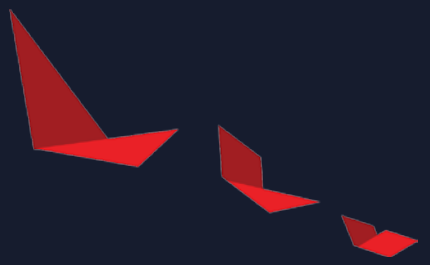


Debugging

Debugging Flows

- Vitis provides application-level debug
  - Host code
  - Kernel code
  - Interactions between them
- Three levels of abstraction for debug
  - Software Emulation
  - Hardware Emulation
  - System

Software Emulation	Hardware Emulation	Hardware Run
Host application runs with a C/C++ model of the kernels	Host application runs with a simulated RTL model of the kernels	Host application runs with actual FPGA implementation of the kernels
Confirm functional correctness of the system	Test host/kernel integration; obtain performance estimates	Confirm system runs correctly and with desired performance
Fastest turnaround time	Best debug capabilities	Accurate performance results

Features and Techniques for Debug

- GDB
- Print from code (“printf”)
- Waveform viewer
- Vivado analyzer
- dmesg
- xbutil
- xclbininfo

Debugging

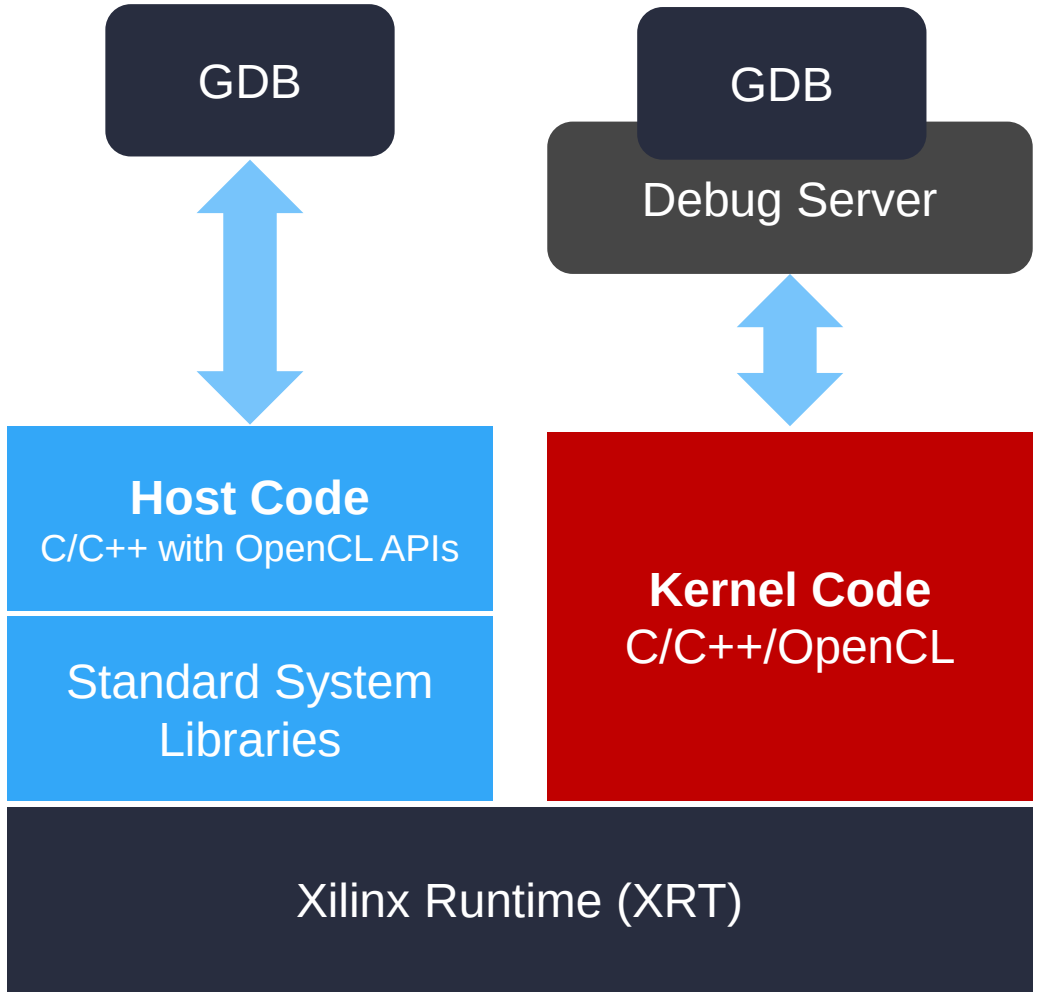
Debugger	Source	Software Emulation	Hardware Emulation	System Run
GNU debugging (GDB) tool	Host code	✓	✓	✓
	Kernel code	✓	Limited support	✗

