Debugging

SDx 2018.2

XILINX.

Objectives

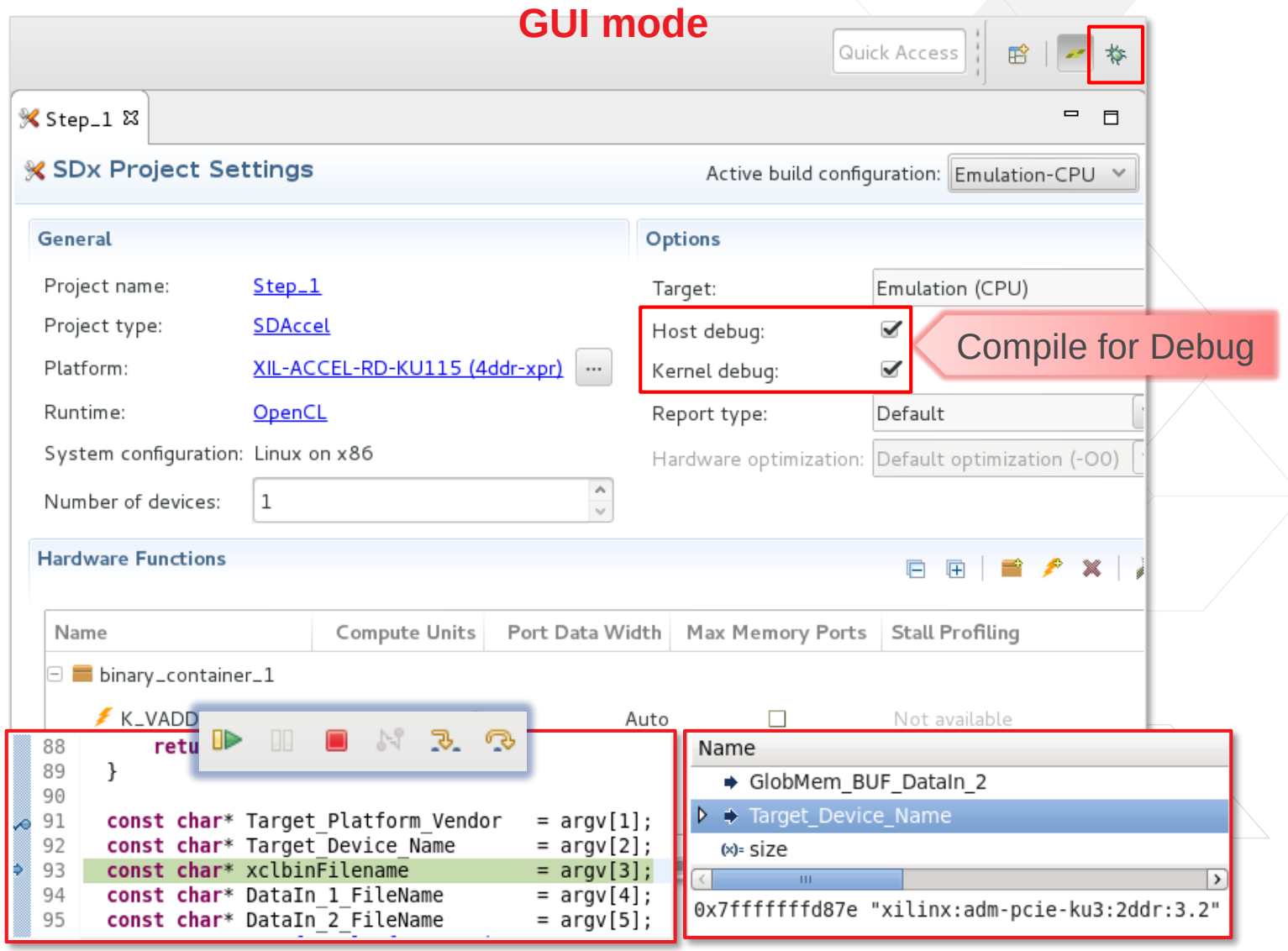
- **After completing this module, you will be able to:**
  - Describe the three levels of debugging, including software emulation, hardware emulation, and hardware execution
  - Describe Chipscope-based hardware debugging in hardware execution

