SDSoC Environment Debugging Guide

UG1282 (v2018.2) July 2, 2018

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
07/02/2018 Version 2018.2	
Entire document	Editorial updates.
06/06/2018 Version 2018.2	
General updates	Initial Xilinx release.

Table of Contents

Introduction to Debugging in SDSoC

The Software-Defined Sytem-On-Chip (SDSoC™) Environment is a tool suite that includes an Eclipse-based integrated development environment (IDE) for implementing heterogeneous embedded systems. SDSoC supports Arm® Cortex™-based applications using the Zynq®-7000 SoC and Zynq UltraScale+ MPSoC devices, as well as MicroBlaze™ processor-based applications on all Xilinx® SoCs and FPGAs.

This user guide is intended to introduce SDSoC debugging capabilities. The goal is to provide you with detailed instructions on how to analyze any failure encountered within the SDSoC flow. It is important to note that if no tool problems are encountered and the behavior of the design is deemed functionally correct, you can look for answers in the *SDSoC Environment Profiling and Optimization Guide* (UG1235) to examine whether the performance of the design can be further improved.

The systems produced by the SDSoC Environment are high-performance and complex hardware and software. There are different aspects to debugging a SDSoC design.

SDSoC Environment Overview

The SDSoC™ Environment also includes system compilers that transform C/C++ programs into complete hardware/software systems with select functions compiled into programmable logic.

The SDSoC system compilers analyze a program to determine the data flow between software and hardware functions, and generate an application specific system-on-chip to realize the program. To achieve high performance, each hardware function runs as an independent thread; the system compilers generate hardware and software components that ensure synchronization between hardware and software threads, while enabling pipelined computation and communication. Application code can involve many hardware functions, multiple instances of a specific hardware function, and calls to a hardware function from different parts of the program.

The SDSoC IDE supports software development workflows including profiling, compilation, linking, system performance analysis, and debugging. In addition, the SDSoC environment provides a fast performance estimation capability to enable "what if" exploration of the hardware/software interface before committing to a full hardware compile.

The SDSoC system compilers target a base platform and invoke the Vivado® High-Level Synthesis (HLS) tool to compile synthesizeable C/C++ functions into programmable logic. They then generate a complete hardware system, including DMAs, interconnects, hardware buffers, other IP, and the Field Programmable Gate Array (FPGA) bitstream by invoking the Vivado Design Suite tools. To ensure all hardware function calls preserve their original behavior, the SDSoC system compilers generate system-specific software stubs and configuration data. The program includes function calls to drivers required to use the generated IP blocks. Application and generated software is compiled and linked using a standard GNU toolchain.

By generating complete applications from "single source," the system compilers let you iterate over design and architecture changes by refactoring at the program level, reducing the time needed to achieve working programs running on the target platform.

Terminology

The following terms are widely used in the context of SDSoC™ designs. The terms are their description is provided below.

- **Accelerators:** Portions of the application code that have been implemented in the hardware in the FPGA fabric. These are also called hardware functions.
- **Data Movers:** The data mover transfers data between Processing System and accelerators, and among accelerators. The SDSoC Environment can generate various types of data movers based on the properties and size of the data being transferred.
- **Pipelining:** Pipelining is a digital design technique that allows the designer to avoid data dependencies and increase the level of parallelism in an algorithm hardware implementation. The data dependence in the original software implementation is preserved for functional equivalence, but the required circuit is divided into a chain of independent stages. All stages in the chain run in parallel on the same clock cycle. The only difference is the source of data for each stage. Each stage in the computation receives its data values from the result computed by the preceding stage during the previous clock cycle.
- **Pragmas:** Special directives that can be inserted int the source code to guide the system compiler. In the SDSoC Environment, you control the system generation process by structuring hardware functions and calls to hardware functions to balance communication and computation, and by inserting pragmas into your source code to guide the system compiler.
- **Processors:** Processors in the context of the SDSoC Environment mean a soft processor such as a MicroBlaze™ or a hard processor such as the Arm® processors on Zynq®-7000 SoCs and Zynq® UltraScale+™ MPSoC.

- **System Port:** A system port connects a data mover to the PS. It can be an ACP, AFI (corresponding to high-performance ports), MIG (corresponding to a PL-based DDR memory controller), or a stream port on the Zynq.

Elements of SDSoC

The SDSoC™ Environment inherits many of the tools in the Xilinx® Software Development Kit (SDK), including:

- GNU toolchains and standard libraries (for example, glibc)
- The Target Communication Framework (TCF)
- A performance analysis perspective within the Eclipse/CDT-based GUI
- Command-line tools

The SDSoC Environment includes the sds++ compiler that generates complete hardware/software systems, an Eclipse-based user interface to create and manage projects and workflows, and a system performance estimation capability to explore different "what if" scenarios for the hardware/software interface.

The sds++ compiler employs underlying tools from the Vivado® Design Suite (System Edition), including Vivado HLS, IP integrator, IP libraries for data movement and interconnect, and the RTL synthesis, placement, routing, and bitstream generation tools.

The principle of design reuse underlies workflows that you employ with the SDSoC Environment, using well-established platform-based design methodologies. The SDSoC system compiler generates an application-specific system-on-chip for a targeted platform. The environment includes a number of standard base-platforms for application development, and other platforms can be developed by third-party partners, or by SDSoC design teams. The *SDSoC Environment Platform Development Guide* (UG1146) describes how to create a platform design using the Vivado Design Suite, specify platform properties to define and configure platform interfaces, and define the corresponding software run-time environment to build a platform for use in the SDSoC Environment.

An SDSoC platform defines a base hardware and software architecture and application context, including:

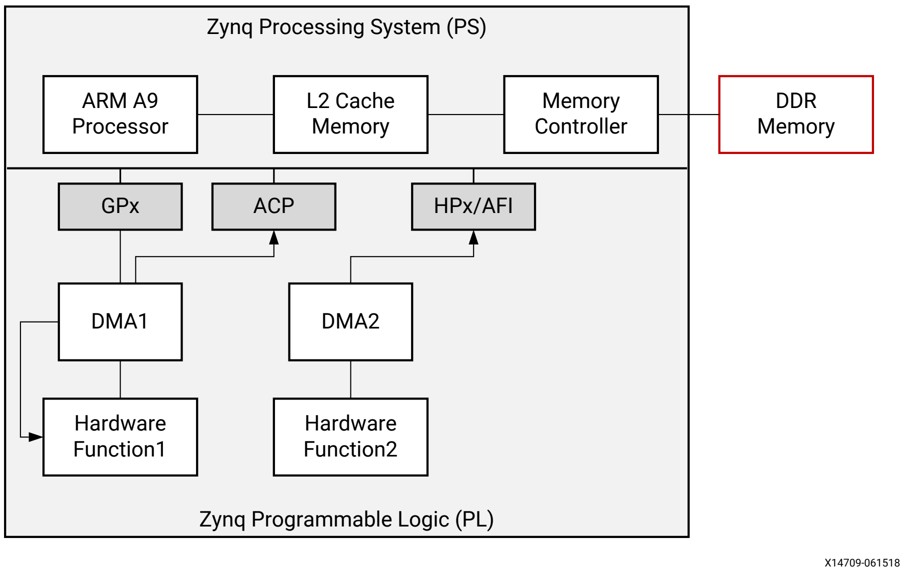
- Processing system
- External memory interfaces
- Custom I/O
- Embedded processor operating system:
    - Boot loaders

- Drivers for root system and peripherals

Every project you create within the SDSoC Environment targets a specific platform, and you employ the tools within the SDx™ IDE to customize the platform with application-specific hardware accelerators and data motion networks connecting accelerators to the platform. In this way, you can easily create highly tailored application-specific systems-on-chip for different base platforms, and can reuse base platforms for many different application-specific systems-on-chip.

A simplified Zynq® and DDR configuration with memory access ports and hardware accelerators is shown below.

Figure 1: **Simplified Zynq + DDR Diagram Showing Memory Access Ports and Memories**

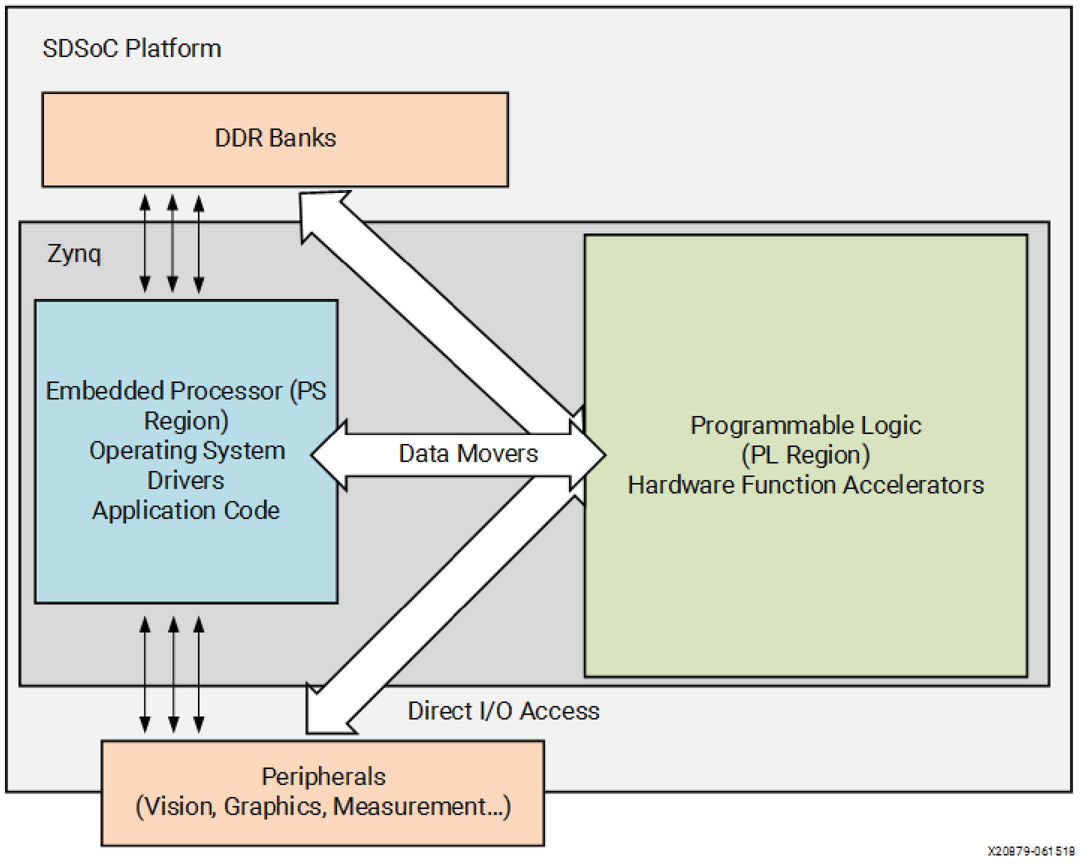


Execution Model of an SDSoC Application

The execution model for an SDSoC™ application can be understood in terms of the normal execution of a C++ program running on the target CPU after the platform has booted. It is useful for the programmer to be aware of how a C++ binary executable interfaces to hardware.

The set of declared hardware functions within a program is compiled into hardware accelerators that are accessed with the standard C run time through calls into these functions. Each hardware function call in effect invokes the accelerator as a task, and each of the arguments to the function is transferred between the CPU and the accelerator, accessible by the program after accelerator task completion. Data transfers between memory and accelerators are accomplished through data movers; either a direct memory access (DMA) engine automatically inserted into the system by the `sds++` system compiler, or by the hardware accelerator itself (such as the `zero_copy` data mover).

Figure 2: **Architecture of an SDSoC System**



To ensure program correctness, the system compiler intercepts each call to a hardware function and replaces it with a call to a generated stub function that has an identical signature, but with a derived name. The stub function orchestrates all data movement and accelerator operation, synchronizing software and accelerator hardware at exit of the hardware function call. Within the stub, all accelerator and data mover control is realized through a set of send/receive APIs provided by the `sds_lib` library.

When program dataflow between hardware function calls involves array arguments that are not accessed after the function calls have been invoked within the program (other than destructors or `free()` calls), and when the hardware accelerators can be connected via streams, the system compiler will transfer data from one hardware accelerator to the next through direct hardware stream connections rather than implementing a round trip to and from memory. This optimization can result in significant performance gains and reduction in hardware resources.

At a high level, the SDSoC execution model of a program includes the following steps.

1. Initialization of the `sds_lib` library occurs during the program's constructor before entering `main()`.

2. Within a program, every call to a hardware function is intercepted by a function call into a stub function with the same function signature (other than name) as the original function. Within the stub function, the following steps occur:

   a. A synchronous accelerator task control command is sent to the hardware.

   b. For each argument to the hardware function, an asynchronous data transfer request is sent to the appropriate data mover, with an associated `wait()` handle. A non-void return value is treated as an implicit output scalar argument.

   c. A barrier `wait()` is issued for each transfer request. If a data transfer between accelerators is implemented as a direct hardware stream, the barrier `wait()` for this transfer occurs in the stub function for the last in the chain of accelerator functions for this argument.

3. Cleanup of the `sds_lib` library occurs during the program's destructor upon exiting `main()`.

TIP: *Steps 2a-c ensure that program correctness is preserved at entrance and exit of accelerator pipelines, while enabling concurrent execution within the pipelines.*

Sometimes the programmer has insight of potential concurrent execution of accelerator tasks that cannot be automatically inferred by the system compiler. In this case, the `sds++` system compiler supports a `#pragma SDS async(ID)` that can be inserted immediately preceding a call to a hardware function. This pragma instructs the compiler to generate a stub function without any barrier `wait()` calls for data transfers. As a result, after issuing all data transfer requests, control returns to the program, enabling concurrent execution of the program while the accelerator is running. In this case, it is the programmer's responsibility to insert a `#pragma SDS wait(ID)` within the program at appropriate synchronization points, which are resolved into `sds_wait(ID)` API calls to correctly synchronize hardware accelerators, their implicit data movers, and the CPU.

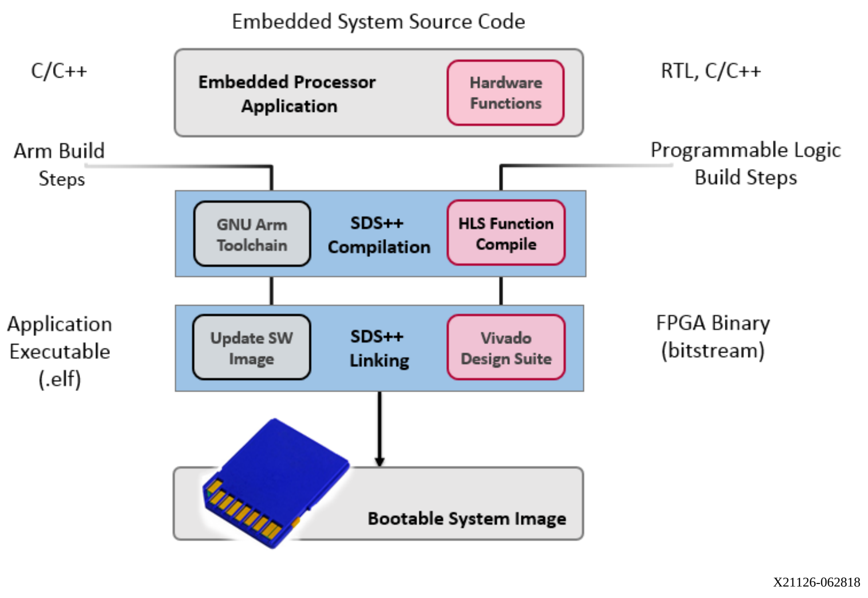
IMPORTANT!: *Every `async(ID)` pragma requires a matching `wait(ID)` pragma.*

SDSoC Build Process

The SDSoC™ environment offers all of the features of a standard software development environment: optimized cross-compilers for the embedded processor application and the hardware function, robust debugging environment to help you identify and resolve issues in the code, performance profilers to let you identify the bottlenecks and optimize your code. Within this environment the SDSoC build process uses a standard compilation and linking process. Similar to g++, the `sds++` system compiler invokes sub-processes to accomplish compilation and linking.

