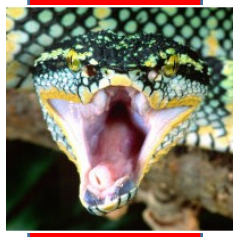
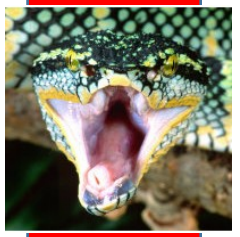


Lab 6:
HDL Application Worker
“Simple”

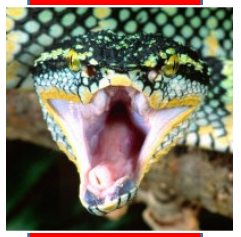
peak_detector.hdl

Objective



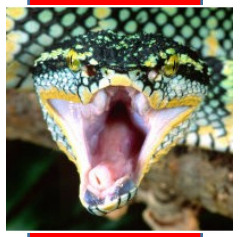
- Design, Build and Test: **HDL** Application Worker
- Function: Peak detection of a data stream (peak_detector.hdl)
- Volatile Properties: Signed Min/Max (OWD)
- Convert Data Port Interface: Interleaved to Parallel (OWD)
- Routes data/messages through the worker “pass-thru” (VHDL)
- Automated Unit Testing on multiple platforms (XML, Python)
  - Simulator: Xilinx XSIM
  - Hardware: Matchstiq-Z1 (Xilinx Zynq-7020)
- “Work-alike” to Simple RCC Worker (Lab 3)

What's Provided



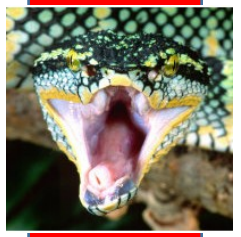
- **REUSE** "peak_detector.test/" directory
  - Same as implemented in Simple RCC Worker (Lab3) "Work-alike"
  - Recall: One {component}.test/ per OCS
- Component Datasheet
  - /home/training/provided/doc/Peak_Detector.pdf
- Worker VHDL with "commented" instructions
  - /home/training/provided/lab6/peak_detector.vhd

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

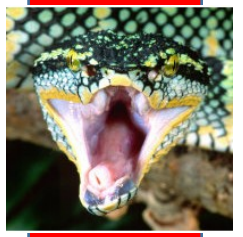
Step 1 – OPS: Use pre-existing or create new



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



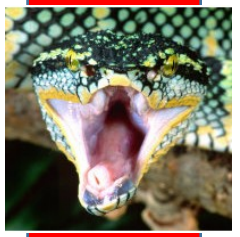
1) Review Component Spec Properties and Ports in Component Datasheet
   - Use Properties and Ports information to answer the following questions
2) Determine if OCS that satisfies the requirements exists
   1) Current project's component library?
      /home/training/training_project/components/specs/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

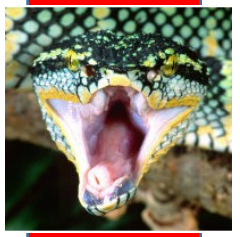
- **IDE: File → New → Other → ANGRYVIPER → OpenCPI Asset Wizard**
  - **Asset Type: Worker**
  - Worker Name: **peak_detector**
  - Add To Library: **components (default)**
  - Component: **ocpi.training.peak_detector-spec**
  - Model: **HDL**
  - Programming Language: **VHDL**
- **IDE: Refresh** the 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



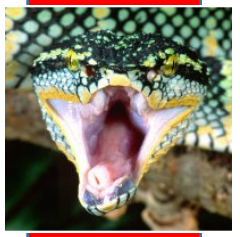
- Generated skeleton code can be built!?
  1) **IDE:** "**Refresh**" the OpenCPI Projects panel
  2) Use the IDE to "**Add**" the App Worker to the Project Operations panel
  3) **Check** the HDL Targets box and **highlight** "xsim" and "zynq", under "xilinx"
  4) **Click** "Build Assets"
  5) Review 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 – Convert Data Port to Parallel