Outline

- Software debugging
- Hardware debugging

SDAccel Software Debugging Environment

- **GDB support**
  - GUI (IDE)
  - Command Line
- **Code that can be debugged**
  - **Host**: SW Emulation
    HW Emulation
    System
  - **Kernel**: SW Emulation,
    HW Emulation



XILINX.

printf – Kernel Debugging

- **Supports: SW/HW Emulation, System**
- **Very valuable in**
  - HW Emulation (limited GDB support)
  - System Run (GDB not supported)
- **Only supported by OpenCL kernels**
  - Not by C/C++ kernels

Example

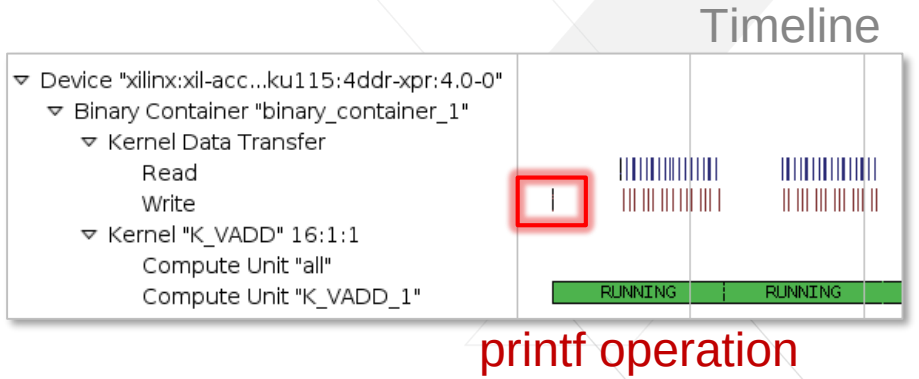
```
...
__kernel
void K_VADD(__global int* A, __global int* B, __global int* R)
{
  #ifdef KERNEL_DEBUG
      printf("\n__Kernel-Debug__: Number of Vector Elements: %d\n\n", N);
  #endif
   ...
}
```

```
HOST-Info: ================================================================
HOST-Info: (Step 5) Set Kernel Arguments and Execute Kernel
HOST-Info: ================================================================
HOST-Info: Setting Kernel arguments ...
HOST-Info: Executing Kernel ...

__Kernel-Debug__: Number of Vector Elements: 256

HOST-Info: Kernel Execution Completed
```

printf – Kernel Debugging (2)



- **How it works:**
  - System allocates 2048 Bytes in Global memory for messages
  - Messages
    - Buffered in the Global Memory
    - Unloaded when kernel execution is completed
- **printf used in multiple kernels – message order not guaranteed**
  - Tip: print for example global id to see where the message comes from

```
__kernel void hello_world(__global int *a)
{
    int idx = get_global_id(0);
    printf("Hello world from work item %d\n", idx);
    …
}
```

```
Hello world from work item 0
Hello world from work item 1
Hello world from work item 2
Hello world from work item 3
```

- **Be aware that printf might impact design performance**

XILINX.

printf – Use cases

Global Size = 1
Work Group Size = 1

> **Vector Addition: Case 1**

```
#define N 128

__kernel __attribute__ ((reqd_work_group_size(1,1,1)))
void K_VADD(__global int* A, … int* B, … int* R)
{
  printf("Number of Vector Elements: %d\n", N);

  for (int i=0; i<MAX_Nb_Of_Elements; i++)
    R[i] = A[i]+B[i];
}
```

Top Memory Writes: Host and Device Global Memory

Buffer Address	Context ID	Command Queue ID	Start Time (ms)	Duration (ms)	Buffer Size (KB)	Writing Rate (MB/s)
0x0	0	0	460.576	N/A	3.072	N/A