As shown in the image below, compilation is extended not only to object code that runs on the CPU, but also includes compilation and linking of hardware functions into IP blocks using the Vivado® HLS tool, and creating standard object files (.o) using the target CPU toolchain. System linking consists of program analysis of caller/callee relationships for all hardware functions, and generation of an application-specific hardware/software network to implement every hardware function call. The `sds++` system compiler invokes all necessary tools, including Vivado HLS (function compiler), the Vivado® Design Suite to implement the generated hardware system, and the Arm® compiler and linker to create the application binaries that run on the CPU, invoking the accelerator (stubs) for each hardware function by outputting a complete bootable system for an SD card.

Figure 3: **SDSoC Build Process**



- The compilation process includes the following tasks:
    - Analyze the code and run a compilation for the main application running on the Arm processor, and a separate compilation for each of the hardware accelerators.

  - The application code is compiled through standard GNU Arm compilation tools with an object (`.o`) file produced as final output.
  - The hardware accelerated functions are run through the Vivado® HLS tools, to start the process of custom hardware creation, with an object (`.o`) file as output.
- After compilation, the linking process includes the following tasks:
  - Analyze the data movement through the design, and modify the hardware platform to accept the accelerators.
  - Implement the hardware accelerators into the programmable logic (PL) region using the Vivado® Design Suite to run synthesis and implementation, and generate the bitstream for the device.
  - Update the software images with hardware access APIs, to call the hardware functions from the embedded processor application.
  - Produce an integrated SD Card image that can boot the board with the application in an ELF file.

Build Targets

As an alternative to building a complete system, you can create an emulation model that will consist of the same platform and application binaries. In this target flow, the `sds++` system compiler will create a simulation model using the source files for the accelerator functions.

The SDSoC environment provides two different build targets, an emulation target used for debug and validation purposes and the system hardware target used to generate the actual FPGA binary:

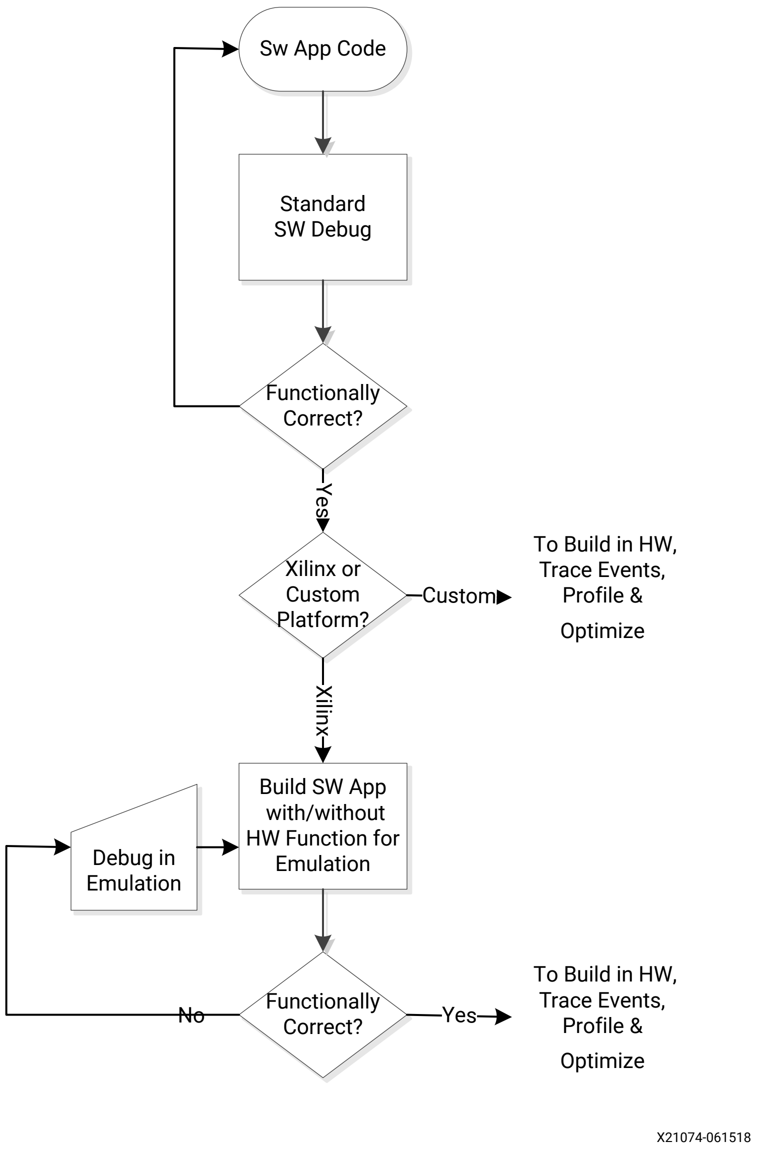
- System Emulation: With system emulation you can debug RTL level transactions in the entire system (PS/PL). Running your application on SDSoC emulator (`sdsoc_emulator`) gives you visibility of data transfers with a debugger. You can debug system hangs, and can inspect associated data transfers in the simulation waveform view, which gives you visibility into signals on the hardware blocks associated with the data transfer.
- Hardware: During hardware execution, you can use the actual hardware platform to run the accelerated hardware functions. The difference between a debug system configuration and the final build of the application code, and the hardware functions, is the inclusion of special debug logic in the platform, such as System ILAs and VIO debug cores, and AXI performance monitors for debug purposes.

SDSoC Debug Flow Overview

The systems produced by the SDSoC™ Environment are high-performance and complex systems, composed of hardware and software components. It can be difficult to understand the execution of applications in such systems with portions of software running in a processor, hardware accelerators executing in the programmable fabric, and many simultaneous data transfers occurring. The SDSoC environment lets you create and debug projects using the Xilinx® Software Debugger, and provides sophisticated hardware/software event tracing with automatic system instrumentation that provides an integrated time line view of data transfers and accelerator tasks including driver software setup as well as execution in hardware. Outside the SDx™ IDE, you can use command-line or scripting options to debug your projects.

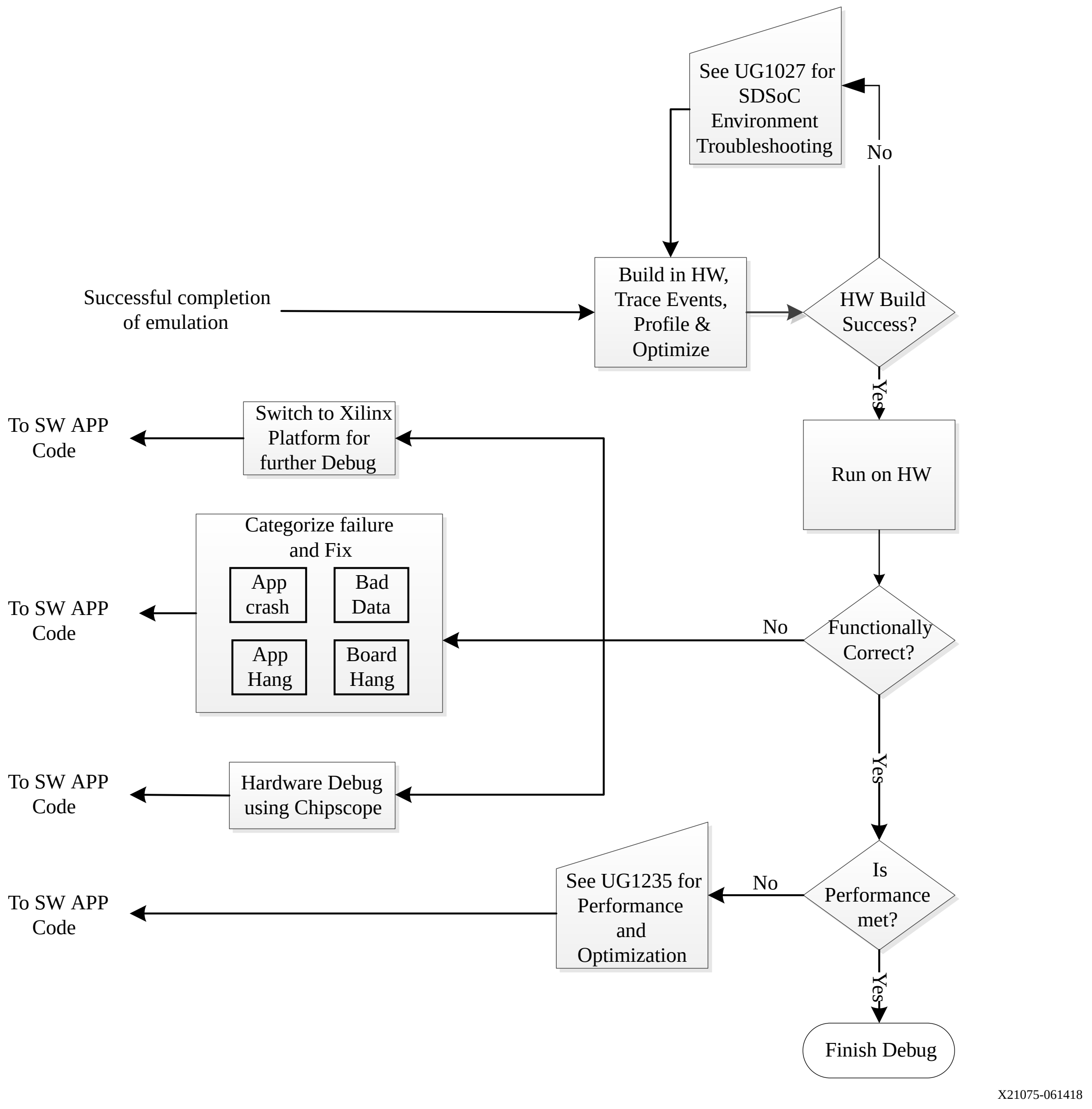
There are several aspects to debugging an SDSoC project. After you identify the hardware functions, you can use system emulation to compile the logic and verify the entire system. This provides a QEMU based emulator that runs the cross-compiled Arm® code, interacting with the XSim RTL simulator within the Vivado® Design Suite. Together, the emulator runs the same binaries as would run in hardware, with full visibility into the hardware; the RTL simulator can display waveforms or can be run without waveforms for faster simulation speed. The emulator can be run within the SDx IDE or on the command line (`sdsoc_emulator`), providing accurate visibility of the final hardware implementation without the need to compile the system into a bitstream and program the device on the board.

Figure 4: **System Emulation Flow**


Sw App Code
Standard SW Debug
Functionally Correct?
Yes
Xilinx or Custom Platform?
Custom
To Build in HW, Trace Events, Profile & Optimize
Xilinx
Debug in Emulation
Build SW App with/without HW Function for Emulation
Functionally Correct?
No
Yes
To Build in HW, Trace Events, Profile & Optimize
X21074-061518

You can also enable hardware and software event tracing to analyze the execution of events and identify any issues. Finally, if there are problems with respect to the hardware design itself, you can use hardware debug in ChipScope by inserting debug cores in the hardware functions implemented on the SDSoC platform. The following flow chart shows a typical hardware build and debug process.

Figure 5: **Hardware Build and Debug Flow**


See UG1027 for SDSoC Environment Troubleshooting
No
Successful completion of emulation
Build in HW, Trace Events, Profile & Optimize
HW Build Success?
Yes
To SW APP Code
Switch to Xilinx Platform for further Debug
Run on HW
Categorize failure and Fix
App crash
Bad Data
App Hang
Board Hang
To SW APP Code
No
Functionally Correct?
Yes
To SW APP Code
Hardware Debug using Chipscope
To SW APP Code
See UG1235 for Performance and Optimization
No
Is Performance met?
Yes
Finish Debug
X21075-061418

You will notice that the two flow diagrams above, System Emulation Flow and Hardware Build and Debug Flow, are connected together by the "To Build in HW" and the "To SW APP" connectors. Standard platforms support both flows, while custom platforms without emulation capabilities support only the hardware build and debug flow.

System Emulation

By running system emulation you can debug RTL level transactions in the entire system (PS/PL). Running your application on the SDSoC™ emulator is a good way to gain visibility of data transfers with a debugger. You will be able to see issues such as a system hang, and can then inspect associated data transfers in the simulation waveform view, which gives you access to signals on the hardware blocks associated with the data transfer.

This is one of the most capable debug features in the SDSoC environment. It can help debug functional issues and determine why an application is hanging. This feature is only available on Xilinx base platforms.

Base platform: Xilinx provides the ZC702, ZC706, ZCU102, ZCU104, ZCU106 and ZedBoard base platforms. All platforms support target Linux, FreeRTOS, or Standalone (bare metal).

Event Tracing

The SDSoC™ environment tracing feature provides you a detailed view of what is happening in the system during the execution of an application. Trace events are produced and gathered into a timeline view, giving you a perspective of the running application. This detailed view can help you understand the performance of your application given the workload, hardware/software partitioning, and system design choices. This view enables event tracing of software running on the processor, as well as hardware accelerators and data transfer links in the system. Such information helps you to identify problems, optimize the design, and improve system implementation.

Tracing an application produces a log that records information about system execution. Compared to event logging, event tracing shows the correlation between events for the duration of the event, rather than an instantaneous event at a particular time. The goal of tracing is to help debug execution by observing what happened when, and how long events took. Tracing shows the performance of execution with more granularity than overall runtime. This is best used to analyze performance and get an indication of whether there is an application hang.

Hardware Execution Flow

During hardware execution, you can use the actual hardware platform execute the accelerated hardware functions. The difference between a debug configuration and the final compilation of the application code, and the hardware functions, is the inclusion of special debug logic in the platform, such as System ILAs and VIO debug cores, and AXI performance monitors for debug purposes. The SDSoC™ Environment provides specific hardware debug capabilities which include ChipScope debug cores (such as System ILAs), that can be viewed in Vivado® Hardware Manager, with waveform analysis, kernel activity reports, and memory access analysis to localize these critical hardware issues.

In-system debugging lets you debug your design in real-time, on your target hardware. This is an essential step in design completion. Invariably, there are situations that are extremely hard to replicate in a simulator. Therefore, there is a need to debug the problem in the running hardware. In this step, you place debug cores into your design to provide you the ability to observe and control the design. After the debugging process is complete, you can remove the debug cores to increase performance and reduce resource utilization of the device.

The SDx™ IDE and command line options provide ways to instrument your design for debugging. The `--dk` compiler switch lets you add ILA debug cores to the interfaces of your hardware function, as described in Hardware Debugging in SDSoC Using ChipScope. To debug C-callable IP that are used in your application code, you must have instantiated the needed debug cores into the RTL code of the IP prior to packaging it as a C-callable IP.