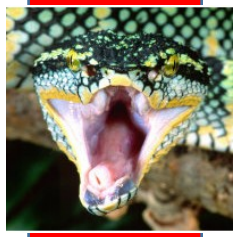
- Convert data port interface from Interleaved to Parallel
  - Note: iqstream_protocol is default interleaved 16bit I/Q sample data
- Determine port configuration settings by examining the “Worker Interfaces” table in the Component Datasheet
- Edit OWD via the IDE: OWD HDL Editor (“Design” tab)
  1) “**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 – Modify App Worker Source Code



- 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/peak_detector.hdl/**
  2) $ **cp -f /home/training/provided/lab6/peak_detector.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:
  - **Define the constant**
  - **Make general signal and data port assignments**
  - **Assign the volatile output properties**

Step 4 - Build App Worker



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

Step 4 – Review Build Logs and Artifacts

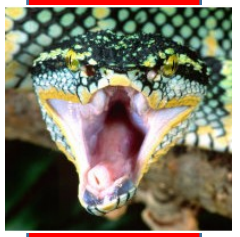
- End of build log should resemble the following, if free of errors:

```
[ocpidev -d /data/training/training_project build worker peak_detector.hdl -l components --hdl-target xsim --hdl-target zynq]

make: Entering directory `/data/training/training_project/components/peak_detector.hdl'
Generating the definition file: gen/peak_detector-defs.vhd
Generating the implementation header file: gen/peak_detector-impl.vhd from peak_detector.xml
Generating the implementation skeleton file: gen/peak_detector-skel.vhd
Generating the VHDL constants file for config 0: target-xsim/generics.vhd
Generating the opposite language definition file: gen/peak_detector-defs.vh
Generating the Verilog constants file for config 0: target-xsim/generics.vh
Building the peak_detector worker for xsim (target-xsim/peak_detector) 0:(ocpi_debug=false ocpi_endian=little)
 Tool "xsim" for target "xsim" succeeded.  0:02.48 at 14:21:04
Creating link to export worker binary: ../lib/hdl/xsim/peak_detector -> target-xsim/peak_detector
Creating link from ../lib/hdl -> peak_detector.xml to expose the peak_detector implementation xml.
Creating link from ../lib/hdl/xsim/peak_detector.vhd -> target-xsim/peak_detector-defs.vhd to expose the definition of worker peak_detector.
Creating link from ../lib/hdl/xsim/peak_detector.v -> target-xsim/peak_detector-defs.vh to expose the other-language stub for worker peak_detector.
Generating the VHDL constants file for config 0: target-zynq/generics.vhd
Generating the Verilog constants file for config 0: target-zynq/generics.vh
Building worker core "peak_detector" for target "zynq" 0:(ocpi_debug=false ocpi_endian=little) target-zynq/peak_detector.edf
 Tool "vivado" for target "zynq" succeeded.  0:33.67 at 14:21:38
Creating link to export worker binary: ../lib/hdl/zynq/peak_detector.edf -> target-zynq/peak_detector.edf
Creating link from ../lib/hdl/zynq/peak_detector.vhd -> target-zynq/peak_detector-defs.vhd to expose the definition of worker peak_detector.
Creating link from ../lib/hdl/zynq/peak_detector.v -> target-zynq/peak_detector-defs.vh to expose the other-language stub for worker peak_detector.
make: Leaving directory `/data/training/training_project/components/peak_detector.hdl'
== > Command completed. Rval = 0
```

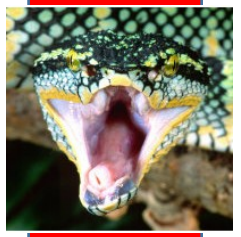
- Observe new artifacts {worker}.hdl/: “target-xsim/” and “target-zynq/”
  - These directories contain output files from their respective FPGA vendor tools (in this case, simply Xilinx Vivado) in addition to a framework auto-generated file, generics.vhd
  - ‘generics.vhd’ contains parameter configuration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