- Waveform-based kernel debugging
  - Hardware designers use this approach
- Using `printf()` or `cout` to debug kernels
  - C/C++-based kernels
    - `printf()` is only supported during software emulation
    - Should be excluded from the Vivado® HLS tool synthesis
  - OpenCL kernels
    - Supports `printf()` for all build configurations: software emulation, hardware emulation, and system run

```
#ifndef __SYNTHESIS__
    printf("Checkpoint 1 reached");
#endif
```

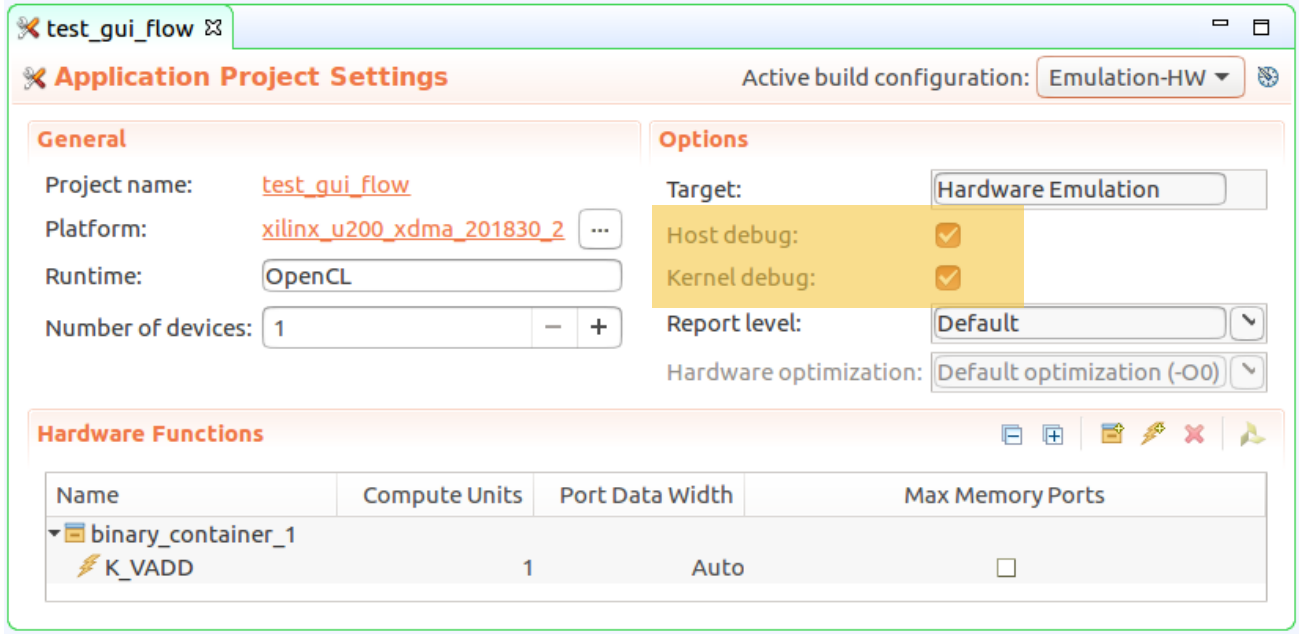
Debugging in Software Emulation

- GDB
  - GUI (IDE)
  - Command line
- Code that can be debugged
  - Host code
    - Software emulation
    - Hardware emulation
    - System flow
  - Kernel code
    - Software emulation
    - Hardware emulation (limited)
- Xilinx recommends iterating the design as much as possible in software emulation, which takes little compile time and executes quickly