IMPORTANT!: *Debugging the hardware function on the SDSoC platform hardware requires additional logic to be incorporated into the overall hardware model. This means that if hardware debugging is enabled, there will be some impact on resource utilization of the Xilinx device, as well as some impact on the performance of the hardware function.*

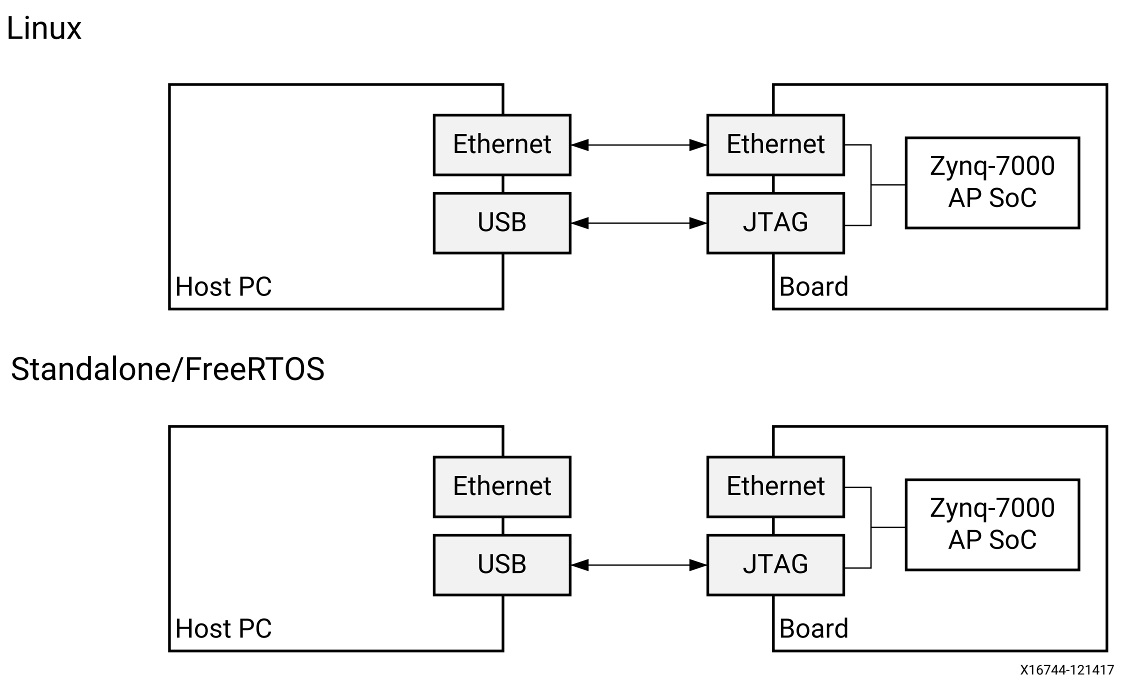
Connecting to the Hardware

The board connection requirements are slightly different depending on the operating system: standalone, FreeRTOS, or Linux.

- For standalone and FreeRTOS, you must download the Executable Link Format file (`.elf`) to the board using the USB/JTAG interface. Trace data is read out over the same USB/JTAG interface as well.
- For Linux, the SDSoC™ environment assumes the OS boots from the SD card. SDx™ then copies the `.elf` and runs it using the TCP/TCF agent running in Linux over the Ethernet connection between the board and host PC. The trace data is read out over the USB/JTAG interface. Both USB/JTAG and TCP/TCF agent interfaces are needed for tracing Linux applications.

The figure below shows the connections required.

Figure 6: **Connections Required When Using Trace with Different Operating Systems**



Chapter 2

SDSoC Debug Features

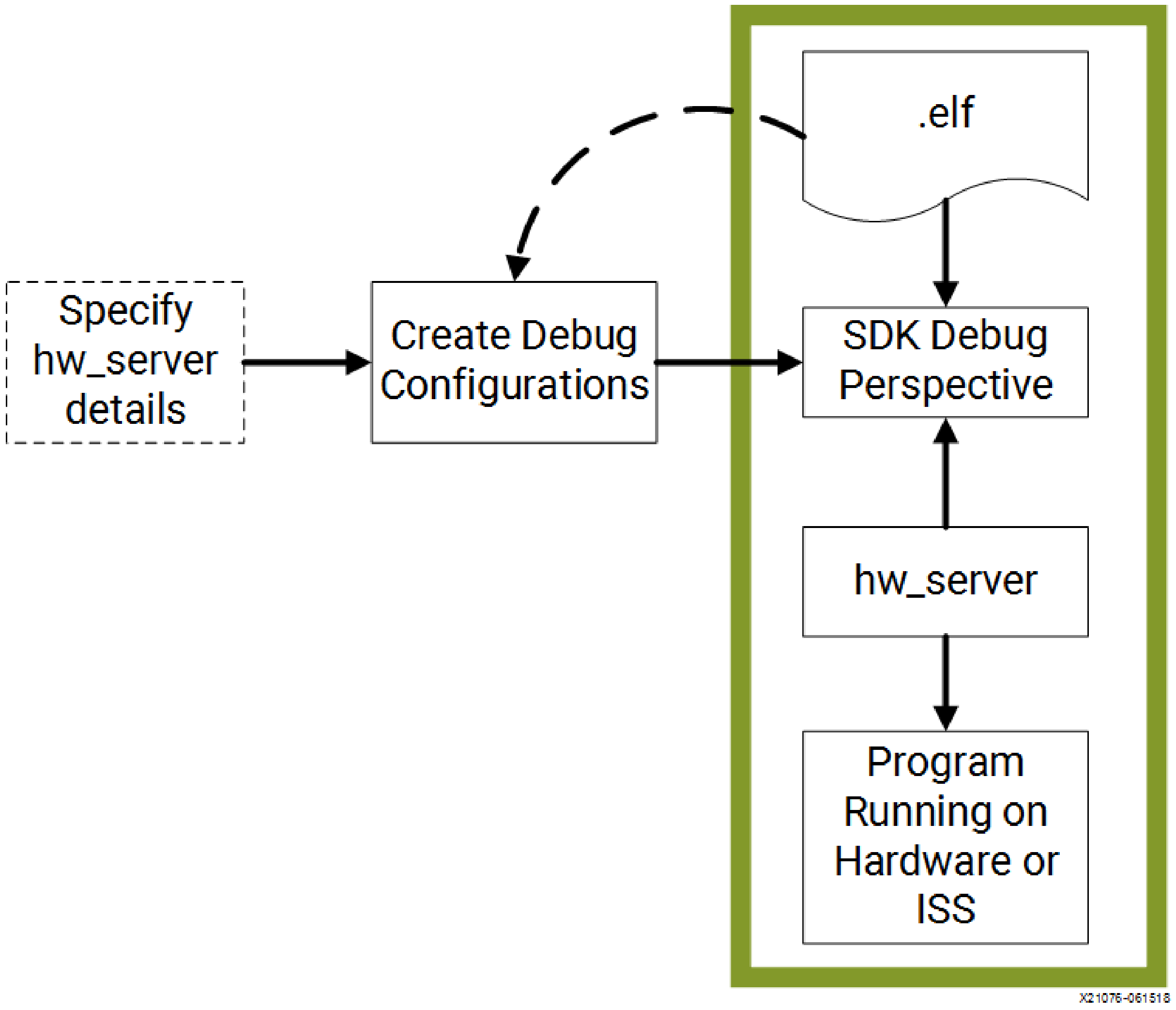
This section provides details on debugging using the GUI or the command line in the SDx™ Environment. Various aspects of debug are also described in detail.

Debug Tools Available in the SDx Environment

The SDx™ environment includes the Xilinx® System Debugger (XSDB) for debugging SDSoC™ designs.

Xilinx System Debugger (XSDB)

Xilinx® System Debugger (XSDB) uses the Xilinx `hw_server` as the underlying debug engine. SDK translates each user interface action into a sequence of Target Communication Frameworks (TCF) commands. It then processes the output from System Debugger to display the current state of the program being debugged. It communicates to the processor on the hardware using Xilinx `hw_server`. You can debug multiple processors simultaneously with a single System Debugger debug configuration. This is the recommended debug engine for SDx™ designs. The System Debugger can either be launched on the hardware or the QEMU engine.


.elf
Specify hw_server details
Create Debug Configurations
SDK Debug Perspective
hw_server
Program Running on Hardware or ISS
X21076-061518

The workflow is made up of the following components:

- Executable and Linkable Format (ELF) File: To debug your application, you must use an ELF or `.elf` file compiled for debugging. The debug `.elf` file contains additional debug information for the debugger to make direct associations between the source code and the binaries generated from that original source. Refer to Build Configurations for more information.
- Debug Configuration: To launch the debug session, you must create a debug configuration in the SDx Enviroment. This configuration captures options required to start a debug session, including the executable name, processor target to debug, and other information. Refer to Launch Configurations for more information.
- SDx Debug Perspective: Using the Debug Perspective, you can manage the debugging or running of a program in the Workbench. You can control the execution of your program by setting breakpoints, suspending launched programs, stepping through your code, and examining the contents of variables.

You can repeat the cycle of modifying the code, building the executable, and debugging the program in the SDx Environment.

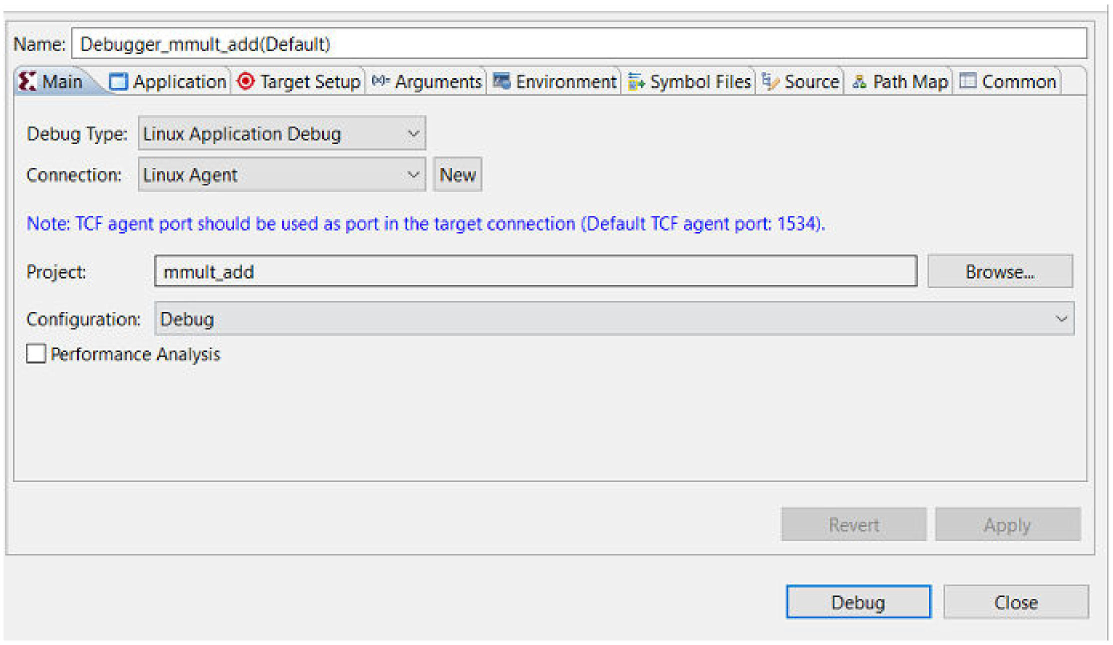
Note: If you edit the source after compiling, the line numbering will be out of step because the debug information is tied directly to the source. Similarly, debugging optimized binaries can also cause unexpected jumps in the execution trace.

Setting Debug Configurations

To debug, run, and profile an application, you must create a launch configuration that captures the settings for executing the application. The configurations for debugging, running, and profiling an application are similar. To launch the **Debug Configuration** dialog, right-click on your application project, and select **Debug As → Debug Configurations**. Alternatively, you can also right-click the **Debug** item under your Project in the Assistant window, and select **Debug → Debug Configurations** from the context menu.

This pops up the **Debug Configurations** dialog box as shown below. Note that the all the tabs may not show up in this dialog box based on the application being debugged. The options presented in these tabs may also be slightly different.

Figure 7: **Debug Configurations**



Name: Debugger_mmult_add(Default)

Debug Type: Linux Application Debug

Connection: Linux Agent

Note: TCF agent port should be used as port in the target connection (Default TCF agent port: 1534).

Project: mmult_add

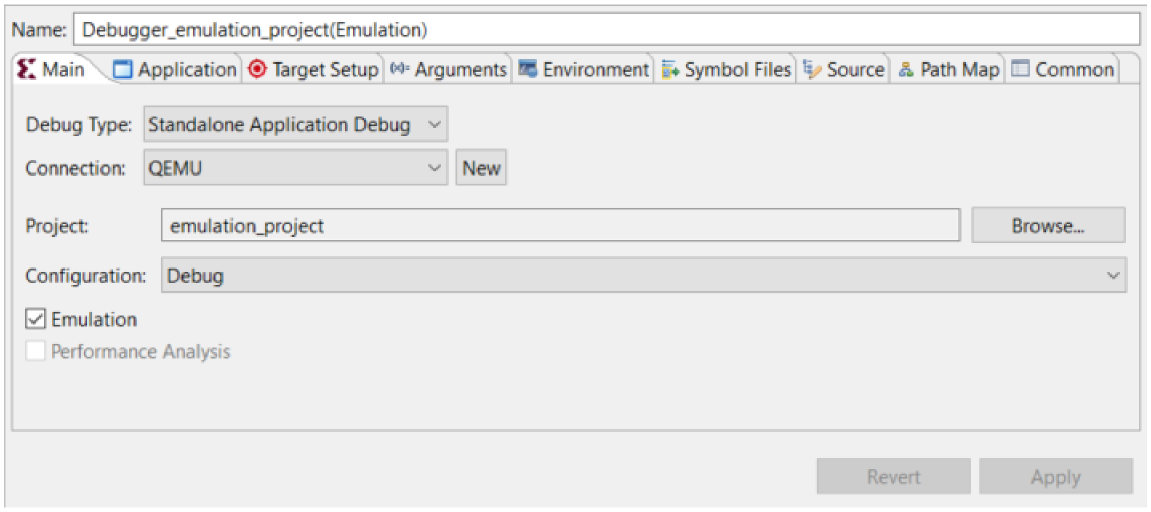
Configuration: Debug

☐ Performance Analysis

Various tabs of this dialog box are described in the following sections.

Main Tab

The **Main** tab shows the drop-down menus for **Debug Type**, and the **Connection**. It also lets you change **Project** or the **Configuration** of the project. The **Debug Type** field can be set to **Standalone Application Debug**, **Linux Application Debug**, or **Attach to running target**.



Name: Debugger_emulation_project(Emulation)

Debug Type: Standalone Application Debug

Connection: QEMU

Project: emulation_project

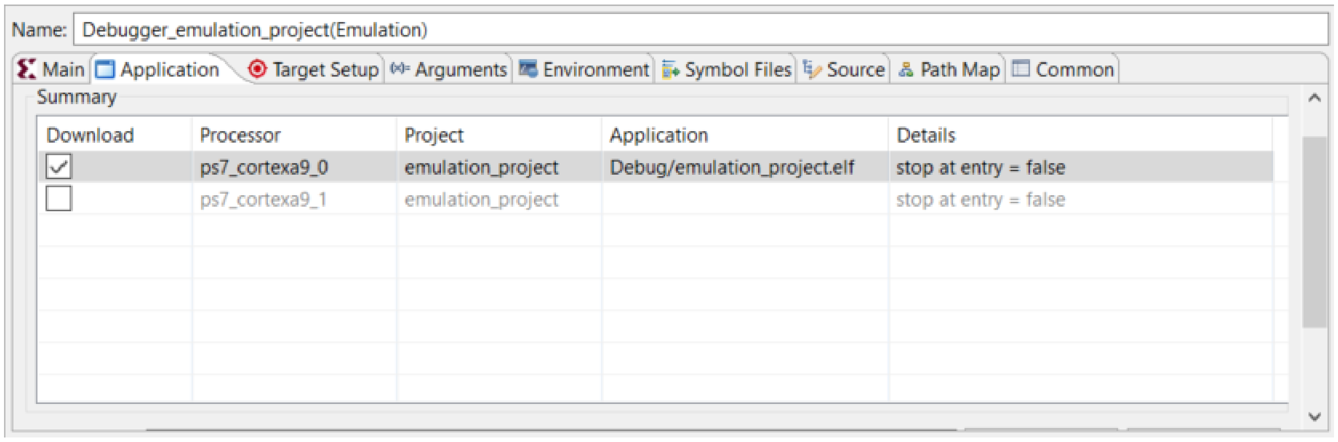
Configuration: Debug

☑ Emulation

☐ Performance Analysis

Application Tab

The **Application** tab shows the processor instance that has been selected to be debugged for the project, and shows other pertinent information such as the Application `.elf` that is being downloaded to be run on the processor.



