 7(b) - Run Unit Test (Ettus E310)



- Via Command-line terminal:

- In a terminal window, execute within the {component}.test/

```
$ OCPI_REMOTE_TEST_SYSTEMS={IP of Ettus E310}=root=root/mnt_training \
```

```
ocpidev run --mode prep_run_verify --only-platform e3xx --keep-simulations --view
```

- Perform an md5sum to ensure the output of the XSIM and E310 tests are the same

complex_mixer.test\$ md5sum run/*/*.hdl.out*

- **72c9e2f951fc9bf87c274ccab9e54e4c** run/e3xx/case00.00.complex_mixer.hdl.out.out
 - **72c9e2f951fc9bf87c274ccab9e54e4c** run/xsim/case00.00.complex_mixer.hdl.out.out
 - **f46a3921b843a4a4f0b35a030c0c3e01** run/e3xx/case00.01.complex_mixer.hdl.out.out
 - **f46a3921b843a4a4f0b35a030c0c3e01** run/xsim/case00.01.complex_mixer.hdl.out.out
 - **c48a7f15859e47004ad0d0beb72a57e5** run/e3xx/case00.02.complex_mixer.hdl.out.out
 - **c48a7f15859e47004ad0d0beb72a57e5** run/xsim/case00.02.complex_mixer.hdl.out.out
 - **c48a7f15859e47004ad0d0beb72a57e5** run/xsim/case00.03.complex_mixer.hdl.out.out
 - **c48a7f15859e47004ad0d0beb72a57e5** run/e3xx/case00.03.complex_mixer.hdl.out.out