Top Memory Reads: Host and Device Global Memory

Buffer Address	Context ID	Command Queue ID	Start Time (ms)	Duration (ms)	Buffer Size (KB)	Reading Rate (MB/s)
0x3000	0	0	1150.980	N/A	2.048	N/A
0x2000	0	0	1162.060	N/A	0.512	N/A

```
3.072: 2.048 - printf
     : 1.024 - data
```

printf operations

Compute Unit Utilization (includes estimated device times)

Device	Compute Unit	Kernel	Global Work Size	Local Work Size	Number Of Calls	Total Time (ms)
xilinx_kcu1500_dynamic_...	K_VADD_1	K_VA...	1:1:1	1:1:1	1	0.003

CU run time

XILINX.

printf – Use cases (2)

- **Vector Addition: Case 2**
  - 2.048 KB allocated for each Kernel run
  - Case 1: Buffer size 2.048 KB

Global Size = 128
Work Group Size = 16

```
#define N 128

__kernel __attribute__ ((reqd_work_group_size(16,1,1)))
void K_VADD(__global int* A, … int* B, … int* R)
{
  int i = get_global_id(0);

  printf("Number of Vector Elements: %d\n", N);

  R[i] = A[i]+B[i];
}
```

Top Memory Writes: Host and Device Global Memory

Buffer Address	Context ID	Command Queue ID	Start Time (ms)	Duration (ms)	Buffer Size (KB)	Writing Rate (MB/s)
0x0	0	0	1371.750	N/A	263.168	N/A

Top Memory Reads: Host and Device Global Memory

Buffer Address	Context ID	Command Queue ID	Start Time (ms)	Duration (ms)	Buffer Size (KB)	Reading Rate (MB/s)
0x3000	0	0	7899.300	N/A	262.144	N/A
0x2000	0	0	7946.950	N/A	0.512	N/A

```
3.072: 262.144 - printf
     :   1.024 - data
```

printf operations

Compute Unit Utilization (includes estimated device times)

Device	Compute Unit	Kernel	Global Work Size	Local Work Size	Number Of Calls	Total Time (ms)
xilinx_kcu1500_dynamic_5…	K_VADD_1	K_VA…	128:1:1	16:1:1	8	0.028

CU run time

XILINX.

printf – Use cases (3)

Global Size = 128
Work Group Size = 16

```
#define N 128

__kernel __attribute__ ((reqd_work_group_size(16,1,1)))
void K_VADD(__global int* A, … int* B, … int* R)
{
  int i = get_global_id(0);

  // printf("Number of Vector Elements: %d\n", N);

  R[i] = A[i]+B[i];
}
```

- **Vector Addition: Case 3**
  - Case 2:
    - Buffer size 262.144 KB
    - CU Run time: 0.028 ms

Top Memory Writes: Host and Device Global Memory

Buffer Address	Context ID	Command Queue ID	Start Time (ms)	Duration (ms)	Buffer Size (KB)	Writing Rate (MB/s)
0x0	0	0	1225.320	N/A	1.024	N/A

Top Memory Reads: Host and Device Global Memory

Buffer Address	Context ID	Command Queue ID	Start Time (ms)	Duration (ms)	Buffer Size (KB)	Reading Rate (MB/s)
0x2000	0	0	7046.210	N/A	0.512	N/A

No printf operations

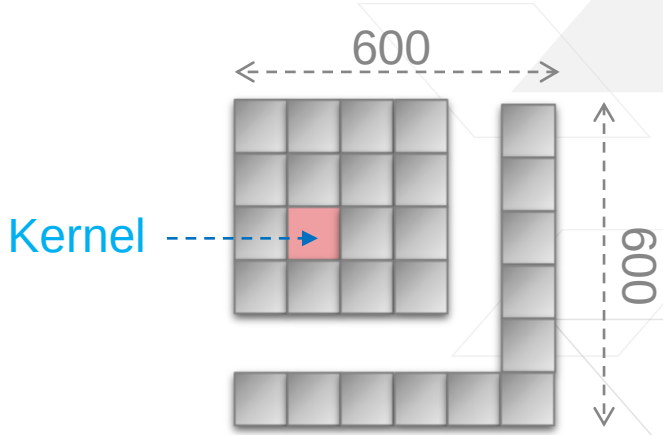
Compute Unit Utilization (includes estimated device times)