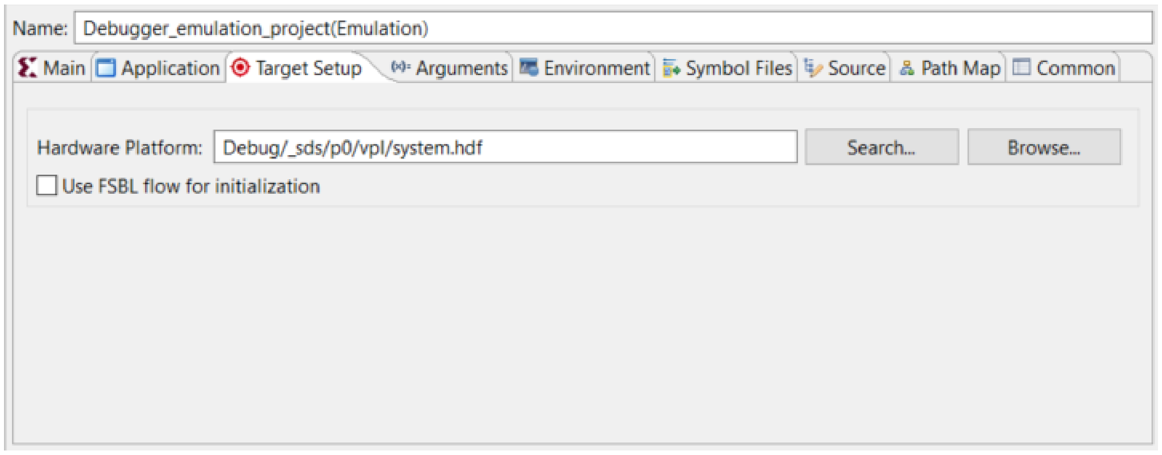
Name: Debugger_emulation_project(Emulation)

Summary

Download	Processor	Project	Application	Details
✓	ps7_cortexa9_0	emulation_project	Debug/emulation_project.elf	stop at entry = false
☐	ps7_cortexa9_1	emulation_project		stop at entry = false

Target Setup Tab

The **Target Setup** tab presents the option to specify the **Hardware Platform** and/or **Use FSBL flow for initialization**.


Name: Debugger_emulation_project(Emulation)

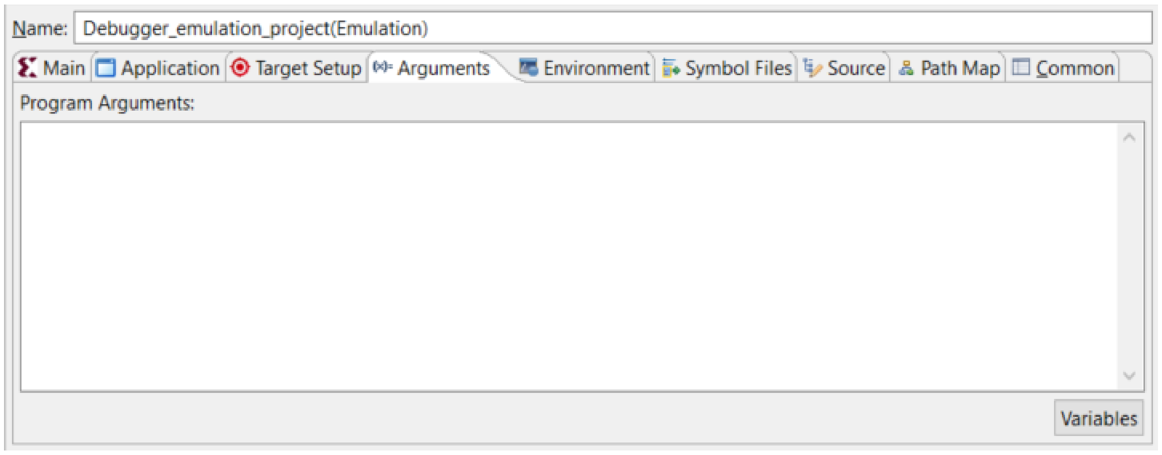
Main | Application | Target Setup | Arguments | Environment | Symbol Files | Source | Path Map | Common

Hardware Platform: Debug/_sds/p0/vpl/system.hdf

Use FSBL flow for initialization

Arguments Tab

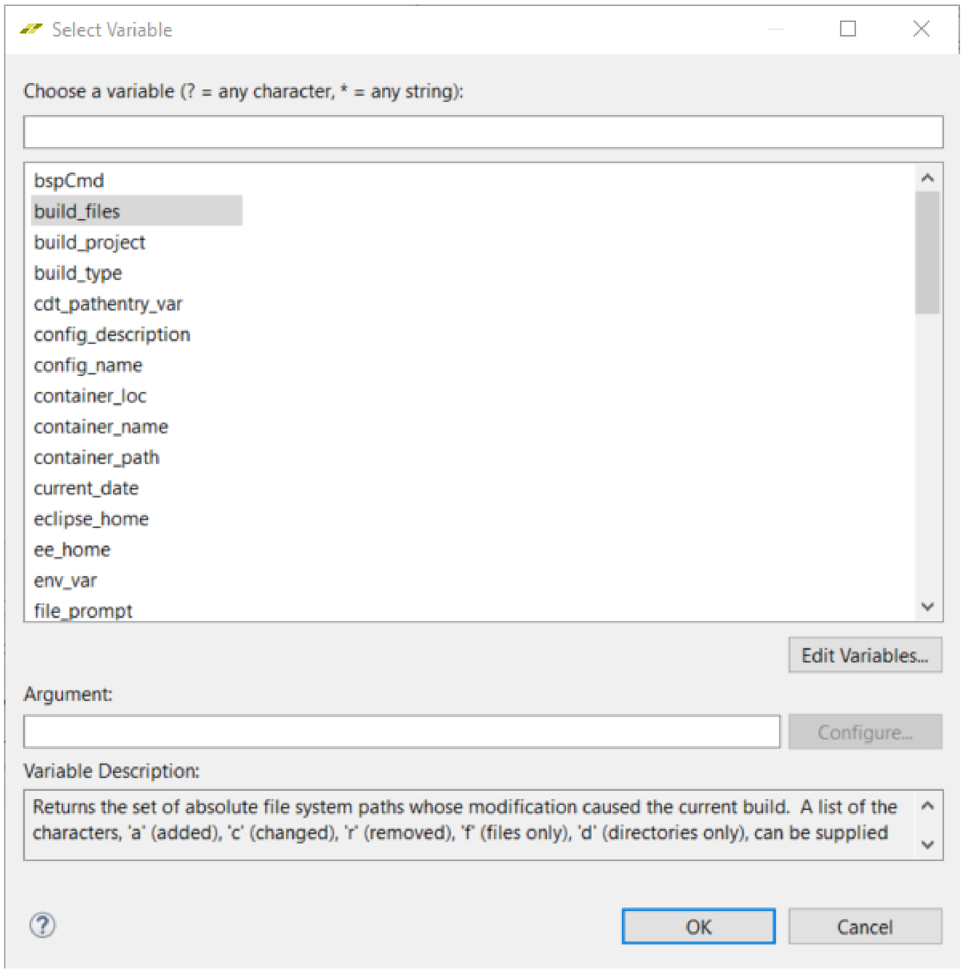
The **Arguments** tab lets you specify any variables that are needed for launching the debug session. Click the **Variables** button, to bring up the **Variables** dialog box, which allows you to set a variable.


Name: Debugger_emulation_project(Emulation)

Main | Application | Target Setup | Arguments | Environment | Symbol Files | Source | Path Map | Common

Program Arguments:

After clicking the **Variables** button the **Select Variable** dialog box appears.



Select Variable

Choose a variable (? = any character, * = any string):

bspCmd
build_files
build_project
build_type
cdt_pathentry_var
config_description
config_name
container_loc
container_name
container_path
current_date
eclipse_home
ee_home
env_var
file_prompt

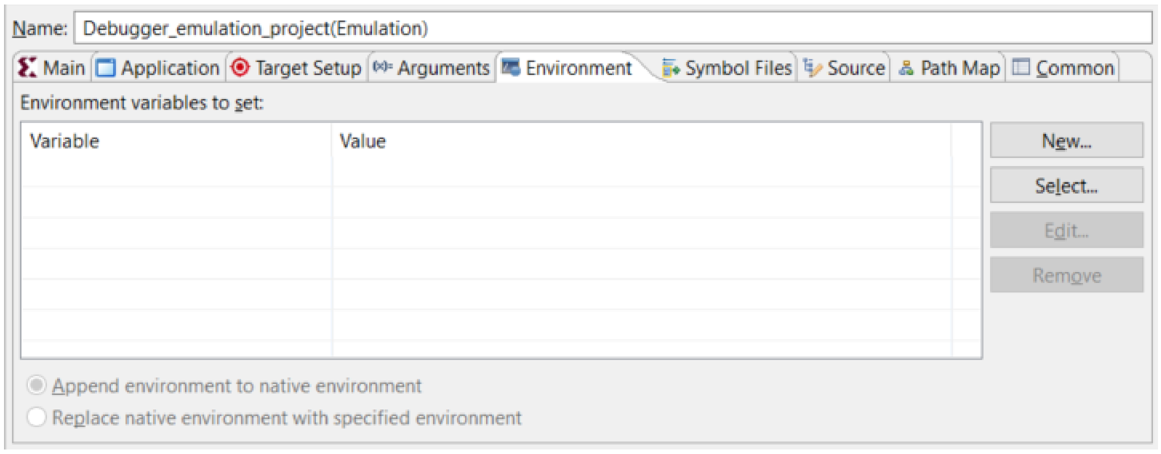
Argument:

Variable Description:

Returns the set of absolute file system paths whose modification caused the current build. A list of the characters, 'a' (added), 'c' (changed), 'r' (removed), 'f' (files only), 'd' (directories only), can be supplied

Environment Tab

The **Environment** tab allows you to set any environment variables for the Debug Configurations.



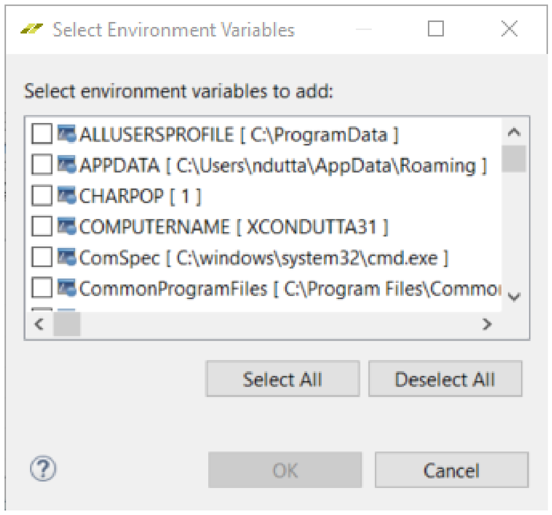
Name: Debugger_emulation_project(Emulation)

Environment variables to set:

Variable	Value

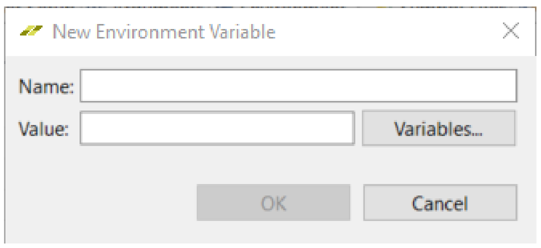
Append environment to native environment
Replace native environment with specified environment

Clicking the **Select** button brings up the existing environment variable that could be set to specific values.


Select Environment Variables
Select environment variables to add:
Select All
Deselect All
OK
Cancel

ALLUSERSPROFILE [C:\ProgramData]
APPDATA [C:\Users\ndutta\AppData\Roaming]
CHARPOP [1]
COMPUTERNAME [XCONDUTTA31]
ComSpec [C:\windows\system32\cmd.exe]
CommonProgramFiles [C:\Program Files\Commo

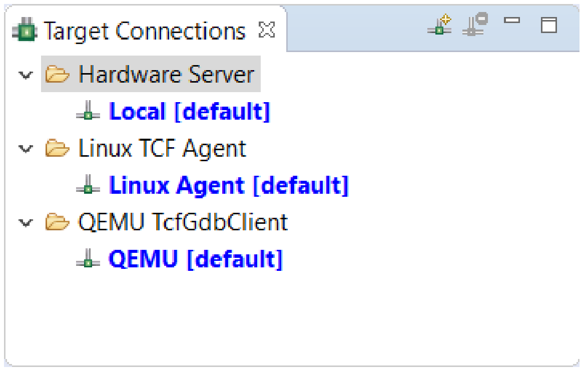
Clicking **New** allows you to create and set a new environment variables.



Target Connections

The **Target Connections** view allows you to configure multiple remote targets. It shows connected targets and gives you an option to add or delete target connections.

The SDx™ Environment establishes target connections through the Hardware Server agent. In order to connect to remote targets, the hardware server agent must be running on the remote host, which is connected to the target.



Refer to Connecting to the Hardware for more information.

Debugging Linux Applications in the SDSoC IDE

Within the SDSoC™ IDE, use the following procedure to debug your application:

1. Make sure the board is connected to your host computer using the JTAG Debug connector, and there is an Ethernet connection between the board and host PC.
2. Set the board to boot from the SD card, per the relevant SDSoC platform User Guide.
3. Select the **Debug** as the active build configuration and build the project.
4. Copy the generated `Debug/sd_card` image to an SD card, and boot the board.
5. Make sure the board is connected to the network, and note its IP address, for example, by executing `ifconfig eth0` on the board at the command prompt using a terminal communicating with the board over UART.
6. Select the **Debug As** option to create a new debug-configuration, and enter the IP address for the board.
7. You now switch to the SDSoC Environment debug perspective which allows you to start, stop, step, set breakpoints, examine variables and memory, and perform various other debug operations.

Debugging Standalone/FreeRTOS Applications in the SDSoC IDE

Use the following procedure to debug a standalone (bare-metal) or FreeRTOS application project using the SDSoC™ IDE.

1. Make sure the board is connected to your host computer using the JTAG Debug connector, and set the board to boot from JTAG.
2. Select **Debug** as the active build configuration, and build the project.
3. Select the **Debug As** option to create a new debug-configuration.

   You now switch to the SDSoC Environment debug perspective, which allows you to start, stop, step, set breakpoints, examine variables and memory, and perform various other debug operations.

   In the SDSoC IDE toolbar, click the **Debug** option, which provides a shortcut to the procedure described above.

Xilinx Software Command Line Tool (XSCT)

Graphical development environments such as the Xilinx® SDx™ are useful for getting up to speed on development for a new processor architecture. It helps to abstract away and group most of the common functions into logical wizards that even the novice can use. However, scriptability of a tool is also essential for providing the flexibility to extend what is done with that tool. It is particularly useful when developing regression tests that will be run nightly or running a set of commands that are used often by the developer.

Xilinx Software Command-line Tool (XSCT) is an interactive and scriptable command-line interface to Xilinx SDx. As with other Xilinx tools, the scripting language for XSCT is based on Tools Command Language (Tcl). You can run XSCT commands interactively or script the commands for automation. XSCT supports the following actions:

- Create hardware, board support packages (BSPs), and application projects.
- Manage repositories.
- Set toolchain preferences.
- Configure and build BSPs/applications
- Download and run applications on hardware targets.
- Create and flash boot images by running Bootgen and program_flash tools.