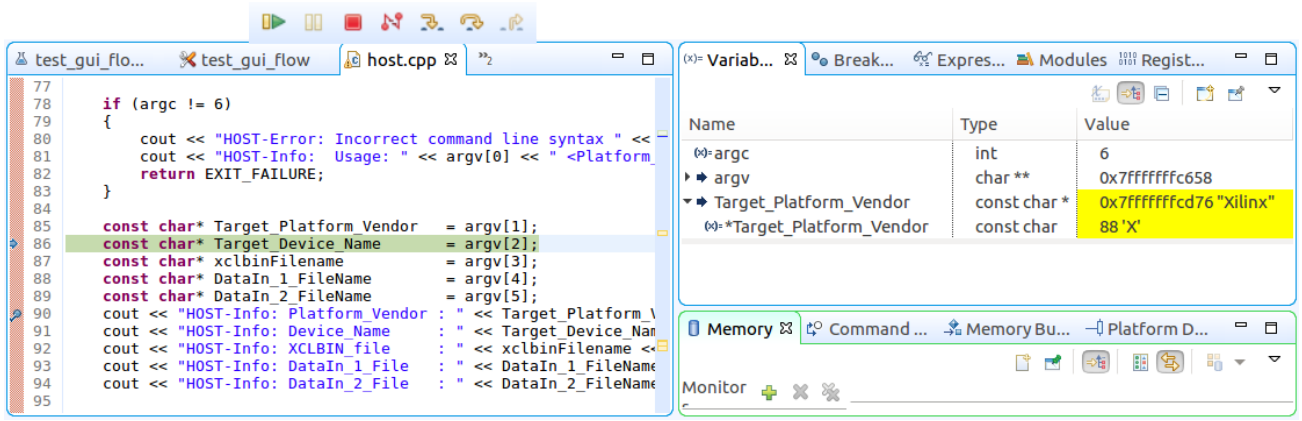


GDB-based Debugging - GUI

- GNU debugging (GDB)
  - Enable
    - Host debug
    - Kernel debug
  - Add breakpoints
  - Inspect variables, registers, memory



- Vitis debugger provides extension to GDB
  - Allows examining the contents of the XRT



GDB-based Debugging – Command Line

- Host code – Enable debug features during compilation and linking

Enable host code debug	`g++ -g ..`

- Kernel code – Enable debug features during compilation and linking

Enable kernel code debug	`v++ -g ..`

- Launching host and kernel debug
  - Run GDB separately for host code and kernel code debug

Host and Kernel Code Debug – Command Line

Terminal 1 – xrt_server

1. Set up the Vitis environment
2. Start the Vitis debug server:

```
xrt_server --sdx-url
```

Terminal 2 – Host Code Debug

1. Set up the Vitis environment
2. Set the emulation mode

```
export XCL_EMULATION_MODE = <sw_emu | hw_emu>
```

3. Enable the debug feature

```
[Debug]
app_debug=true
```

4. Start gdb

```
xgdb –args <host> <xclbin>
```

Terminal 3 – Kernel Code Debug

1. Set up the Vitis environment
2. Launch xgdb and run the command given below

Software Emulation

```
file <Vitis_path>/data/emulation/unified/cpu_em/generic_pcie/model/genericpciemodel
```