Device	Compute Unit	Kernel	Global Work Size	Local Work Size	Number Of Calls	Total Time (ms)
xilinx_kcu1500_dynamic_5…	K_VADD_1	K_VA…	128:1:1	16:1:1	8	0.026

CU run time:
~7% faster

XILINX.

printf – Use cases (4)



- **Real Design Case – Image processing**
  - Image: 600x600 pixels
  - Single Kernel with printf
    - Processes: 1 pixel
    - Memory consumed by printf: `2048 x (600 x 600)  =  737,280,000`
  - Design - 6 different kernels with printf
    - Memory consumed by printf: `2048 x (600 x 600) x 6 = 4,423,680,000`

- **Tip: Comment out printf commands when optimizing your application**

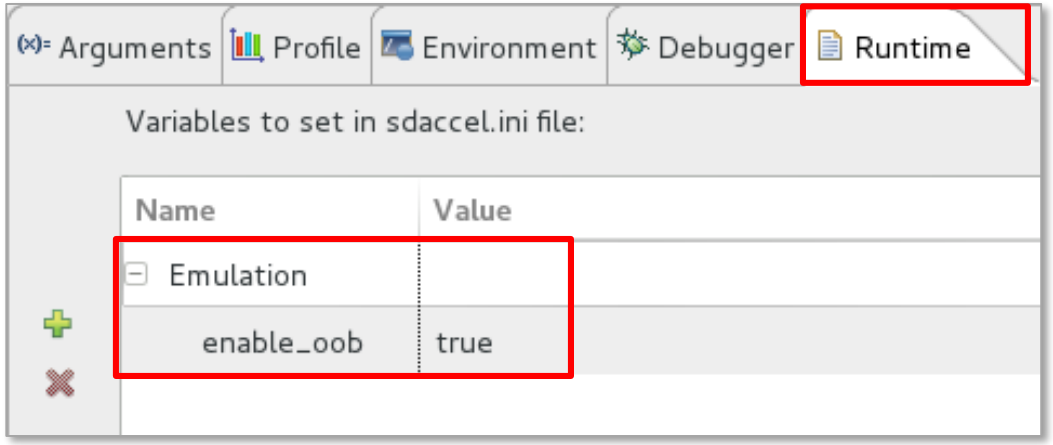
Enhanced Debug Checks in HW Emulation

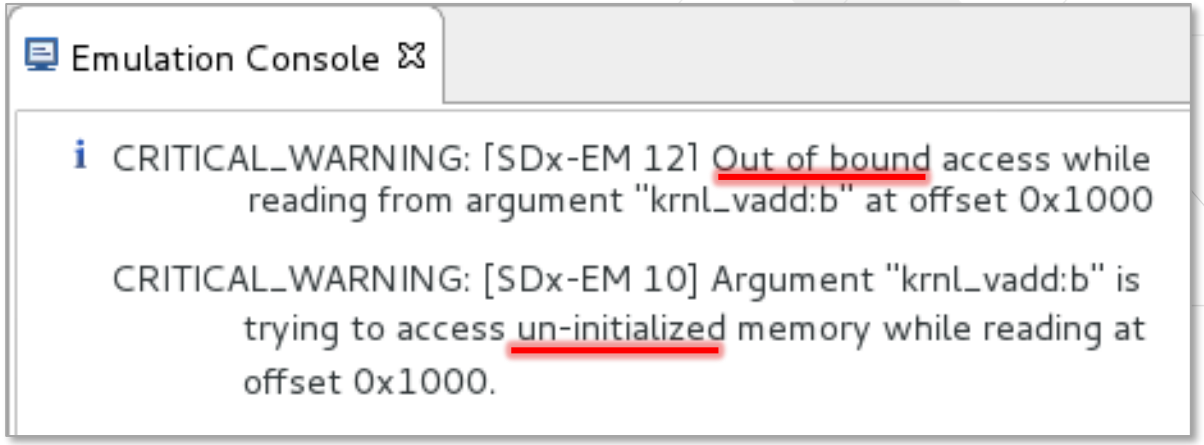
- **Checks**
  - **Un-initialized** memory read by kernel
  - **Out of Bounds** array access
  - Use `enable_oob = true`
  - Control via `sdaccel.ini` or GUI

```
// Vector Addition
__kernel
void krnl_vadd(__global int* a, … * b, … *c, …) {
  for(int i = 0; i < length; i++)
    c[i] = a[i] + b[i+1];
}
```