For information on XSCT commands please refer to the XSCT section in *SDK Online Help* (UG782).

System Emulation

System emulation can be run on System Debugger using the Target Communications Framework (TCF) Server. Note that emulation is not supported for custom platforms currently. Only the base platforms provided by Xilinx® support emulation.

Running System Emulation from the GUI

System emulation provides the same level of accuracy as the final implementation without the need to compile the system into a bitstream and program the device on the board.

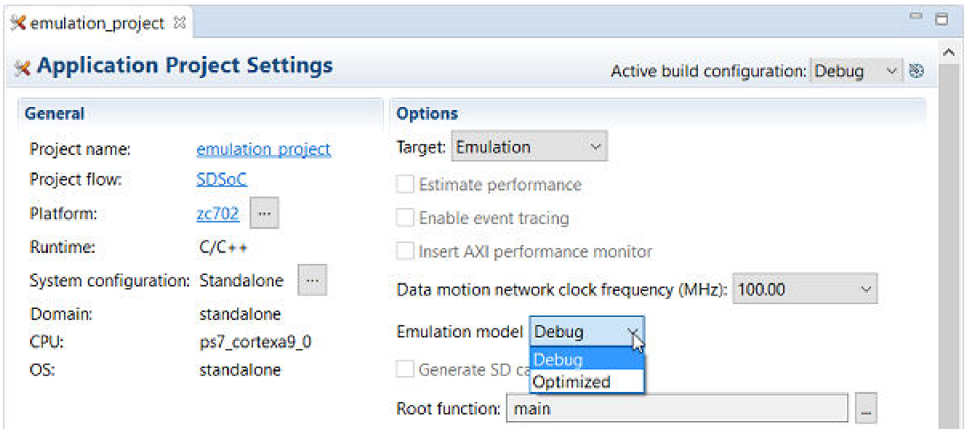
Next, see:

- Enabling System Emulation
- Invoking the System Emulator
- Viewing Emulation Output

Enabling System Emulation

To enable system emulation, within the **Application Project Settings** window do the following:

1. Set the **Active build configuration** to **Debug**.
2. Set the **Target** to **Emulation**.
3. Set the **Emulation Model** to either **Debug** or **Optimized**.



emulation_project

Application Project Settings

Active build configuration: Debug

General

Project name: emulation_project
Project flow: SDSoC
Platform: zc702
Runtime: C/C++
System configuration: Standalone
Domain: standalone
CPU: ps7_cortexa9_0
OS: standalone

Options

Target: Emulation
Estimate performance
Enable event tracing
Insert AXI performance monitor
Data motion network clock frequency (MHz): 100.00
Emulation model Debug
Generate SD ca
Root function: main

Because emulation does not require a full system compile, the tool disables the generation of the bitstream and **Generate SD card image** option to improve run time and iteration time. System emulation allows you to verify and debug the system with the same level of accuracy as a full bitstream compilation.

Selecting the Emulation Model

Emulation model offers two modes:

- **Debug**: Builds the system through RTL generation, and the IP integrator block design containing the hardware function, elaborates the hardware design, and runs behavioral simulation on the design, with a Waveform viewer to help you analyze the results. Users will interact with the XSim simulator inside Vivado® Design Suite to analyze the waveforms.
- **Optimized**: Runs the behavioral simulation in batch mode, returning the results without the waveform data. While **Optimized** can be faster, it returns less information than the Debug model.

For faster emulation without capturing this hardware debug information, select **Optimized**. As an example to debug system hang issues, use the Debug mode and look at the state of different signals in the Waveform viewer within the Vivado XSim simulator. Alternatively, if Debug is to be done purely on the application software, you can use the **Optimized** mode.

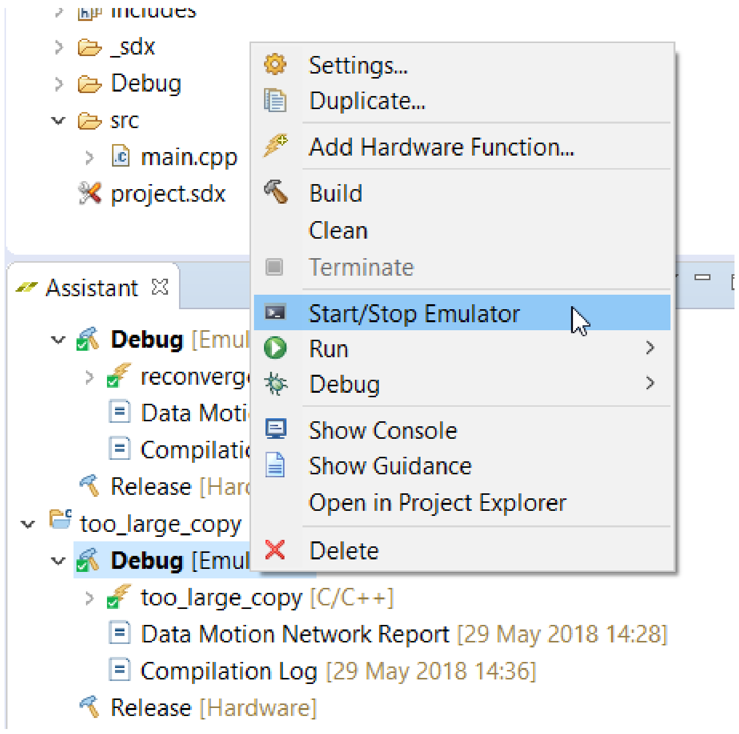
Running the Build

After specifying the **Generate emulation model** option, use the **Build** ( ) command to compile the system for emulation.

The Build process can take some time, depending on your application code, the size of your hardware functions, and the various options you have selected. To compile the hardware functions, the tool stack includes SDx, Vivado HLS, and the Vivado Design Suite.

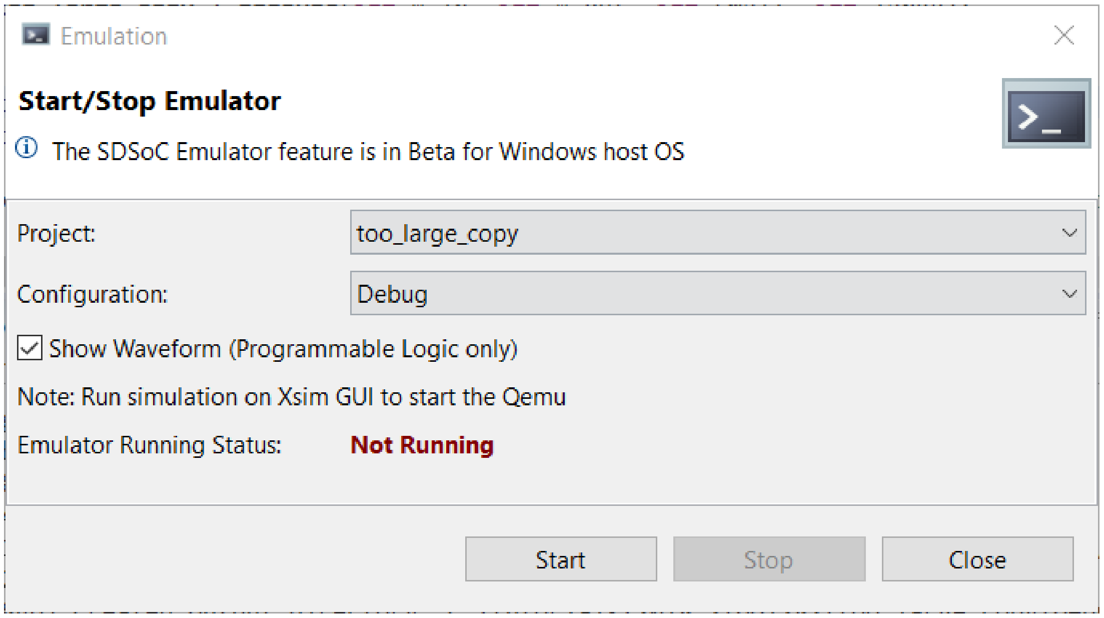
Invoking the System Emulator

After building the emulation target, you can invoke the system emulator using the **Xilinx→Start/Stop Emulator** menu command. Alternatively, you can also select the application in the **Assistant** panel, right-click and from the context menu select **Start/Stop Emulator**.



When the **Start/Stop Emulator** dialog box opens, the emulation mode is specified.

- If the emulation mode is **Debug**, you can choose to run the emulation with or without waveforms.
- If the emulation mode is **Optimized**, the **Show waveforms** checkbox is disabled, and cannot be changed.


Emulation

Start/Stop Emulator

ⓘ The SDSoC Emulator feature is in Beta for Windows host OS

Project: too_large_copy

Configuration: Debug

☑ Show Waveform (Programmable Logic only)

Note: Run simulation on Xsim GUI to start the Qemu

Emulator Running Status: Not Running

The **Start/Stop Emulator** dialog box displays the **Project** name, the Build **Configuration**, and has the **Show Waveform** option.

Disabling the **Show Waveform** option lets you run emulation with the output directed solely at the Emulation Console view, which shows all system messages including the results of any print statements in the source code. Some of these statements might include the values transferred to and from the hardware functions, or a statement that the application has completed successfully, which would verify that the source code running on the PS and the compiled hardware functions running in the PL are functionally correct.

Enabling the **Show Waveform** option provides the same functionality in the Console window, plus the behavioral simulation of the RTL, with a waveform window. The RTL waveform window allows you to see the value of any signal in the hardware functions over time. When using **Show Waveform**, you must manually add signals to the waveform window before starting the emulation.

1. Use the **Scopes** pane to navigate the design hierarchy.
2. Select the signals to monitor in the **Object** pane, and right-click to add the signals to the waveform pane.
3. Click the **Run All** toolbar button to start updates to the waveform window.

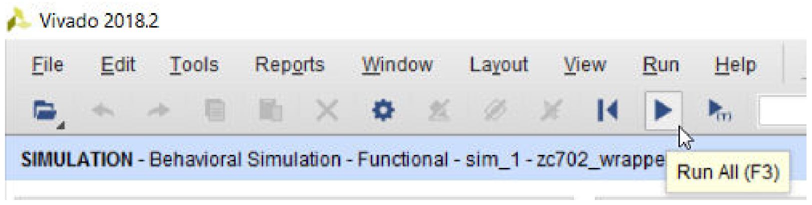
For more information about working with the Vivado® simulator waveform window, refer to the *Vivado Design Suite User Guide: Logic Simulation* (UG900).

Note: Running with RTL waveforms results in a slower run time, but enables detailed analysis into the operation of the hardware functions.

TIP: *You can also start the system emulation by selecting the active project in the Project Explorer view and right-clicking to select the* ***Run As → Launch on Emulator*** *menu command, or the* ***Debug As → Launch on Emulator*** *menu command. Launching the emulator from the* ***Debug As*** *menu causes the perspective change to the debug perspective to arrange the windows and views to facilitate debugging the project. See Working with SDx in the SDSoC Environment User Guide (UG1027) for more information on changing perspectives.*

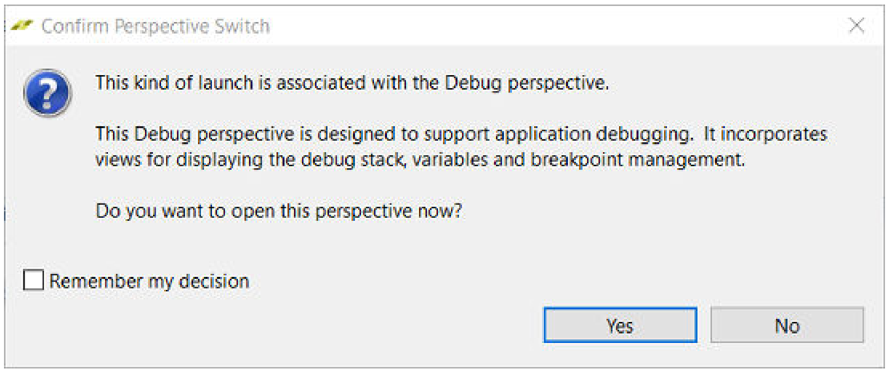
Viewing Emulation Output

After you invoke the system emulator, you see the program output in the console tab, and if the **Show Waveform** option was selected, Vivado® IDE is launched with the simulator running. Add waveforms to the Waveforms window as desired. Start simulation by clicking the **Run All** button.


Vivado 2018.2
File Edit Tools Reports Window Layout View Run Help
SIMULATION - Behavioral Simulation - Functional - sim_1 - zc702_wrappe
Run All (F3)

To launch a debug session with the emulator running, you can right-click on the application and from the context menu select **Debug As → Launch on Emulator (SDx Application Debugger)**.

This brings up the **Confirm Perspective Switch** dialog box. Click **Yes** to switch to the Debug perspective.


Confirm Perspective Switch

This kind of launch is associated with the Debug perspective.

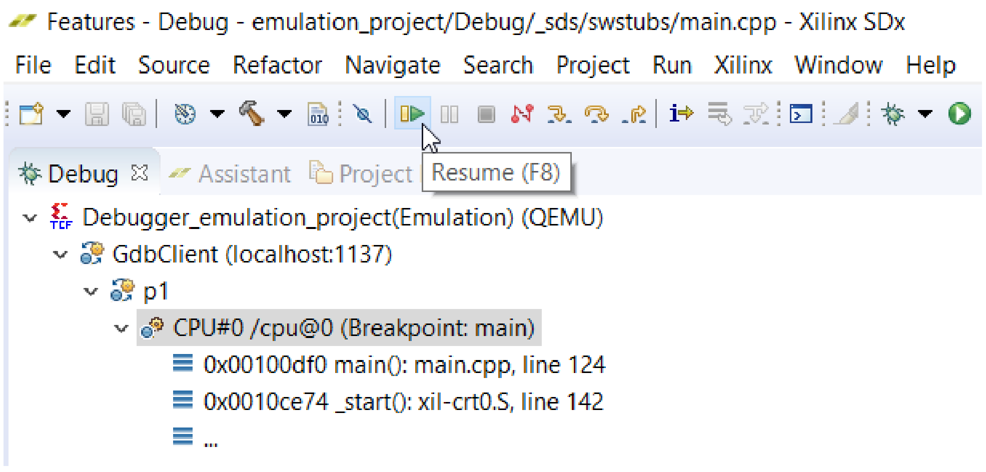
This Debug perspective is designed to support application debugging. It incorporates views for displaying the debug stack, variables and breakpoint management.

Do you want to open this perspective now?

☐ Remember my decision

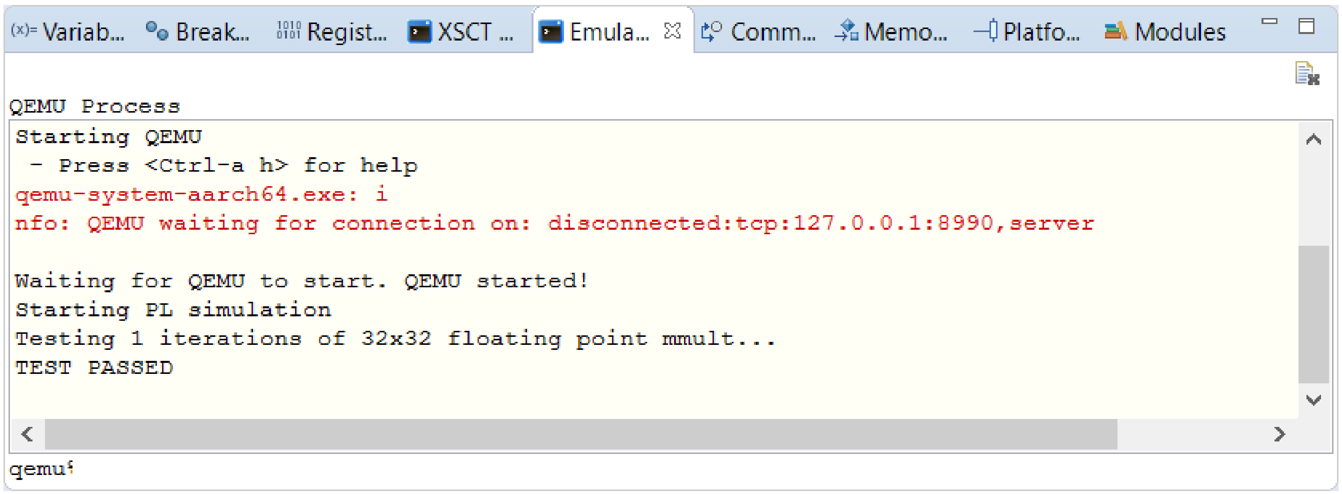
Yes No

The application is launched in the Debug perspective and the program execution is stopped at the main function. To resume the execution of the application code, click **Resume**.


