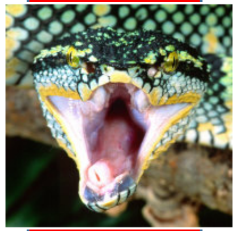
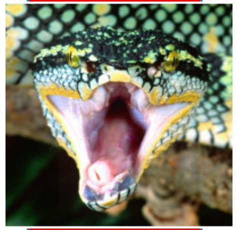
Lab 8:
HDL Application Worker using 3rd Party Cores (Vivado IP)



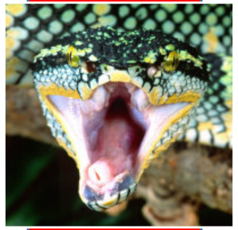
complex_mixer.hdl



Objective

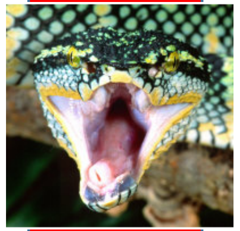
- Design, Build and Test: **HDL** Application Worker
- Function: Complex Mixer (complex_mixer.hdl)
- OCS and OWD Properties (XML)
- Convert Data Port Interface: Interleaved to Parallel (OWD)
- Routes data/messages through the worker “pass-thru” (VHDL)
- Wraps Vivado IP (3rd party) cores (netlist, VHDL)
- Automated Unit Testing on multiple platforms (XML, Python)
  - Simulator: Xilinx XSIM
  - Hardware: Matchstiq-Z1 (Xilinx Zynq-7020)
- ‘Work-alike’ to integrating a 3rd party LiquidDSP into an RCC Worker (Lab 4)

What’s Provided



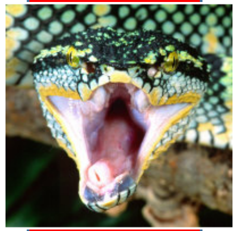
- **REUSE** “complex_mixer.test/”directory
  - Same as implemented in RCC Worker, 3rd Party LiquidDSP (Lab4)
  - Recall: One {component}.test/ per OCS
- Component Datasheet
  - /home/training/training_project/components/complex_mixer.test/doc/Complex_Mixer.pdf
- Worker VHDL with “commented” instructions
  - /home/training/provided/lab8/complex_mixer.vhd
- 3rd party core files: (Vivado IP output files)
  - complex_multiplier[_stub.vhd|.edf] and dds_complier[_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) OPS: Use pre-existing or create new
2) 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



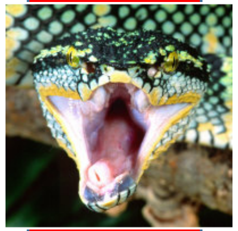
1) Identify the OPS(s) declared by this component
   - Examine the “Component Ports” table in the Component Datasheet
2) Determine if OPS(s) exists
   1) Current project's component library?

      /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 ⇨ Create new OPS

ANSWER: REUSE! OPS XML file is available from framework

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



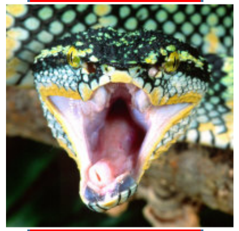
1) Review Component Spec Properties and Ports in Component Datasheet
   - Use Properties and Ports information to answer the following questions
2) Determine if OCS that satisfies the requirements exists
   1) Current project's component library?

      /home/training/training_project/components/specs/
   2) Other projects' components/specs/ directories within scope

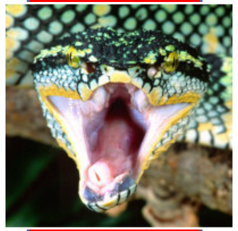
      Intersection of Project-registry and ProjectDependencies= in {my_project}/Project.mk
3) If NO to all questions ⇨ Create new OCS

ANSWER: REUSE! OCS XML file is available from training_project

Step 3 - Create App Worker



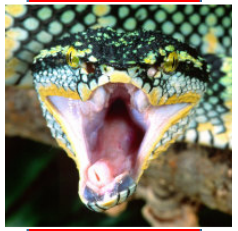
- **IDE: File → New → Other → ANGRYVIPER → OpenCPI Asset Wizard**
  - **Asset Type: Worker**
  - Worker Name: **complex_mixer**
  - Add to Project: **training_project**
  - Add To Library: **components (default)**
  - Component: **complex_mixer-spec (ocpi.training)**
  - Model: **HDL**
  - Programming Language: **VHDL**
- **IDE: Refresh** the Project Explorer, then **Refresh** the OpenCPI Projects
- 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