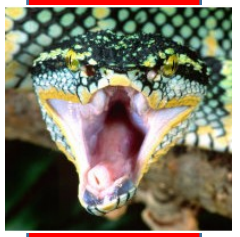


- **REUSE** from “Work-alike” Simple RCC Worker (Lab3)!
- Located in “peak_detector.test/” directory
- No changes required to Test XML

```
<tests useHDLFileIo='true'>
  <input port='in' script='generate.py 32768' messagesize='8192'/>
  <output port='out' script='verify.py 32768' view='view.sh'/>
</tests>
```

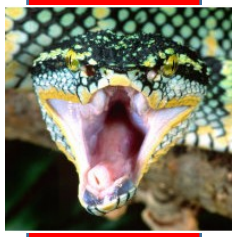
- **CRITICAL NOTE:**
  - **IDE does not currently support creation of unit test directory**
  - **Use “ocpidev create test <OCS>” to create the {OCS}.test directory**

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



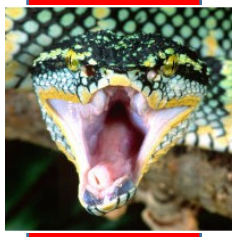
- Build Unit Test Suite for target simulation platform
  1) Use the IDE to "**Add**" the Unit Test to the Project Operations panel
  2) **Highlight** "xsim" the HDL Platforms panel (HDL Targets box unchecked)
  3) **Click** "Build Tests"
  4) Review the Console window messages and address any errors
- Observe new artifacts in peak_detector.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 peak_detector.test/gen/assemblies/peak_detector_0_frw
  - **peak_detector_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-**peak_detector_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 launching 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) **Click** "Run Tests"
  4) 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/

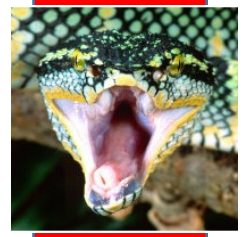
  $ **make run** {run on all available platforms, no plotting, discard resulting simulation}

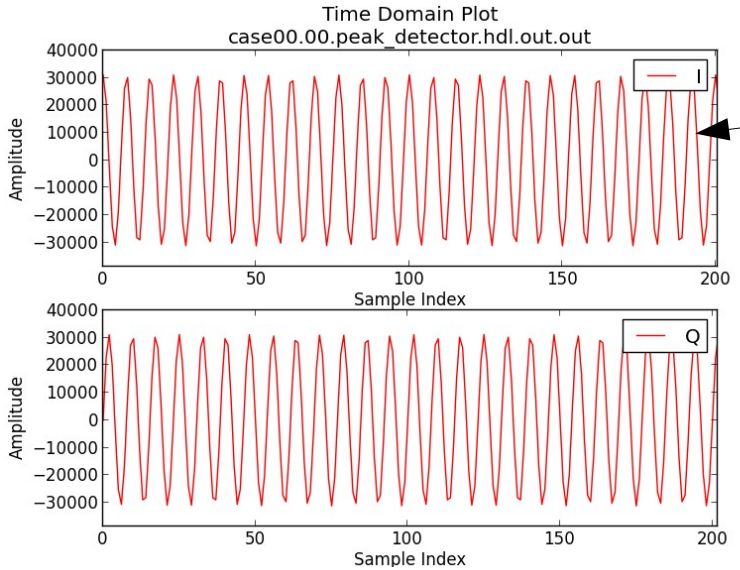
  $ **make run OnlyPlatforms=xsim KeepSimulations=1 View=1** (PLOTS ON NEXT PAGE)