Error: Result mismatch

Run Configurations …



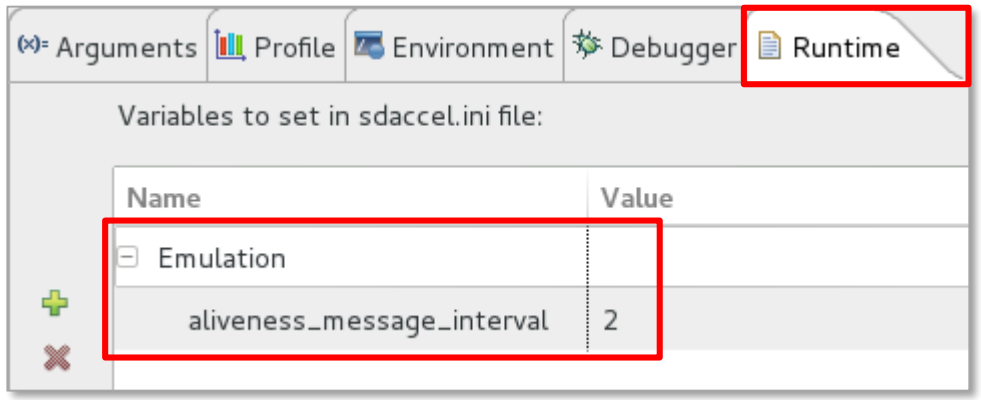


XILINX.

Enhanced Debug Checks in HW Emulation (2)

- **Periodic aliveness status during long HW Emulation runs**
  - Can be controlled via `aliveness_message_interval = N` (sec)
  - Control via `sdaccel.ini` or GUI

Run Configurations …



Console

```
INFO: [SDx-EM 22] [Wall clock time: 18:31, Emulation time: 0.0267003 ms]
                  Data transfer between kernel(s) and global memory(s)
INFO: [SDx-EM 22] [Wall clock time: 18:33, Emulation time: 0.0287005 ms]
                  Data transfer between kernel(s) and global memory(s)
```


XILINX.

Xilinx GDB Extensions

- New GDB commands: two groups
  - Group 1: Visibility into OpenCL run-time data structures
    - **cl_command_queue**, **cl_event**, **cl_mem**
  - Group 2: Visibility into the IPs on the platform (system run only)

```
xprint queue [<cl_command_queue>]
xprint event <cl_event>
xprint mem [<cl_mem>]
xprint kernel
xprint all
```

```
xstatus all
xstatus --<ipname>
xstatus --<ipname>
```

- Typical usage: application hangs
  - For example: host program is waiting for command queue to finish or on an event
  - Use **xprintf** to see unfinished events