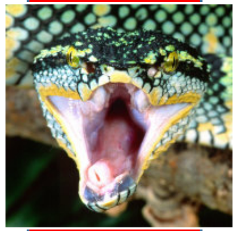
- Generated skeleton code can be built!?
  1) **IDE:** “**Refresh**” the OpenCPI Projects panel
  2) Use the IDE to “**Add**” the App Worker to the Project Operations panel
  3) **Check** the <u>HDL Targets</u> box and **highlight** “xsim” and “zynq”, under “xilinx”
  4) **Check** “Assets” Radio button
  5) **Click** “Build”
  6) Review the Console window messages to ensure this step is error free
- New Build Artifacts in **target-xsim/** and **target-zynq/**
- Although not very exciting, this step proves the skeleton source code is build-able and the build engine is functional

Step 3 – Add Properties to Worker



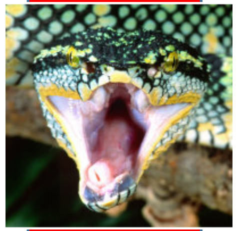
- Reference the “Worker Properties” table in the Component Datasheet
- Using IDE: OWD HDL Editor (“Design” tab), add properties:
  - data_select
    - Selects data to output when in bypass mode (input or NCO)
- Ensure all attributes and default values are set for each property/parameter

Step 3 – Convert Data Port to 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: OWD HDL Editor (“Design” tab)
  1) “**Add a StreamInterface**” port named “**in**” and set “DataWidth” to **32**
  2) “**Add a StreamInterface**” port named “**out**” and set “DataWidth” to **32**
- Expanding the data interface to 32 bit allows 16bit I and 16bit Q data to be transmitted simultaneously, I.e parallel

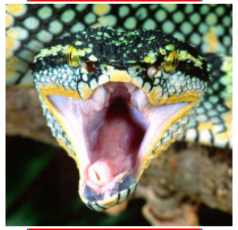
Step 3 – Rebuild App Worker



- Build Worker with changes from OWD.
  1) "**Refresh**" the OpenCPI Projects panel
  2) Use the IDE to "**Add**" the App Worker to the Project Operations panel
  3) **Check** the <u>HDL Targets</u> 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

- Notice that the new properties and ports modifications are now added in the gen/complex_mixer-impl.vhd.

Step 3 – Copy 3rd Party Cores into Worker/



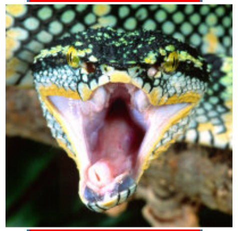
- Copy provided 3rd party cores directory into {worker}.hdl directory
  1) $ **cd /home/training/training_project/components/complex_mixer.hdl/**
  2) $ **cp -r /home/training/provided/lab8/vivado_ip/ .**

- Modify worker Makefile to reference the cores' VHDL files
  - Add the follow **PRIOR** to the include statement:

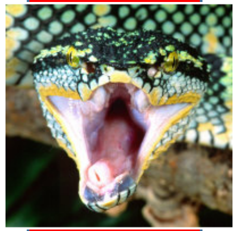
```
SourceFiles=./vivado_ip/complex_multiplier_sim_net.vhd ./vivado_ip/dds_compiler_sim_net.vhd
```

NOTE: ONLY VALID FOR TARGETING SIMULATORS

Step 3 - Create App Worker (cont)



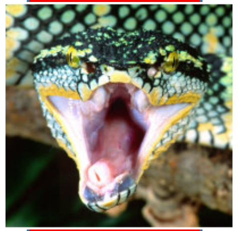
- The skeleton .vhd file is an empty architecture
  - The entity is generated VHDL based on OCS and OWD, and located in the gen/{worker}-impl.vhd
- Replace skeleton .vhd file with *provided* .vhd
  - Contains instructions to modify the VHDL source code
  1) $ **cd /home/training/training_project/components/complex_mixer.hdl/**
  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. A 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 App Worker