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

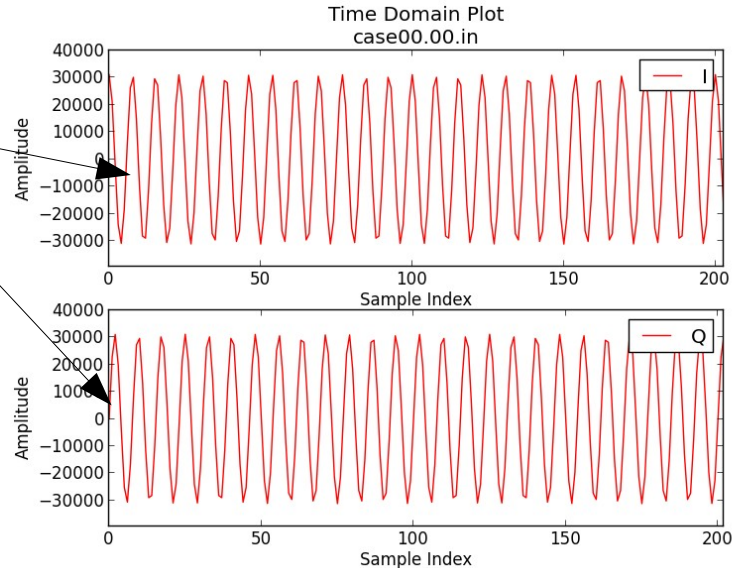
  $ **make view** {plot previous results}

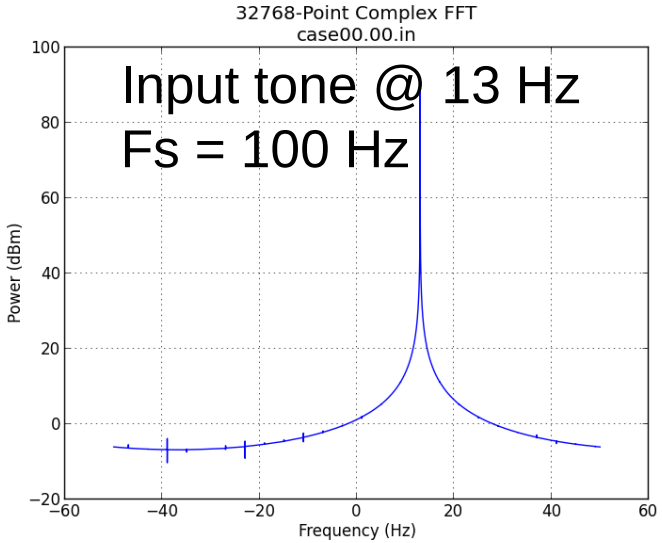
Step 7(a) - Unit Test I/O Plots (Xilinx XSIM)