© Copyright 2018 Xilinx

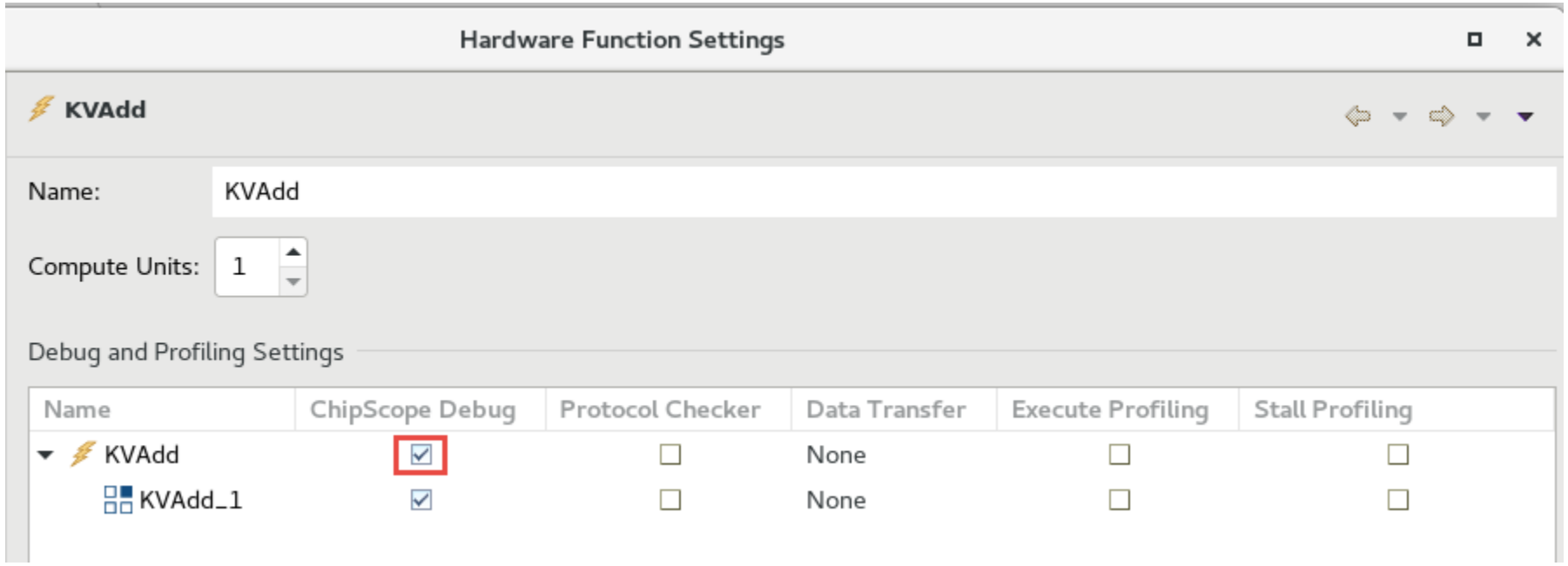
Hardware Debugging

Debugging with ChipScope Bridge and Cores

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

Enabling ChipScope Debug and Protocol Checker

- **In SDAccel GUI, select project.sdx to view project settings**
- **Under Hardware Function settings pane, click checkboxes 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
- **Instance 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**