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

Step 4 – Review Build Logs and Artifacts

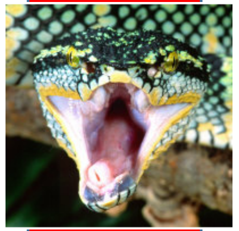


- End of build log should resemble the following, if free of errors:

```
[ocpidev -d /data/test_training build worker complex_mixer.hdl -l components --hdl-target xsim]

make: Entering directory `/data/test_training/components/complex_mixer.hdl'
Generating the definition file: gen/complex_mixer-defs.vhd
Generating the implementation header file: gen/complex_mixer-impl.vhd from complex_mixer.xml
Generating the implementation skeleton file: gen/complex_mixer-skel.vhd
Generating the VHDL constants file for config 0: target-xsim/generics.vhd
Generating the opposite language definition file: gen/complex_mixer-defs.vh
Generating the Verilog constants file for config 0: target-xsim/generics.vh
Building the complex_mixer worker for xsim (target-xsim/complex_mixer) 0:(ocpi_debug=false ocpi_endian=little)
 Tool "xsim" for target "xsim" succeeded.  0:13.66 at 12:32:40
Creating link to export worker binary: ../lib/hdl/xsim/complex_mixer -> target-xsim/complex_mixer
make: Leaving directory `/data/test_training/components/complex_mixer.hdl'
== > Command completed. Rval = 0
```

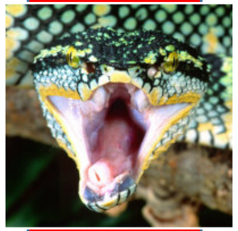
- Observe new artifacts {worker}.hdl/: “target-xsim/”
  - This directory contains output files from their respective FPGA vendor tools (in this case, simply Xilinx Vivado) in addition to a framework auto-generated file, generics.vhd
  - ‘generics.vhd’ contains parameter configuration settings of the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

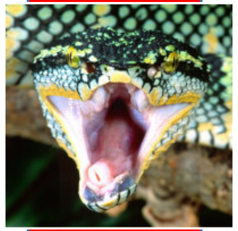
Step 5(a) - Create Unit Test



- **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’ messagesize=‘8192’/>
 <output port=‘out’ script=‘verify.py 100 16384’ view=‘view.sh’/>
 <property name=‘phs_inc’ values=‘-8192’/>
 <property name=‘enable’ values=‘0,1’/>
<tests>
```

Step 5(a) – Edit test XML

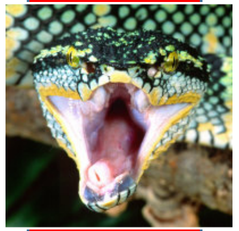


- Edit the unit test description file *complex_mixer-test.xml* to include HDL Worker specific property (OWD), as shown below, in bold:

```
<tests useHDLFileIO=”true”>
 <input port=”in” script=”generate.py 100 12.5 32767 16384” messagesize=”8192”/>
 <output port=”out” script=”verify.py 100 16384” view=”view.sh”/>
 <property name=”phs_inc” values=”-8192”/>
 <property name=”enable” values=”0,1”/>
 <property name=”data_select” values=”0,1”/>
<tests>
```

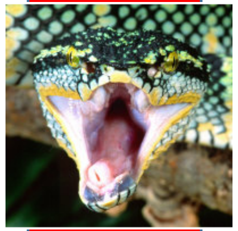
- When “bypass” is enabled, “data_select” routes either the input data or NCO output to the output data port

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



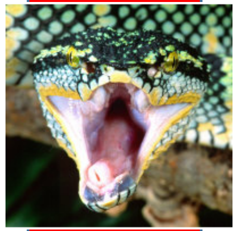
- Build Unit Test Suite for target simulation platform
  1) Use the IDE to "**Add**" the Unit Test to the Project Operations panel
  2) **Highlight** "xsim" the HDL Platforms panel (HDL Targets box unchecked)
  3) **Check "**Tests" Radio button
  4) **Click** "gen + build"
  5) Review the Console window messages and address any errors
- Observe new artifacts in complex_mixer.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)



- Run Unit Test Suite for target simulation platform
  1) Use the IDE to "**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) **Check** "run view script" to see the output of the view script
  5) **Check** " keep simulations" so that we can see the waveform output later.
  6) **Click** "prep + run + verify" to run the tests
  7) Review the Console window messages and address any errors