Hardware Emulation

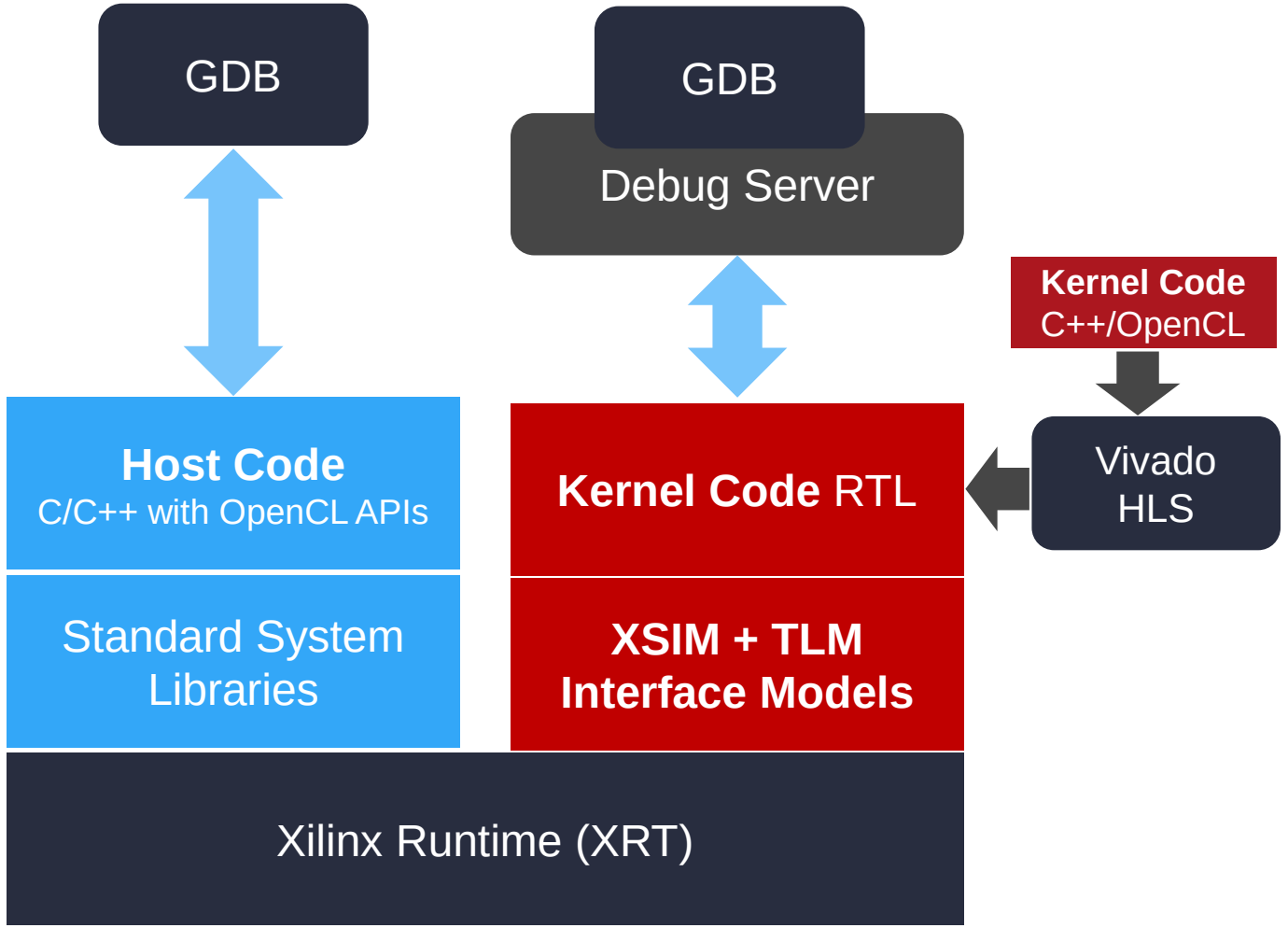
```
file /tmp/sdx/$uid/$pid/NUM.DWARF
```

Connect to the kernel process

```
Target remote :<num>
```

Debugging in Hardware Emulation

- In hardware emulation
  - Kernel code – C/++/OpenCL/RTL
    - Translated to RTL
    - Validate the RTL logic of kernels
  - Host code
    - Executed concurrently with a behavioral simulation of the RTL model of the kernel
- Execution time for hardware emulation is longer than software emulation



Waveform-based Kernel Debugging

- Enable debug code generation during compilation and linking using the command

```
v++ -c -g -t hw_emu ...
```