Features - Debug - emulation_project/Debug/_sds/swstubs/main.cpp - Xilinx SDx
File Edit Source Refactor Navigate Search Project Run Xilinx Window Help
Debug
Assistant
Project
Resume (F8)

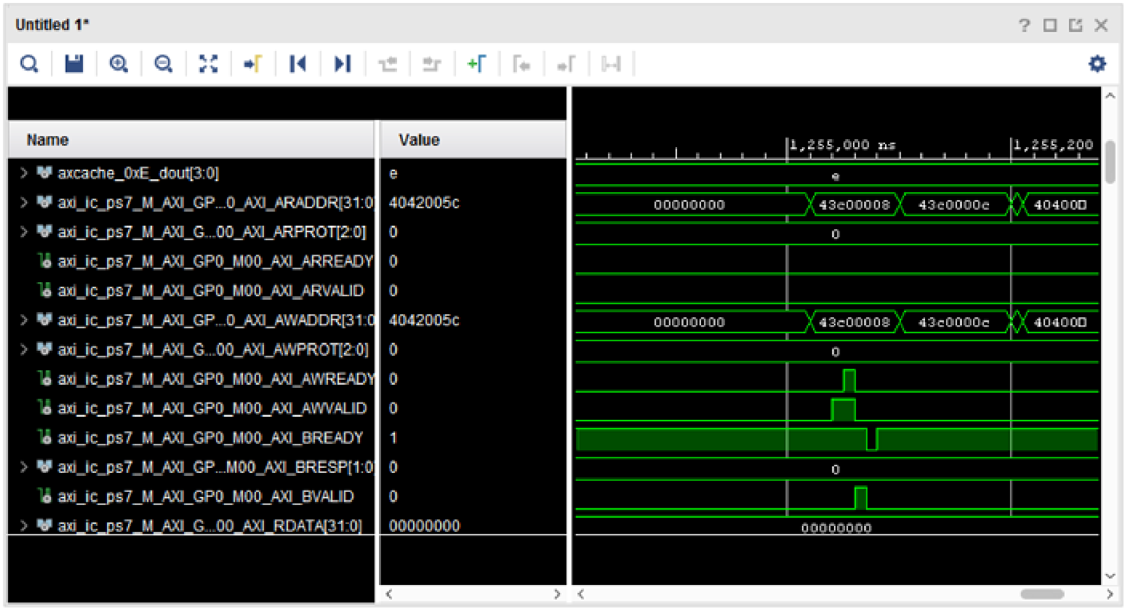
```
Debugger_emulation_project(Emulation) (QEMU)
  GdbClient (localhost:1137)
    p1
      CPU#0 /cpu@0 (Breakpoint: main)
        0x00100df0 main(): main.cpp, line 124
        0x0010ce74 _start(): xil-crt0.S, line 142
        ...
```

This starts execution of the application code. The output of the application code can be seen in the **Emulation Console** as shown below.



```
QEMU Process
Starting QEMU
 - Press <Ctrl-a h> for help
qemu-system-aarch64.exe: i
nfo: QEMU waiting for connection on: disconnected:tcp:127.0.0.1:8990,server

Waiting for QEMU to start. QEMU started!
Starting PL simulation
Testing 1 iterations of 32x32 floating point mmult...
TEST PASSED

qemu%
```

You can see the status of different signals in the Vivado Waveform window. You also see any appropriate response in the hardware functions in the RTL waveform. During any pause in the execution of the code, the RTL waveform window continues to execute and update, just like an FPGA running on the board.



Name	Value
axcache_0xE_dout[3:0]	e
axi_ic_ps7_M_AXI_GP...0_AXI_ARADDR[31:0	4042005c
axi_ic_ps7_M_AXI_G...00_AXI_ARPROT[2:0]	0
axi_ic_ps7_M_AXI_GP0_M00_AXI_ARREADY	0
axi_ic_ps7_M_AXI_GP0_M00_AXI_ARVALID	0
axi_ic_ps7_M_AXI_GP...0_AXI_AWADDR[31:0	4042005c
axi_ic_ps7_M_AXI_G...00_AXI_AWPROT[2:0]	0
axi_ic_ps7_M_AXI_GP0_M00_AXI_AWREADY	0
axi_ic_ps7_M_AXI_GP0_M00_AXI_AWVALID	0
axi_ic_ps7_M_AXI_GP0_M00_AXI_BREADY	1
axi_ic_ps7_M_AXI_GP...M00_AXI_BRESP[1:0	0
axi_ic_ps7_M_AXI_GP0_M00_AXI_BVALID	0
axi_ic_ps7_M_AXI_G...00_AXI_RDATA[31:0]	00000000

You can stop the emulation at any time using the menu option **Xilinx→Start/Stop Emulator** and selecting **Stop**.

TIP: *For an example project to demonstrate emulation, create a new SDx™ project using the Emulation Example template. The* `README.txt` *file in the project has a step-by-step guide for doing emulation on both the SDx GUI and the command line.*

Running System Emulation from the Command Line

You can create a design outside of the SDx™ IDE in a general command-line flow, using individual SDx commands to build and compile the project, or with a Makefile flow. In the following sample script, the `TARGET` flag defines that the compilation should be done for emulation.

```
# FPGA Board Platform (Default ~ zcu102)
PLATFORM := zcu102

# Run Target:
#   hw  - Compile for hardware
#   emu - Compile for emulation (Default)
TARGET := emu
```

The emulation mode, as shown in the sample script below, can be specified with one of two options:

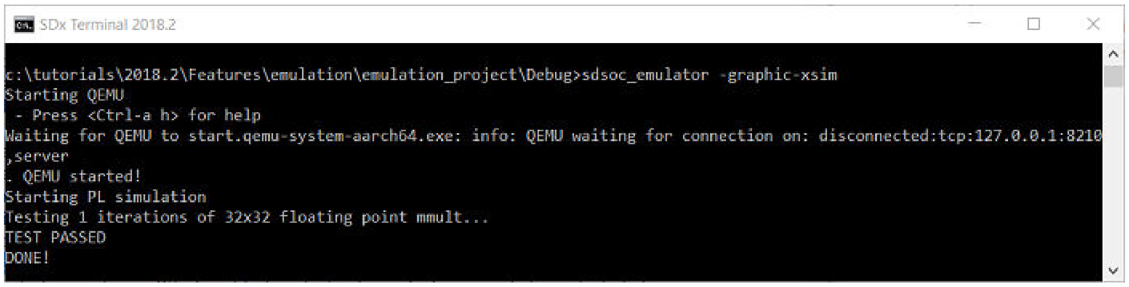
- `debug`: Captures waveform data from the PL hardware emulation for viewing and debugging.

- `optimized`: Provides faster emulation without capturing hardware debug information.

```
# Target OS:
#     linux (Default), standalone
TARGET_OS := linux

# Emulation Mode:
#     debug     - Include debug data
#     optimized - Exclude debug data (Default)
EMU_MODE := optimized
```

Type `make` to build the program at the command prompt.

If you want to view the waveform in the simulator, change directory to the level where you have the `_sds` directory, then type `sdsoc_emulator -graphic-xsim`. This starts the Vivado® Simulator, as shown below.



```
c:\tutorials\2018.2\Features\emulation\emulation_project\Debug>sdsoc_emulator -graphic-xsim
Starting QEMU
 - Press <Ctrl-a h> for help
Waiting for QEMU to start.qemu-system-aarch64.exe: info: QEMU waiting for connection on: disconnected:tcp:127.0.0.1:8210
,server
. QEMU started!
Starting PL simulation
Testing 1 iterations of 32x32 floating point mmult...
TEST PASSED
DONE!
```

Hardware Execution Features Available to All Platforms

Although system emulation is only available for application projects running on Xilinx provided SDSoC™ base platforms, the hardware execution flow is available to run on any platform that is the target of an SDSoC project. The hardware execution flow is simply the embedded processor operating system, the application code, and the hardware functions running in concert, as designed, on the hardware platform. The types of debugging you can perform on the hardware include the following:

1. Full software debug using the Xilinx® System Debugger
2. Hardware-Software co-debug using the Xilinx System Debugger

Please note that, hardware debug includes instrumenting the hardware for analyzing signals in the Vivado® Hardware Manager. The application needs to be built with special instructions to instrument the hardware for this.

Hardware Debugging in SDSoC Using ChipScope

After the final system image is generated and executed on the SDSoC™ platform, the entire system including the embedded processor OS, the application code, and the accelerated hardware functions, can be validated to be executing correctly on the actual hardware and any necessary debug activity can be performed.

This debugging step could reveal issues around connecting to the target platform, booting the processor, programming the hardware with the system image, problems with interactions between the application code and the hardware functions in the form of protocol violations, and validating multiple hardware functions with the application code. Finally this step could also reveal system performance metrics that could shift your focus from debug to performance tuning.

The SDx™ tool provides the capability to instrument the hardware to analyze transactions on the interfaces of the hardware accelerators and adapters. This provides users the ability to debug the hardware portion of the design. The following section illustrates this capability.

Using --dk to Enable Debugging the Accelerated Function

Visibility into a running design is crucial for debugging difficult situations like application hangs. The System ILA debug core provides transaction level visibility into an accelerated kernel or function running on hardware. AXI traffic of interest can also be captured and viewed using the System ILA core.

The ILA core can be instantiated in the overall hardware of an existing SDSoC™ design, to enable debugging features within that design, or can be inserted automatically by the compiler. The SDSCC compiler, also referred to as `sds++`, provides the `--dk` switch to attach System ILA cores at the interfaces to the hardware functions for debugging and performance monitoring purposes. Use the `--dk` option to enable ILA IP core insertion:

```
 --dk arg <[protocol|chipscope|
list_ports]:<compute_unit_name>:<interface_name>>
```

The following is an example of the `--dk` option in use:

```
sds++ -c --dk chipscope:vadd_cu0:s_axi_control --dk
chipscope:vadd_cu0:m_axi_gmem
```

A sample makefile to insert debug cores is shown below.

```
APPSOURCES = main.cpp mmult.cpp madd.cpp
EXECUTABLE = mmultadd.elf

PLATFORM = zc702
CLKID    =
DMCLKID  =
SDSFLAGS = -sds-pf ${PLATFORM} ${DMCLKID} \
    -sds-hw mmult mmult.cpp ${CLKID} -sds-end \
    -sds-hw madd madd.cpp ${CLKID} -sds-end \
    -debug-port mmult:A \
        -debug-port madd:C \
        --dk chipscope:madd_1:A \
        --dk chipscope:madd_1_if:ap_ctrl


CC = sds++ ${SDSFLAGS}

CFLAGS = -O3 -c
CFLAGS += -MMD -MP -MF"$(@:%.o=%.d)"
LFLAGS = -O3

OBJECTS := $(APPSOURCES:.cpp=.o)
DEPS := $(OBJECTS:.o=.d)

.PHONY: all clean ultraclean

all: ${EXECUTABLE}

${EXECUTABLE}: ${OBJECTS}
    ${CC} ${LFLAGS} $^ -o $@

-include ${DEPS}

%.o: %.cpp
    ${CC} ${CFLAGS} $^ -o $@

clean:
    ${RM} ${EXECUTABLE} ${OBJECTS} ${DEPS}

ultraclean: clean
    ${RM} ${EXECUTABLE}.bit
    ${RM} -rf _sds sd_card
```

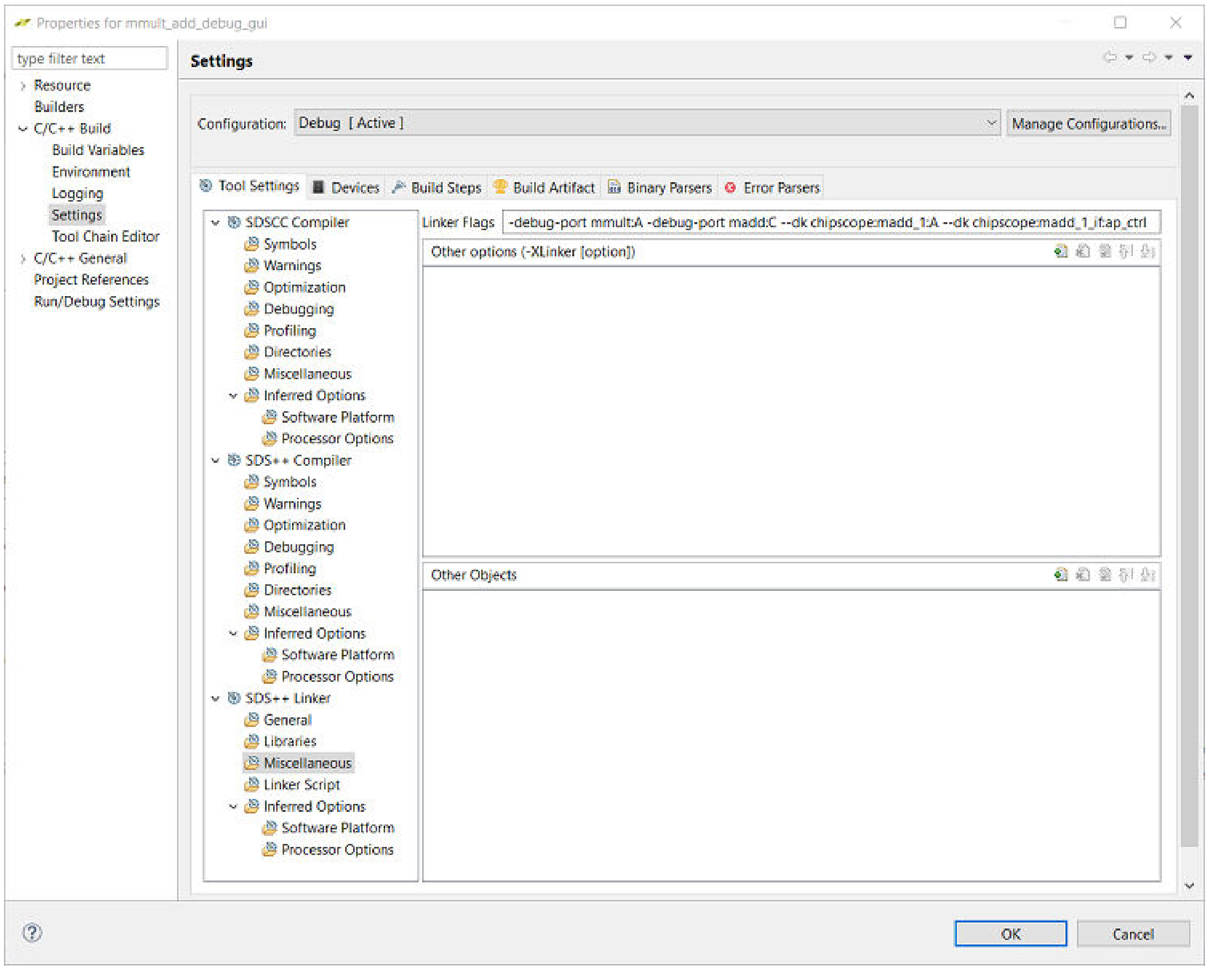
The `-debug-port` option specifies a function name and argument name. The lower level `--dk` option specifies the Tcl file used to recreate the block design instance and port name. For example:

- `-debug-port mmult:A` is equivalent to `--dk chipscope:mmult_1:A`, but the `sds++` command figures out what the instance and port names are in the Tcl file used to recreate the block design. `-debug-port` can only be used to insert the System ILA for accelerators.
- `--xp param:compiler.userPostSysLinkTcl=<user_tcl_file>`, where `<user_tcl_file>` contains IP integrator Tcl commands for advanced users who need to perform post-processing of the System ILA in the block diagram after system linking and before synthesis.
- `--dk` can be used to insert the System ILA for accelerator and adapter ports. You need to use this option to observe the adapter ports. Once the design is built, you can debug the design using the Hardware Manager as described in *Vivado Design Suite User Guide: Programming and Debugging* (UG908).