- Simulation takes approximately 1 minute to complete. Completion of each test case is reported in Console with a "PASSED" along with final values for the min/max peaks.
- Other operations not currently supported by IDE.
  In a terminal window, execute within the {component}.test/

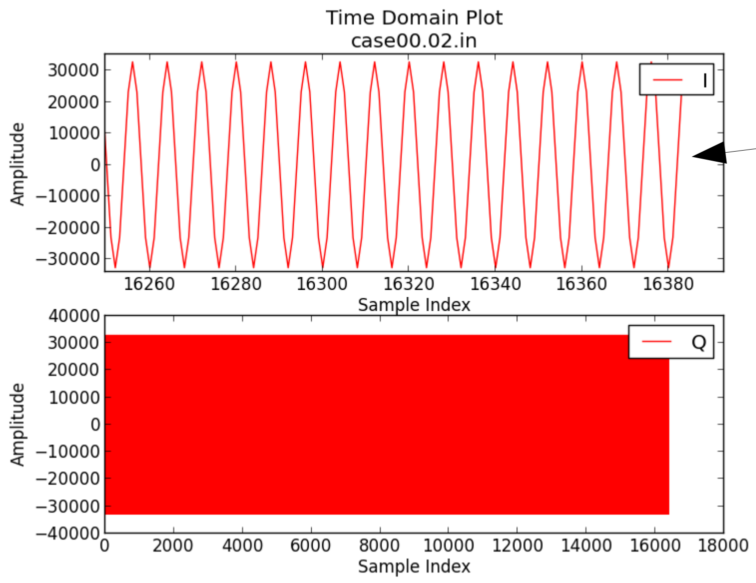
  $ **make run Cases=case00.01** {run specific case/subcase}

  $ **make run Cases=case00.0*** {run all subcases of case00}

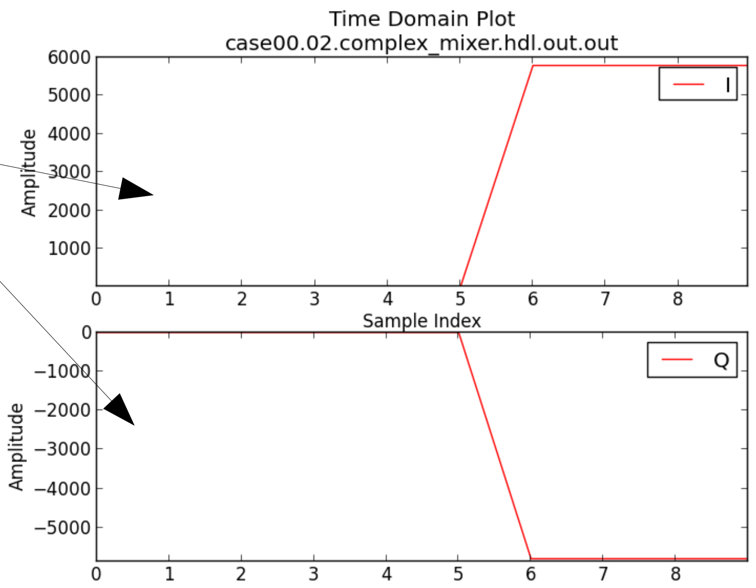
  $ **make verify Cases='case00.01 case00.03'** {verify previous results of specific cases}

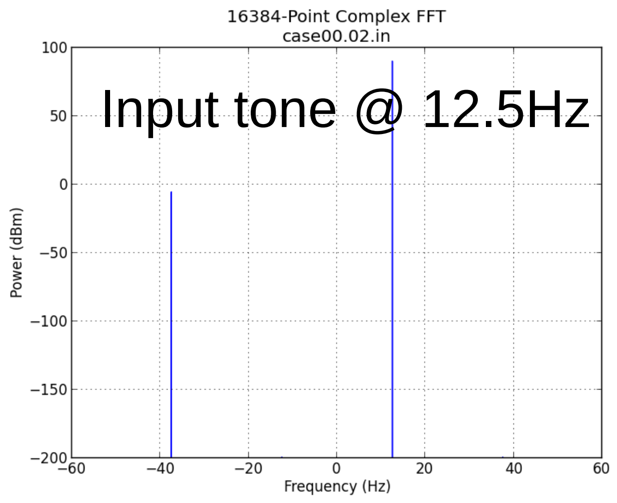
  $ **make view Cases=case00.01** {verify previous results}

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



Note:
Zoomed-In

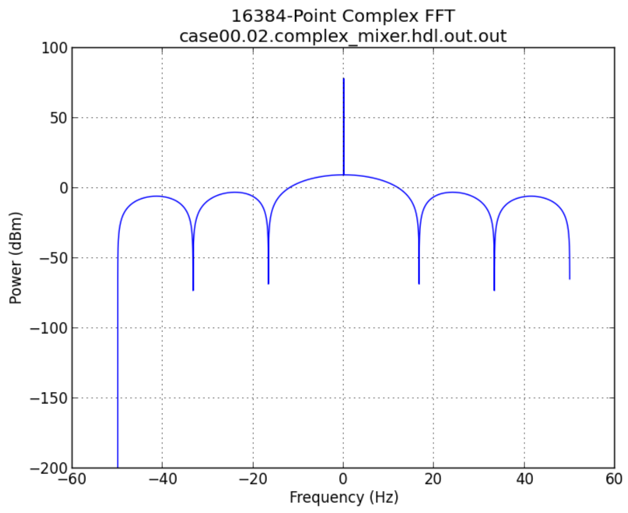




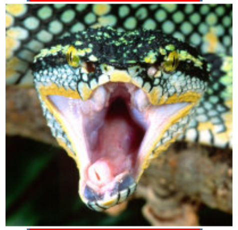
Input tone @ 12.5Hz

phs_inc = -8192

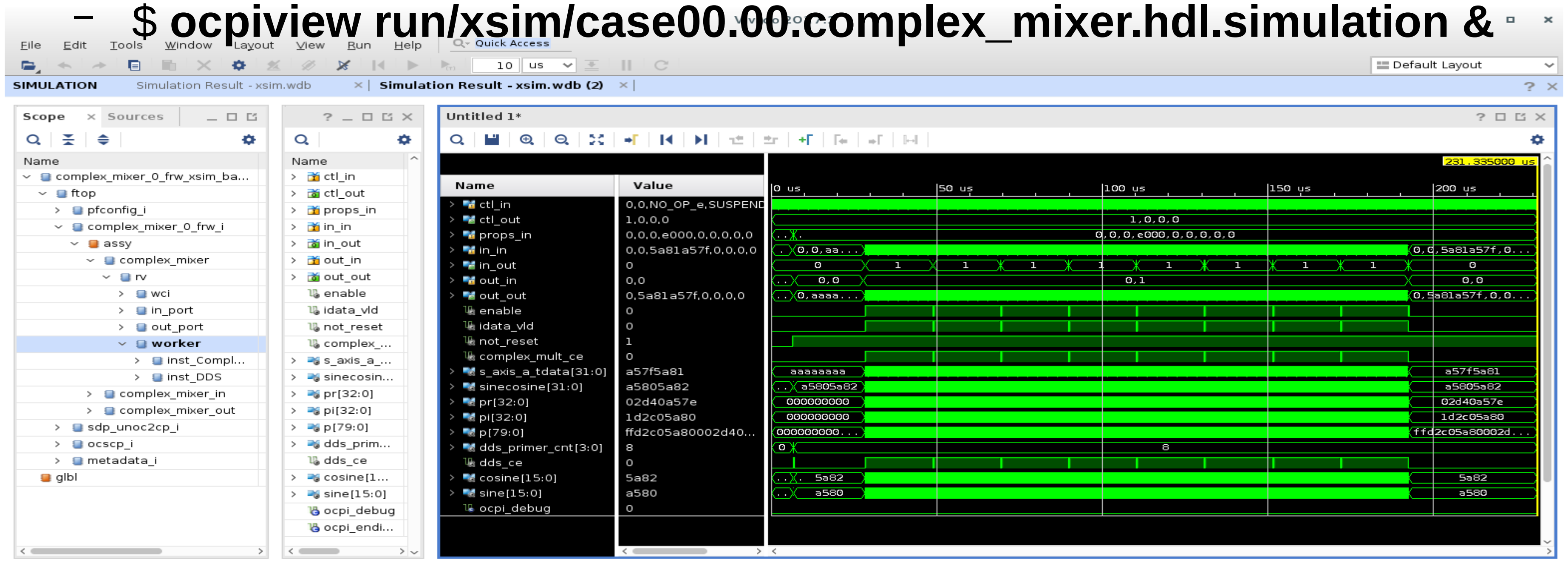
Fnco=100MHz*phs_inc/65536
Fnco=-12.5 MHz



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



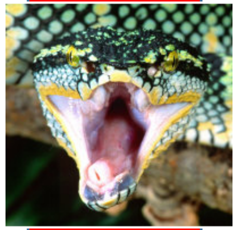
- Must have checked “keep simulations” or ran “make run OnlyPlatforms=xsim **KeepSimulations=1**”
- In a terminal window, browse to complex_mixer.test/ and execute
  - $ **ocpiview run/xsim/case00.00.complex_mixer.hdl.simulation &**


File Edit Tools Window Layout View Run Help
Quick Access
Default Layout
SIMULATION
Simulation Result - xsim.wdb
Simulation Result - xsim.wdb (2)
Scope
Sources
Untitled 1*
231.335000 us
0 us
50 us
100 us
150 us
200 us