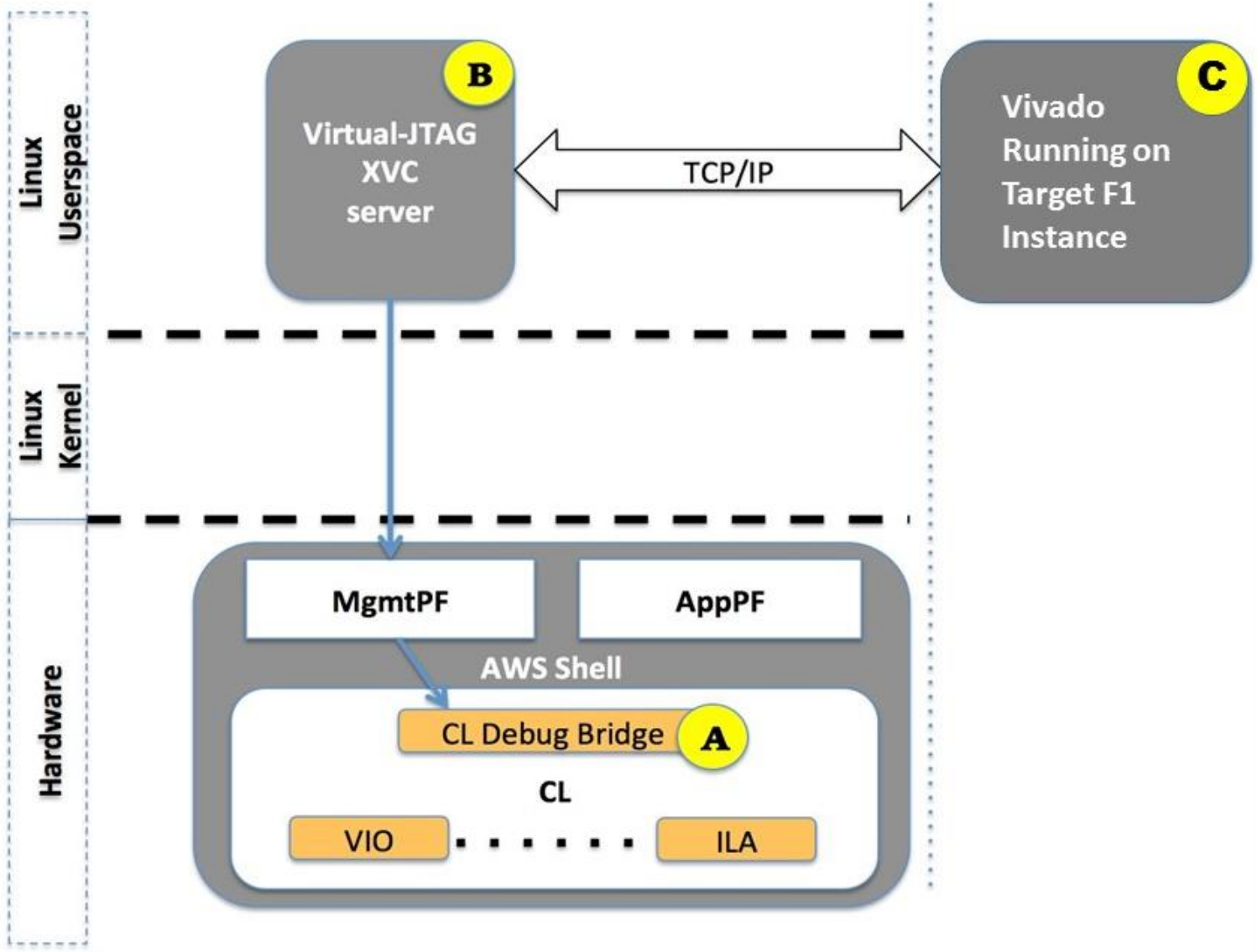
Xilinx Virtual Cable for Debugging

- **EC2 FPGA platforms support Virtual JTAG capability by emulating JTAG over PCIe**
- **AWS FPGA Management Tools enables running an in-target service (in Linux userspace) implementing Xilinx Virtual Cable (XVC) protocol which allows (local or remote) Vivado to connect to a target FPGA**
- **Execute following command in Linux shell on the target instance to start the server**

```
$ sudo fpga-start-virtual-jtag -P 10201 -S 0
```