- Enable the profiling, data gathering option in the xrt.ini file

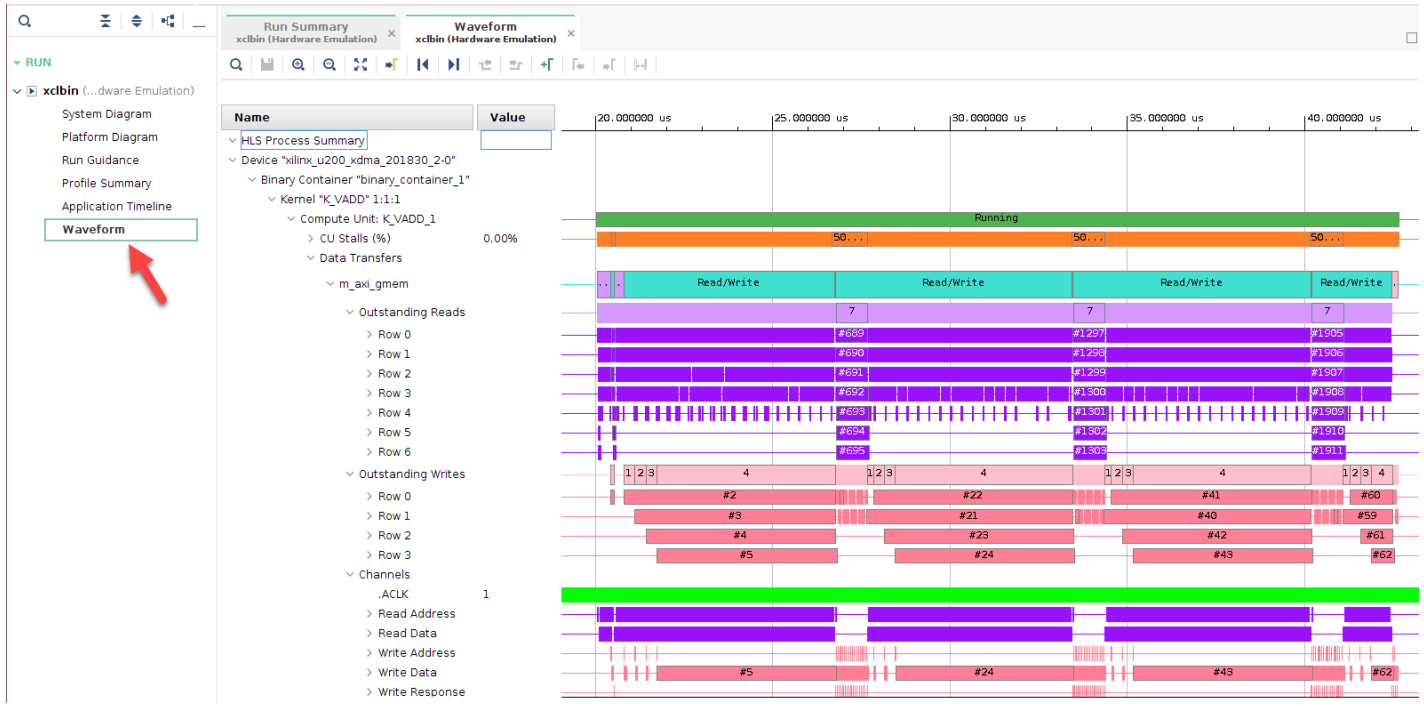
```
#Start of Debug group
[Debug]
profile=true
timeline_trace=true
#Start of Emulation group
launch_waveform=batch
```

- For the Live Waveform viewer

```
#Start of Emulation group
launch_waveform=gui
```

- Open Waveform view in Vitis analyzer

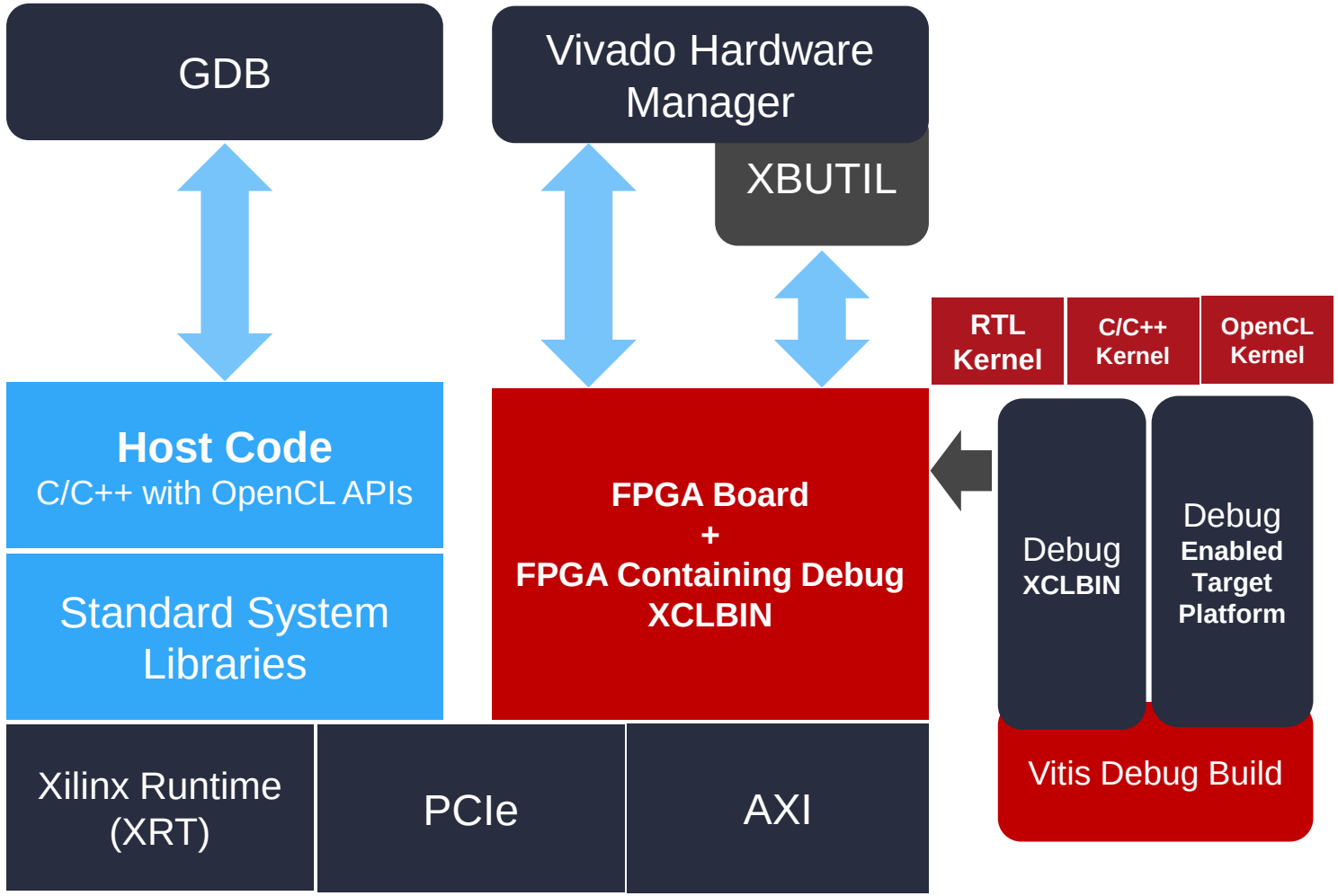
```
vitis_analyzer <output>.run_summary
```