If working in the SDSoC GUI environment, the above mentioned flags can be added in the build settings as shown below.

1. Right-click on an application project and from the context menu choose, **C/C++ Build Settings**.
2. In the **Properties for <application_name>** dialog box, select **SDS++ Linker** → **Miscellaneous**. In the **Linker Flags** field, add the debug flags as needed.

Figure 8: **Properties dialog box for application**



Properties for mmult_add_debug_gui

Settings

Configuration: Debug [Active]

Linker Flags: -debug-port mmult:A -debug-port madd:C --dk chipscope:madd_1:A --dk chipscope:madd_1_if:ap_ctrl

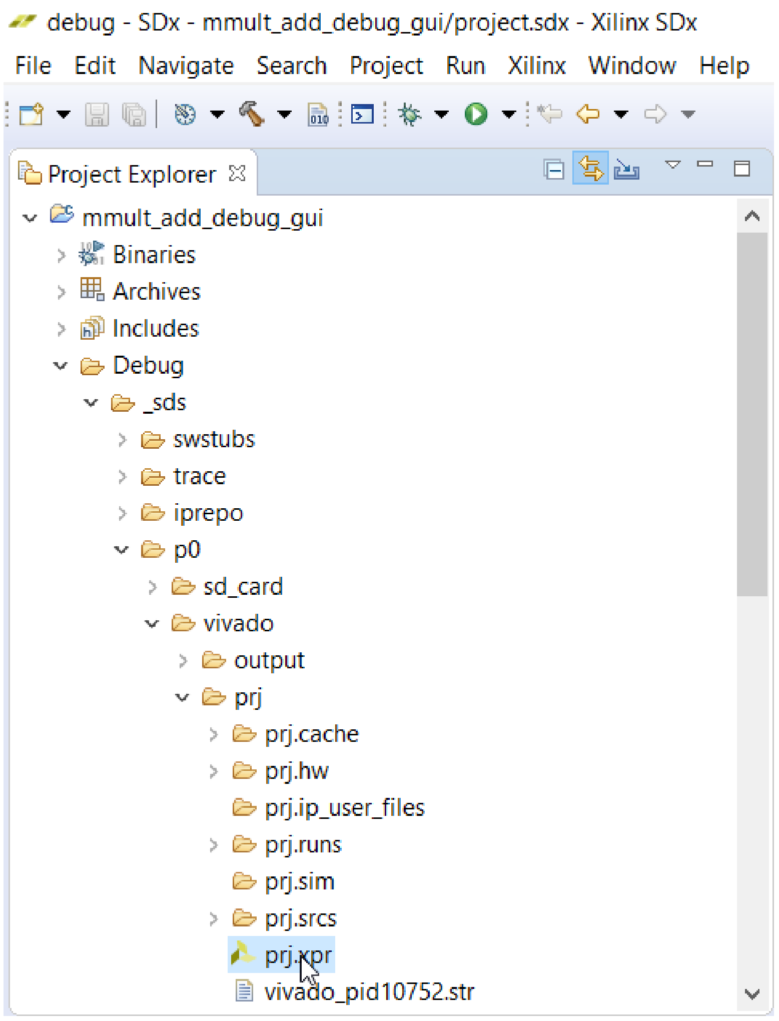
Refer to the *SDx Command and Utility Reference Guide* (UG1279) for more information.

Analyzing the hardware design

Once the design has been built with appropriate ILA instances, you can open and analyze the Vivado® design. To do this:

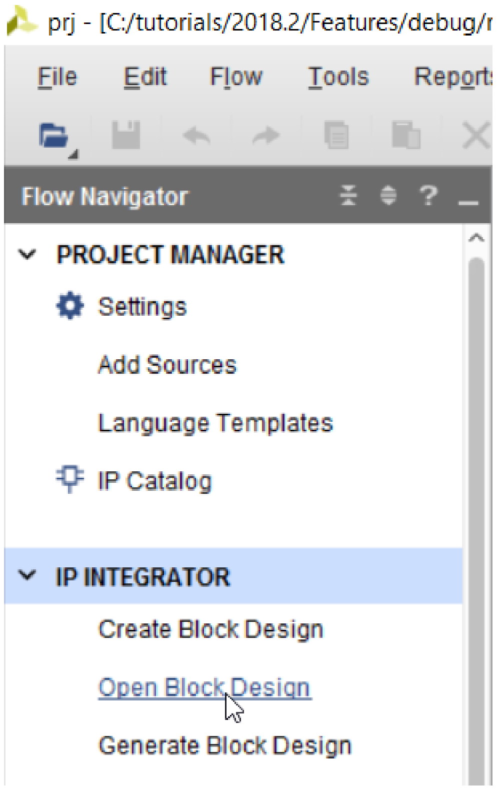
1. To confirm which signals are debuggable, navigate to the **Debug/Release** → **_sds** → **p0** → **vivado** → **prj** folder in the **Project Explorer** window.

Figure 9: **Opening the design in Vivado IDE**



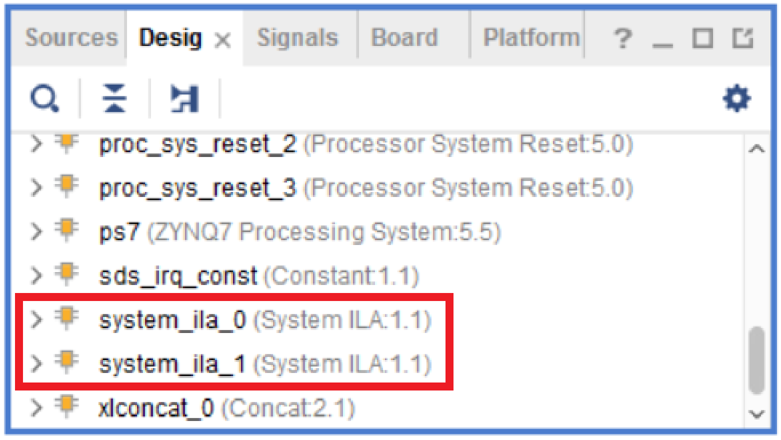
2. Double click `prj.xpr`. This opens the design in the Vivado IDE.
3. In Vivado IDE, click **Open Block Design** in the Flow Navigator under IP Integrator.

Figure 10: **Opening the Block Design in Vivado IDE**



4. In the **Designs** window, look for the instances of `system_ila_x`.

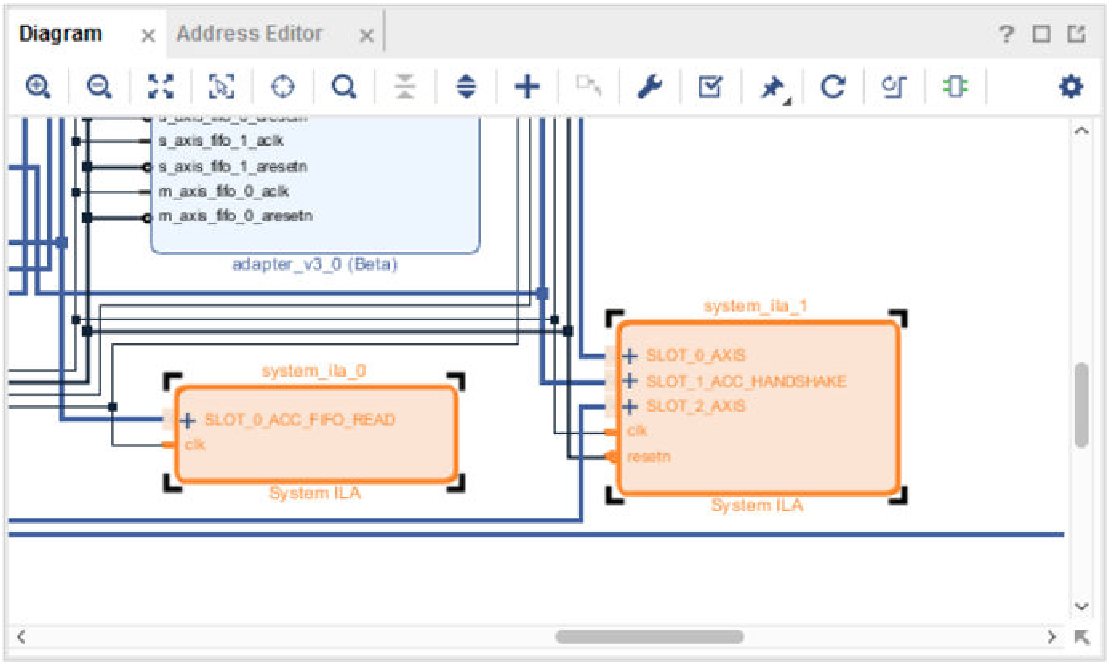
Figure 11: **Finding ILA Instances in the Design**



proc_sys_reset_2 (Processor System Reset:5.0)
proc_sys_reset_3 (Processor System Reset:5.0)
ps7 (ZYNQ7 Processing System:5.5)
sds_irq_const (Constant:1.1)
system_ila_0 (System ILA:1.1)
system_ila_1 (System ILA:1.1)
xlconcat_0 (Concat:2.1)

5. Selecting the ILA instance(s) in the **Design** window, highlights the instances in the block design.

Figure 12: **Highlighted Instances of ILA in the Block Design**



6. Select the interface nets connnected to the ILA and ensure that they have been connected to the interfaces specified in the SDx™ IDE.

Adding Debug IP to C-Callable IP

IMPORTANT!: *This debug technique requires familiarity with the Vivado® Design Suite, and RTL design.*

You need to instantiate debug cores like the Integrated Logic Analyzer (ILA) or Virtual Input/Output (VIO) into the RTL code of a C-Callable IP for debug purposes. From within the Vivado Design Suite, edit the RTL kernel to instantiate an ILA IP customization, or a VIO IP, into the RTL code – similar to using any other IP in the Vivado IDE. Refer to the *Vivado Design Suite User Guide: Programming and Debugging* (UG908) to learn more about using the ILA or other debug cores in the RTL Instantiation flow.

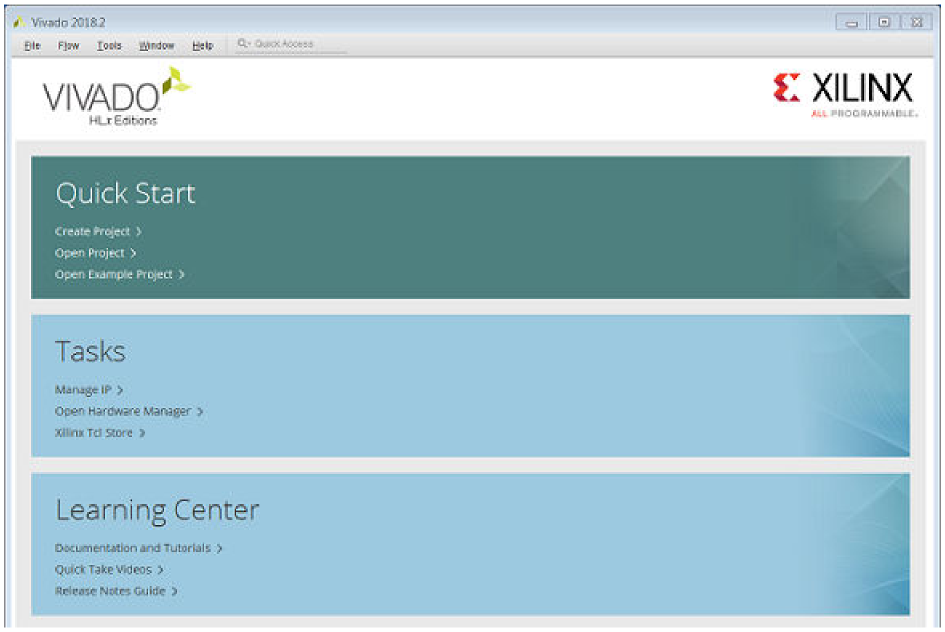
TIP: *The best time to add debug cores to your C-callable IP is when you create it. Refer to "Creating C-Callable IP Libraries" in the SDSoC Environment User Guide (UG1027) for more information.*

After the RTL kernel has been instrumented for debug with the appropriate debug cores, you can analyze the hardware in the ChipScope tool as described in *Vivado Design Suite User Guide: Programming and Debugging* (UG908).

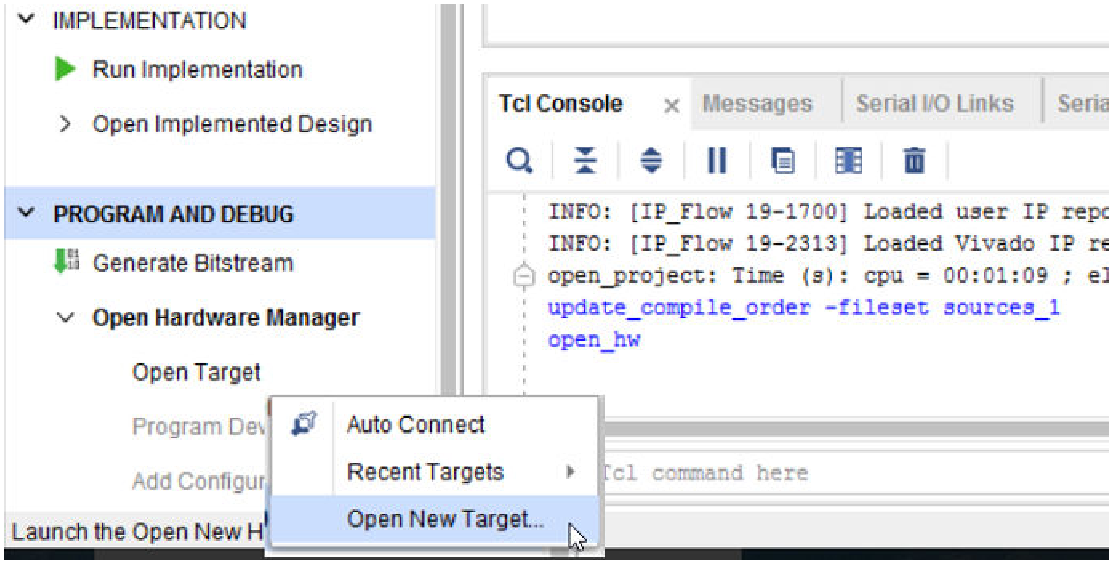
Debugging Designs using Vivado Hardware Manager

Once the SDx™ Application has been instrumented to insert debug cores, the next step is to connect to the Vivado® Hardware Manager and look at Integrated Logic Analyzer (ILA) core transactions. To connect to the target board using the Vivado Hardware Manager, perform the following steps.