```
complex_mixer_0_frw_xsim_ba...
  ftop
    pfconfig_i
    complex_mixer_0_frw_i
      assy
        complex_mixer
          rv
            wci
            in_port
            out_port
            worker
              inst_Compl...
              inst_DDS
        complex_mixer_in
        complex_mixer_out
    sdp_unoc2cp_i
    ocscp_i
    metadata_i
  glbl
```

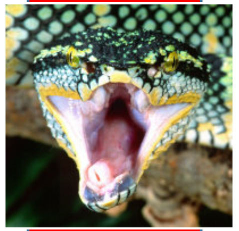
```
Name                      Value
ctl_in                    0,0,NO_OP_e,SUSPEND
ctl_out                   1,0,0,0
props_in                  0,0,0,e000,0,0,0,0,0
in_in                     0,0,5a81a57f,0,0,0,0
in_out                    0
out_in                    0,0
out_out                   0,5a81a57f,0,0,0,0
enable                    0
idata_vld                 0
not_reset                 1
complex_mult_ce           0
s_axis_a_tdata[31:0]      a57f5a81
sinecosine[31:0]          a5805a82
pr[32:0]                  02d40a57e
pi[32:0]                  1d2c05a80
p[79:0]                   ffd2c05a80002d40...
dds_primer_cnt[3:0]       8
dds_ce                    0
cosine[15:0]              5a82
sine[15:0]                a580
ocpi_debug                0
```

Step 5(b) – 7(b) - Unit Test Matchstiq-Z1



- 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 Matchstiq-Z1
  - 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 Matchstiq-Z1 (matchstiq_z1)
  - 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



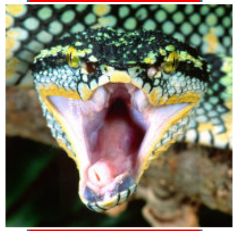
- Modify worker Makefile to reference the cores’ VHDL files
  - ***<u>PRIOR</u>*** to the include statement, ***<u>CHANGE</u>*** the **SourceFiles** and **Cores** variables to the following:

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

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

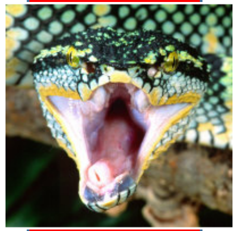
    <u>NOTE: ONLY VALID FOR TARGETING HARDWARE</u>

Backtrack to Step 4 - Build App Worker



- Build HDL App Worker for Zynq Targets
    1) Use the IDE to "**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:

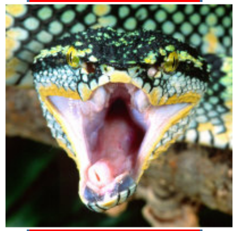
```
[ocpidev -d /data/test_training build worker complex_mixer.hdl -l components --hdl-target zynq]