XILINX.

Debugging During Hardware Execution

- In Hardware execution
  - Kernels execute on the actual hardware platform
  - Evaluate the performance of the host program and accelerated kernels
- Debugging the hardware build requires
  - Additional logics such as ILA and VIO
  - Impact both the FPGA resources and the performance pf the kernels



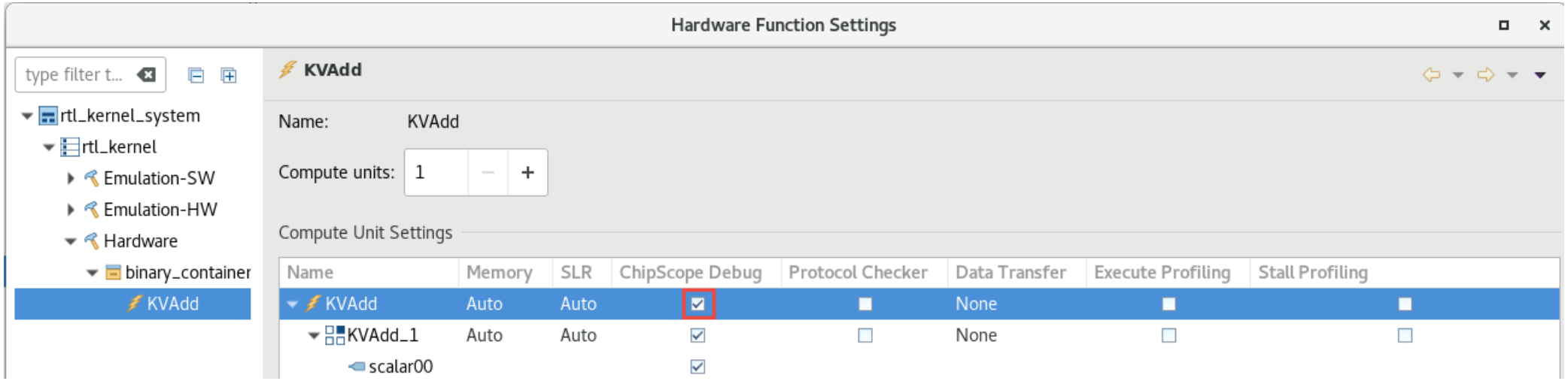
XILINX.

Debugging with ChipScope Bridge and Cores

- Vitis supports hardware debugging using ChipScope cores
  - Bridge
  - System ILA
  - ILA
  - VIO
  - Performance Monitor
- Vitis also supports protocol checker option
  - Additional hardware is inserted which monitors various AXI protocols
- Adding cores will increase resource utilization

Enabling ChipScope Debug and Protocol Checker

- In Hardware target compilation, select kernel
- Under Compute Unit settings, click check boxes of ChipScope Debug and Protocol Checker for each kernel you want to debug
  - Compiler will instantiate appropriate cores, depending on data ports type, at the kernel interface level which then connects to Debug bridge
  - Vivado Hardware Manager will communicate with Debug bridge



XILINX.