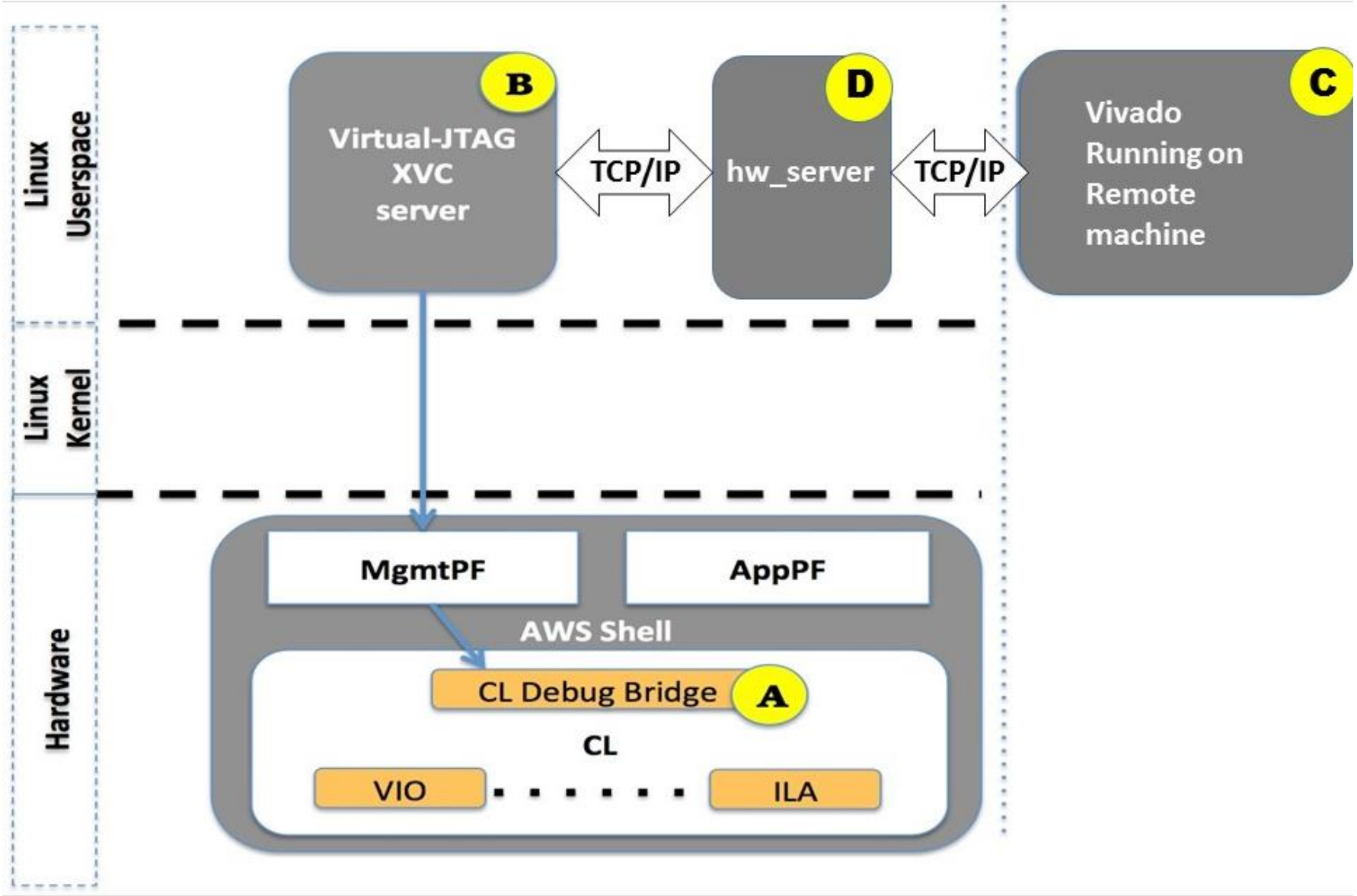
  - `-P 10201` is the port for which XVC should have been configured
  - `-S 0` is the slot number where the FPGA AFI is typically loaded

Debugging Locally on AWS F1



XILINX.

Debugging Remotely



Debugging Steps

- Enable ChipScope bridge
- 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

Summary

Summary

- Host code can be debugged in software emulation, hardware emulation, and system run
- Kernel code can be debugged in software emulation and hardware emulation (limited)
- Use printf() to debug kernels
- Use ChipScope bridge, ILA, VIO, performance monitor, and protocol checker cores to debug hardware
- Use Xilinx Virtual Cable management tool to talk to the hardware

Lab Intro

Lab Intro

- **This lab is continuation of the previous (RTL-Kernel Wizard Lab) lab. You will add ChipScope cores to monitor the activities taking place at the kernel interface level and perform software debugging using SDx debug capabilities.**

Adaptable.
Intelligent.

XILINX