make: Entering directory `/data/test_training/components/complex_mixer.hdl'
grep: vivado_ip/complex_multiplier.edf/zynq/complex_multiplier.edf.sources: Not a directory
grep: vivado_ip/dds_compiler.edf/zynq/dds_compiler.edf.sources: Not a directory
Building worker core "complex_mixer" for target "zynq" 0:(ocpi_debug=false ocpi_endian=little) target-zynq/complex_mixer.edf
 Tool "vivado" for target "zynq" succeeded.  0:33.78 at 13:04:57
Creating link to export worker binary: ../lib/hdl/zynq/complex_mixer.edf -> target-zynq/complex_mixer.edf
make: Leaving directory `/data/test_training/components/complex_mixer.hdl'
== > 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-”

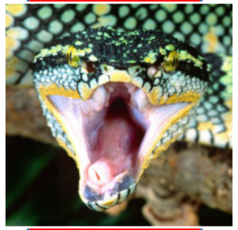
Step 5(b) - Create Unit Test

- **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" messagesize="8192"/>
 <output port="out" script="verify.py 100 16384" view="view.sh"/>
 <property name="phs_inc" values="-8192"/>
 <property name="enable" values="0,1"/>
 <property name="data_select" values="0,1"/>
<tests>
```

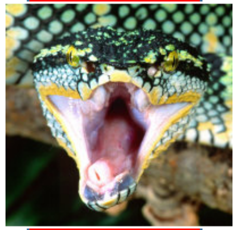


Step 6(b) - Build Unit Test (Matchstiq-Z1)



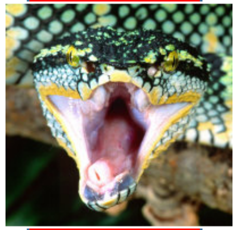
- Build Unit Test Suite for target hardware platform
  1) Use the IDE to "**Add**" the Unit Test to the Project Operations panel
  2) **Highlight** "matchstiq_z1" the HDL Platforms panel (HDL Targets box unchecked)
  3) **Check "**Tests" Radio Button
  4) **Click** "gen + build"
  5) **Check** "run view script"
  6) **Click** "prep+run+verify" to Run Tests
  7) 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 (Matchstiq-Z1)

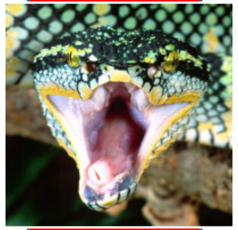


- Observe the console window to make sure the GUI completed successfully.

Step 6(b) – Review Build Artifacts (Matchstiq-Z1)



- 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**_matchstiq_z1_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

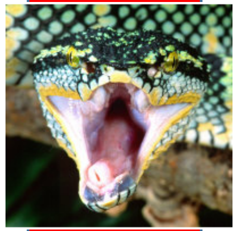


Step 7(b) – Run Unit Test (Matchstiq-Z1)

- Setup deployment platform
  1. Connect to serial port via USB on rear of Matchstiq-Z1 using host
     - ‘screen /dev/matchstiq_z1_0 115200’
  2. Boot and login into PetaLinux
     - User/Password = root:root
  3. Verify host and Matchstiq-Z1 have valid IP addresses
     - For training, they should both be on the same subnet
  4. Run setup script on Matchstiq-Z1
     - ‘source /mnt/card/opencpi/mynetsetup.sh <host ip address>’

More detail on this process can be found in the **Matchstiq-Z1 Getting Started Guide** document

Step 7(b) - Run Unit Test (Matchstiq-Z1)

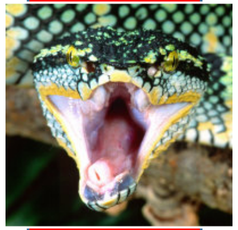


- Setup the remote test system:
  - In the Project Operations Panel
    - **Check** “Tests”
    - **Select** “remotes” radio button
    - **Click** “+remotes”
    - **Change** remote variable text to use matchstiq’s ip and point to the training project:
      {IP of Matchstiq-Z1}=root=root=/mnt/training_project
    - **Select** the newly created remote. This will be the target remote test system. Unselected remotes will not be run.

- OR Prior to launching the IDE, OCPI_REMOTE_TEST_SYSTEMS must be set

```
$ export OCPI_REMOTE_TEST_SYSTEMS=<IP of Matchstiq-Z1>=root=root=/mnt/training_project
```

Step 7(b) - Run Unit Test (Matchstiq-Z1)



- Run Unit Test Suite for target hardware platform
  1) Use the IDE to "**Add**" the Unit Test to the Project Operations panel
  2) **Highlight** "matchstiq_z1" the HDL Platforms panel (HDL Targets box unchecked)
  3) **Check** "run view script"
  4) **Click** "prep+run+verify" to Run Tests
  5) Review the Console window messages and address any errors
- Terminal operations:
  In a terminal window, execute within the {component}.test/

  $ **make run Cases=case00.01 OnlyPlatforms=matchstiq_z1 View=1**

  $ **make run Cases=case00.0* OnlyPlatforms=matchstiq_z1 View=1**

  $ **make verify** {verify previous results}

  $ **make view** {plot previous results}