Debugging RTL Kernel on AWS

- The Custom Logic (CL) is required to include the CL Debug Bridge provided by AWS as part of the HDK
- Add any required standard Xilinx debug IP components like ILAs and VIOs
- The CL Debug Bridge must be present in the design. If the CL debug bridge is not detected, Vivado will automatically insert one into the CL design

CL Bridge Instantiation

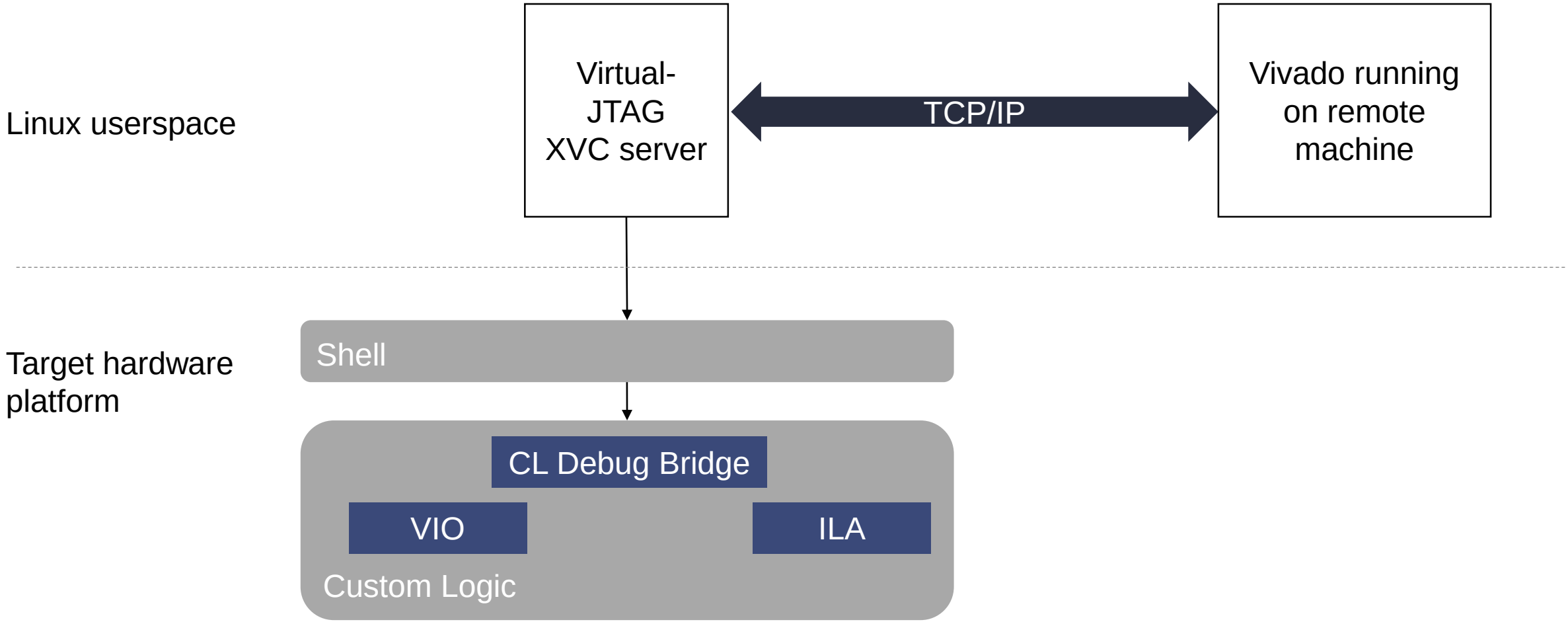
- The nets connecting to the CL Debug Bridge must have the same names as the port names of the CL Debug Bridge, except the clock
- The clock to the CL Debug Bridge should be one of the various input CL clocks (clk_main_a0 and all the clk_xtra_*)
- When the net names are correct, these nets will connect automatically to the top level of the CL

```
cl_debug_bridge CL_DEBUG_BRIDGE (
    .clk(clk_main_a0),
    .drck(drck),
    .shift(shift),
    .tdi(tdi),
    .update(update),
    .sel(sel),
    .tdo(tdo),
    .tms(tms),
    .tck(tck),
    .runtest(runtest),
    .reset(reset),
    .capture(capture),
    .bscanid(bscanid)
);
```