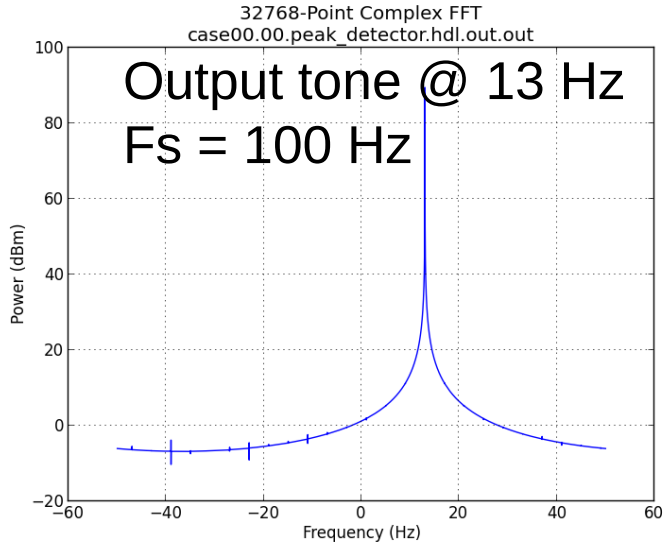


Note:
Zoomed-In



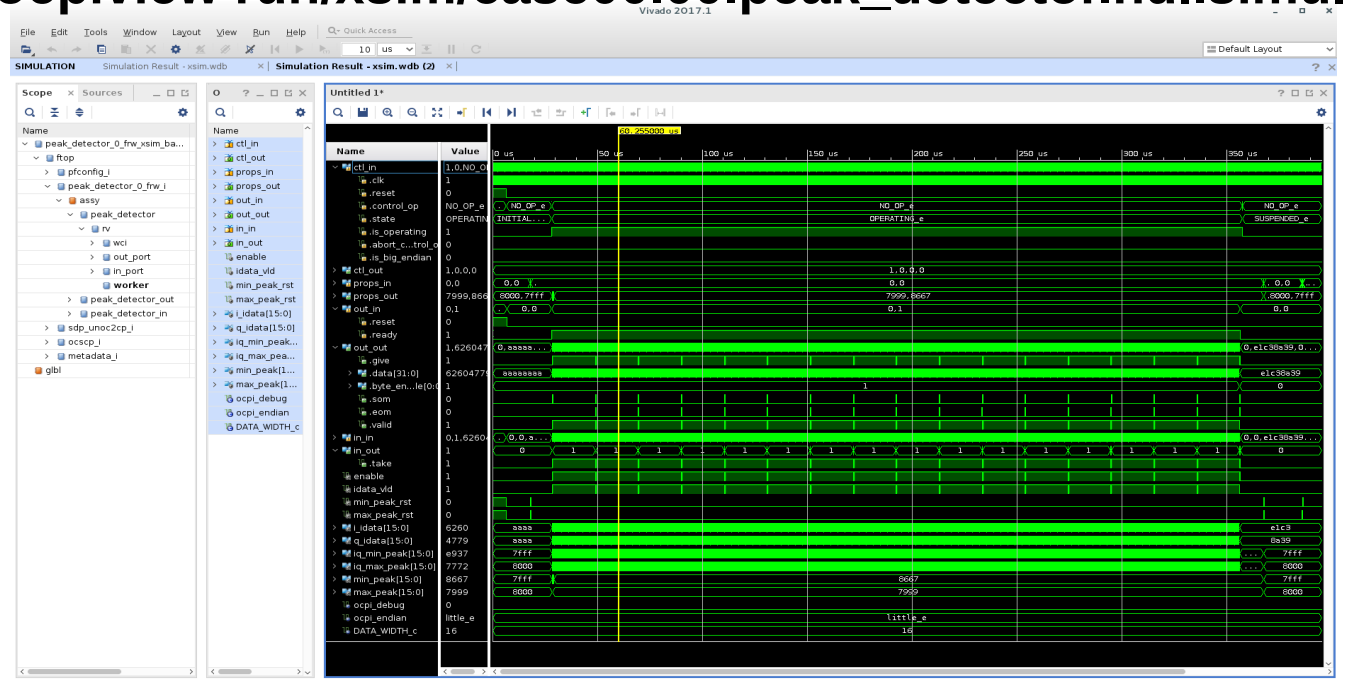


min_peak = -31129
max_peak = 31129

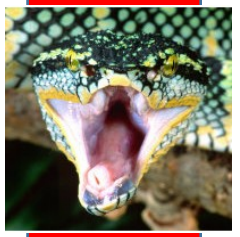


Step 7(a) – View Simulation Waveforms (Xilinx XSIM)

- Must have ran "make run OnlyPlatforms=xsim **KeepSimulations=1**"
- In a terminal window, browse to peak_detector.test/ and execute
  - $ **ocpiview run/xsim/case00.00.peak_detector.hdl.simulation &**

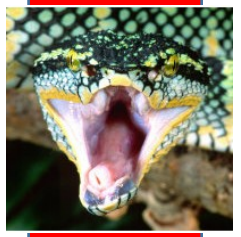


Step 5(b) – 7(b) - Unit Test Matchstiq-Z1



- 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)

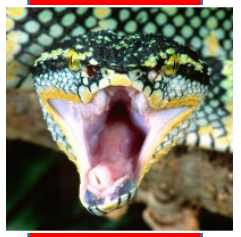
Step 5(b) - Create Unit Test



- **REUSE** from 'Work-alike' Simple RCC Worker (Lab3)!
- **REUSE** from Simulation portion of this lab!
- Located in “peak_detector.test/” directory
- No changes required Test XML

- CRITICAL NOTE:
  - IDE does not currently support creation of unit test directory
  - Use “ocpidev create test <OCS>” to create the {OCS}.test directory

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) **Click** "Build Tests"
  4) 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)

- Below is an example of a successful build
  - (only the end portion is shown)

```
Building container core "peak_detector_0_matchstiq_z1_base" for target "zynq" 0:(sdp_width=1 ocpi_debug=false ocpi_endian=little) target-zynq/peak_detector_0_matchstiq_z1_base.edf
Generating UUID, artifact xml file and metadata ROM file for container peak_detector_0_matchstiq_z1_base (0).
Wrote bram file 'target-zynq/metadatarom.dat' (1876 bytes) from file 'target-zynq/peak_detector_0_matchstiq_z1_base-art.xml' (12690 bytes)
 Tool "vivado" for target "zynq" succeeded.  0:45.34 at 12:58:31
Creating link to export worker binary: ../../peak_detector_0/lib/hdl/zynq/peak_detector_0_matchstiq_z1_base.edf -> target-zynq/peak_detector_0_matchstiq_z1_base.edf
Creating link from ../../peak_detector_0/lib/hdl -> gen/peak_detector_0_matchstiq_z1_base.xml to expose the container-peak_detector_0_matchstiq_z1_base implementation xml.
Creating link from ../../peak_detector_0/lib/hdl/zynq/peak_detector_0_matchstiq_z1_base.vhd -> target-zynq/peak_detector_0_matchstiq_z1_base-defs.vhd to expose the definition of worker peak_detector_0_matchstiq_z1_base.
Creating link from ../../peak_detector_0/lib/hdl/zynq/peak_detector_0_matchstiq_z1_base.v -> target-zynq/peak_detector_0_matchstiq_z1_base-defs.vh to expose the other-language stub for worker peak_detector_0_matchstiq_z1_base.
For peak_detector_0 on matchstiq_z1 using config base: creating optimized DCP file using "opt_design". Details in opt.out
Time: 0:38.66 at 12:59:10
 Tool "vivado" for target "zynq" succeeded on stage "opt".
For peak_detector_0 on matchstiq_z1 using config base: creating placed DCP file using "place_design". Details in place.out
Time: 0:54.32 at 13:00:04
 Tool "vivado" for target "zynq" succeeded on stage "place".
For peak_detector_0 on matchstiq_z1 using config base: creating routed DCP file using "route_design". Details in route.out
Time: 1:03.33 at 13:01:08
 Tool "vivado" for target "zynq" succeeded on stage "route".
Generating timing report (RPX) for peak_detector_0 on matchstiq_z1 using base using "report_timing". Details in timing.out
Time: 0:28.55 at 13:01:36
 Tool "vivado" for target "zynq" succeeded on stage "timing".
For peak_detector_0 on matchstiq_z1 using config base: Generating Xilinx Vivado bitstream file target-zynq/peak_detector_0_matchstiq_z1_base.bit. Details in bit.out
Time: 0:41.37 at 13:02:18
 Tool "vivado" for target "zynq" succeeded on stage "bit".
Making compressed bit file: target-zynq/peak_detector_0_matchstiq_z1_base.bit.gz from target-zynq/peak_detector_0_matchstiq_z1_base.bit and target-zynq/peak_detector_0_matchstiq_z1_base-art.xml
```

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

- Observe new artifacts in peak_detector.test/gen/assemblies/**peak_detector_0**/
  - **peak_detector_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-**peak_detector_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/ttyUSB0 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)

- Prior to launching the IDE, OCPI_REMOTE_TEST_SYSTEMS must be set

  $ **export OCPI_REMOTE_TEST_SYSTEMS=<IP of Matchsitq-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) **Click** "Run Tests"
  4) Review the Console window messages and address any errors
- Other operations not currently supported by IDE.
  In a terminal window, executed within the {component}.test/

  $ **make run** {run on all available platforms, no plotting, discard resulting simulation}

  $ **make run OnlyPlatforms=matchstiq_z1 View=1**

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

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

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

- Python script verifies output data from the Unit Test

```
*** Python: Peak Detector ***
*** Validate output against expected data ***
File to validate:  case00.00.peak_detector.hdl.out.out
uut_min_peak =  -31129
uut_max_peak =  31129
file_min_peak =  -31129
file_max_peak =  31129
Data matched expected results.
PASSED
*** End validation ***
```