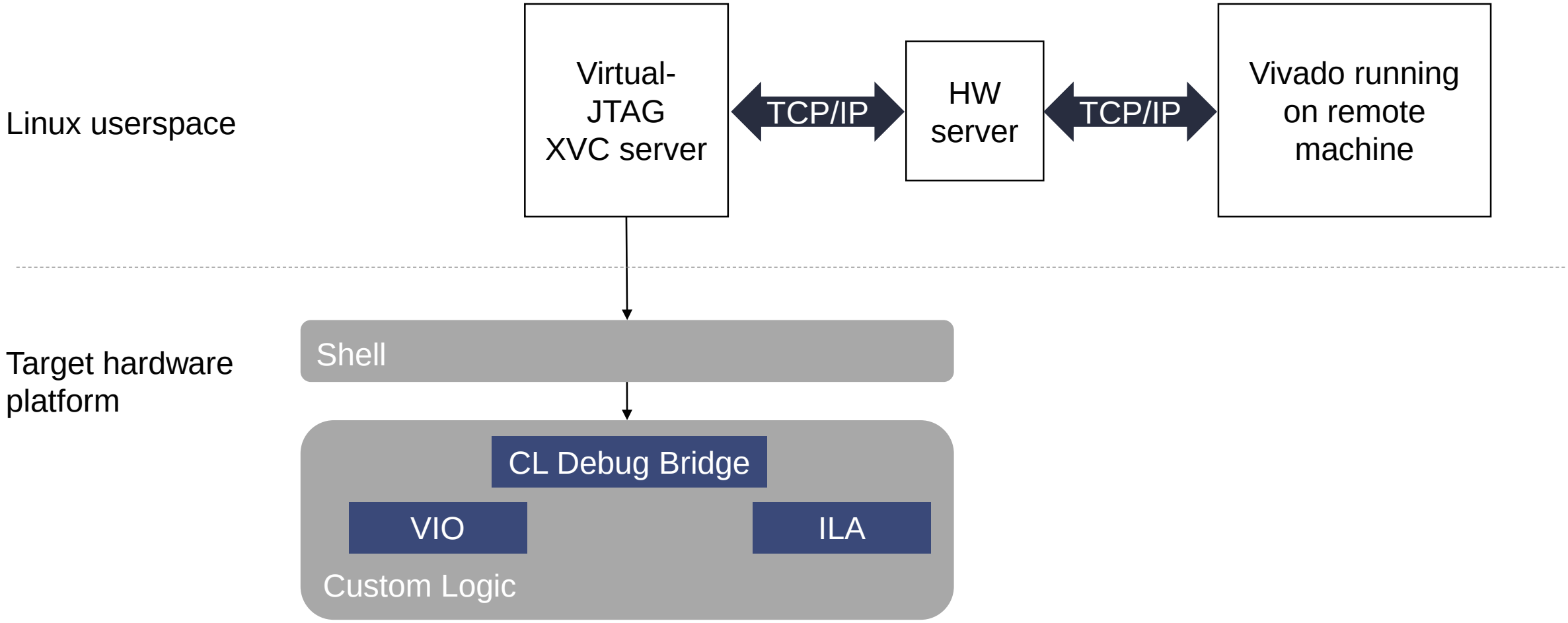
Preparing RTL IP for Debugging

- Instantiate Xilinx' Integrated Logic Analyzer (ILA) if desired
  - An ILA IP should be created using Vivado IP Catalog and it should be customized according to the desired probes
    - The ILA can be instanced at any level in the hierarchy inside the CL and the nets requiring debug have to be connected with the probe input ports of the ILA
    - The clock to the ILA should be the same clock of the clock domain to which the nets under debug belong to
- Instantiate Xilinx' Virtual Input/Output (VIO)
  - A VIO IP should be created using Vivado IP Catalog and it should be customized as needed
    - The VIO can be instanced at any level in the hierarchy inside the CL and the input/output nets should be connected as desired
    - The clock to the VIO should be the same clock of the clock domain to which the VIO output/input probe signals belong to
- Set `set_param chipscope.enablePRFlow true` in the tcl command during synthesis and implementation

Debugging Locally on AWS F1



Debugging Locally on AWS F1



Debugging Steps

- Enable ChipScope bridge and then generate bitstream
- Modify host code to pause after FPGA is programmed if using Run mode, otherwise use Debug mode and set a breakpoint after the FPGA is programmed
- Launch Xilinx Virtual Cable service
- Launch Vivado, open hardware manager, and make hardware connection
- Load debug nets info (load *.ltx file)
- Setup trigger conditions, arm ILA cores, continue application execution
- Analyze signals in waveform window after the desired trigger condition is met

Thank You