1. Launch the Vivado Design Suite.
2. Select **Open Hardware Manager** from the **Tasks** menu, as shown in the following figure.

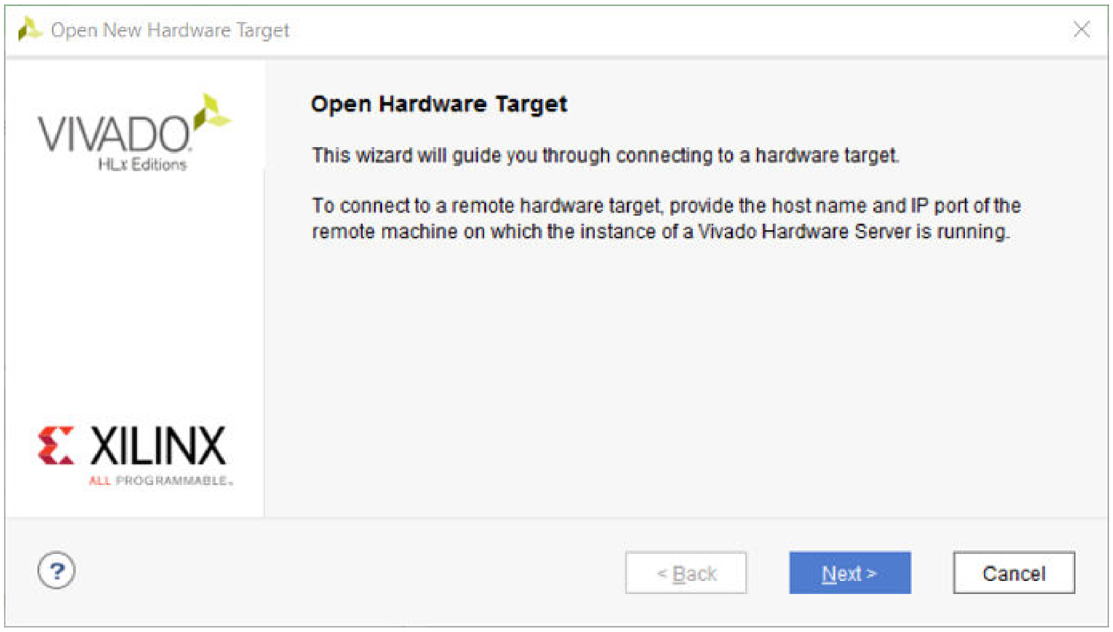


Alternatively, you can also launch Vivado, open the Vivado project, under `<appication_project_name>/Debug/_sds/p0/vivado/prj/prj.xpr` and then from the **Flow Navigator**, click **Program and Debug** → **Open Hardware Manager** → **Open Target** → **Open New Target**.



```
INFO: [IP_Flow 19-1700] Loaded user IP repo
INFO: [IP_Flow 19-2313] Loaded Vivado IP re
open_project: Time (s): cpu = 00:01:09 ; el
update_compile_order -fileset sources_1
open_hw
```

3. The **Open New Hardware Target** wizard opens. Click **Next**.

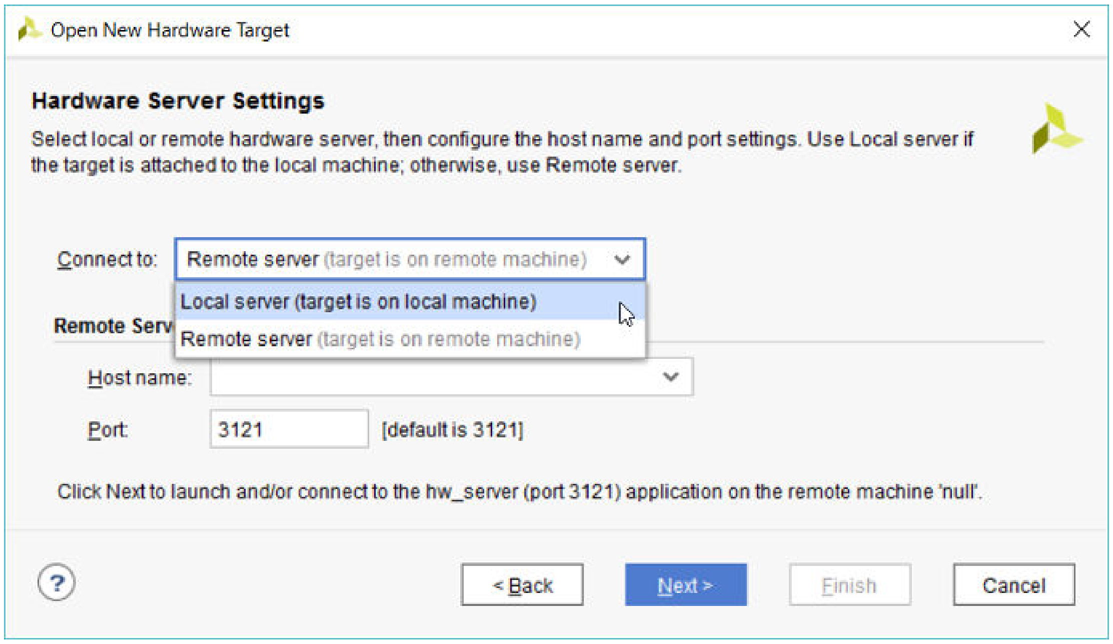


Open Hardware Target

This wizard will guide you through connecting to a hardware target.

To connect to a remote hardware target, provide the host name and IP port of the remote machine on which the instance of a Vivado Hardware Server is running.

4. In the **Hardware Server Settings** page connect to the correct target by clicking on the **Connect to** pull-down menu and selecting either **Remote Server** or **Local Server**. In case you select the **Remote Server** option, you will need to add a **Host name** and the correct **Port** number. In this example, we have assumed that you are connected locally.


Open New Hardware Target

Hardware Server Settings

Select local or remote hardware server, then configure the host name and port settings. Use Local server if the target is attached to the local machine; otherwise, use Remote server.

Connect to: Remote server (target is on remote machine)

Local server (target is on local machine)

Remote server (target is on remote machine)

Remote Serv

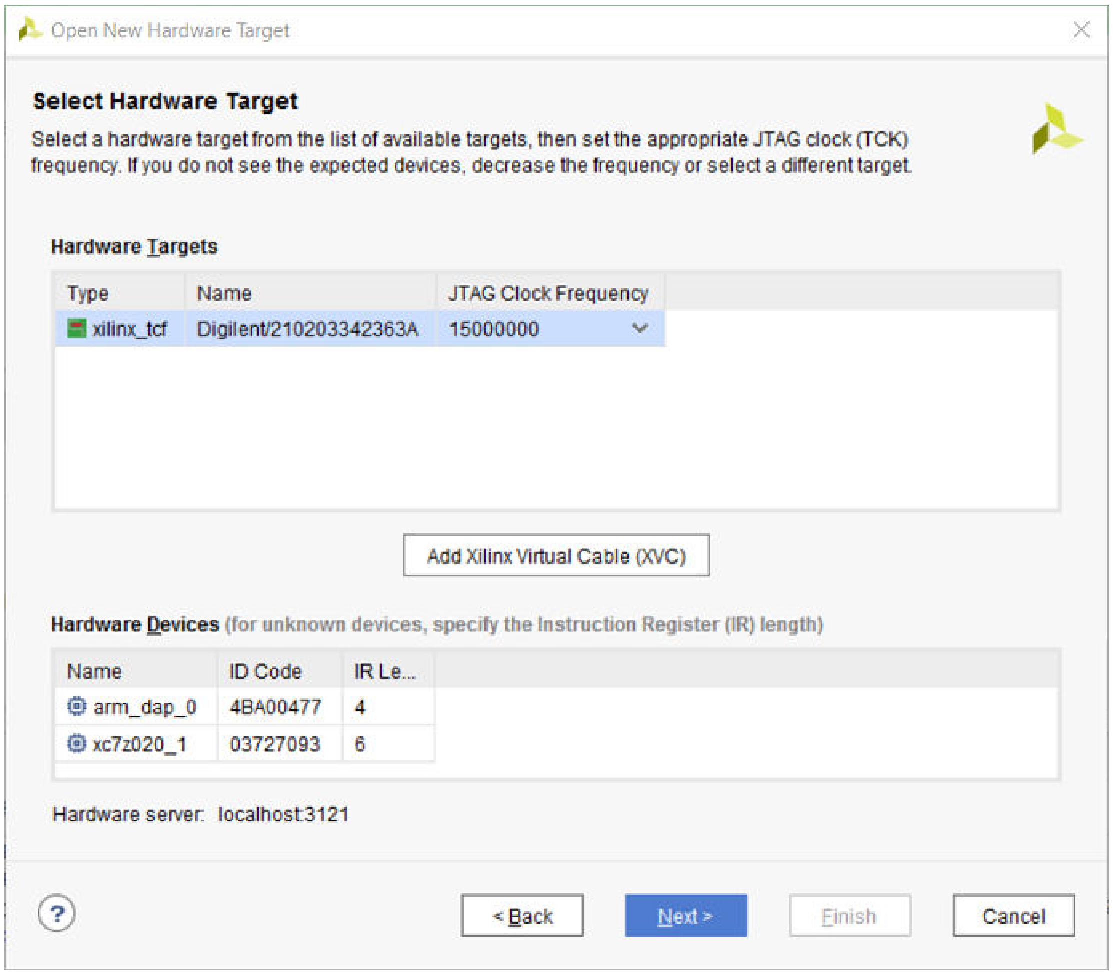
Host name:

Port: 3121 [default is 3121]

Click Next to launch and/or connect to the hw_server (port 3121) application on the remote machine 'null'.

Click **Next**.

5. The **Select Hardware Target** page opens which identifies the target(s) present on the board.



Open New Hardware Target

Select Hardware Target

Select a hardware target from the list of available targets, then set the appropriate JTAG clock (TCK) frequency. If you do not see the expected devices, decrease the frequency or select a different target.

Hardware Targets

Type	Name	JTAG Clock Frequency
xilinx_tcf	Digilent/210203342363A	15000000

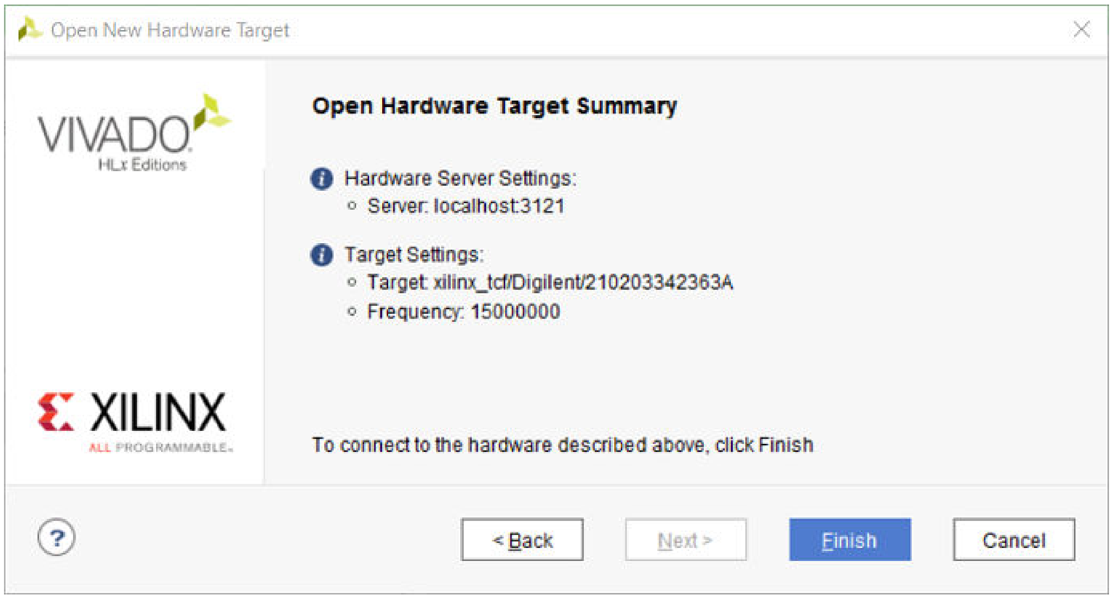
Hardware Devices (for unknown devices, specify the Instruction Register (IR) length)

Name	ID Code	IR Le...
arm_dap_0	4BA00477	4
xc7z020_1	03727093	6

Hardware server: localhost:3121

Click **Next**.

6. The **Open Hardware Target Summary** page opens which summarizes the Server name, the port to which it is connected to and also the correct target and the operating frequency.



Open New Hardware Target

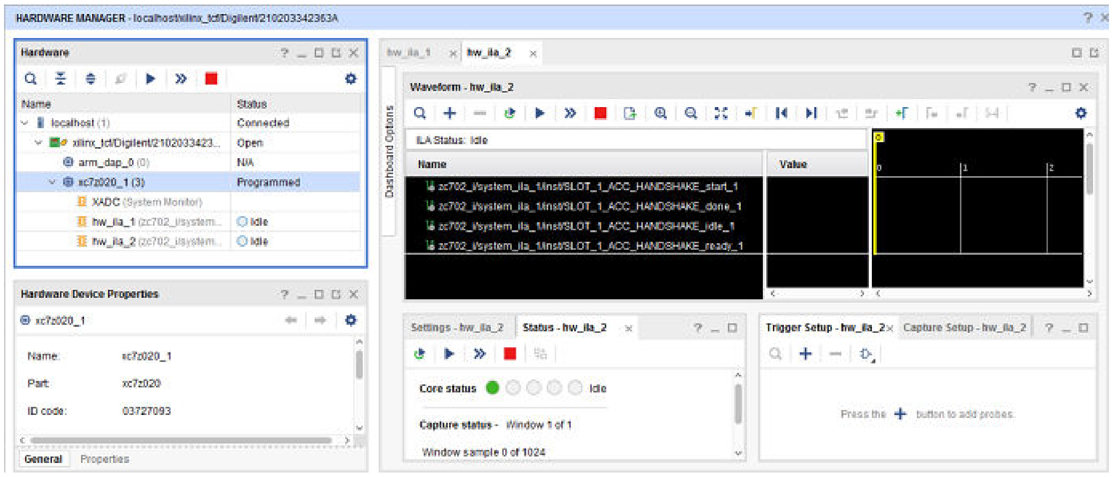
Open Hardware Target Summary

- Hardware Server Settings:
  - Server: localhost:3121
- Target Settings:
  - Target: xilinx_tcf/Digilent/210203342363A
  - Frequency: 15000000

To connect to the hardware described above, click Finish

Click **Finish**.

7. The Hardware Manager window opens as shown below.



HARDWARE MANAGER - localhost/xilinx_tcf/Digilent/210203342363A

Hardware

Name	Status
localhost (1)	Connected
xilinx_tcf/Digilent/2102033423...	Open
arm_dap_0 (0)	N/A
xc7z020_1 (3)	Programmed
XADC (System Monitor)	
hw_ila_1 (zc702_i/system...	Idle
hw_ila_2 (zc702_i/system...	Idle

Hardware Device Properties

xc7z020_1

Name:	xc7z020_1
Part:	xc7z020
ID code:	03727093

Waveform - hw_ila_2

ILA Status: Idle

Name	Value
zc702_i/system_ila_1/inst/SLOT_1_ACC_HANDSHAKE_start_1	
zc702_i/system_ila_1/inst/SLOT_1_ACC_HANDSHAKE_done_1	
zc702_i/system_ila_1/inst/SLOT_1_ACC_HANDSHAKE_idle_1	
zc702_i/system_ila_1/inst/SLOT_1_ACC_HANDSHAKE_ready_1	

Status - hw_ila_2

Core status ● ○ ○ ○ ○ Idle

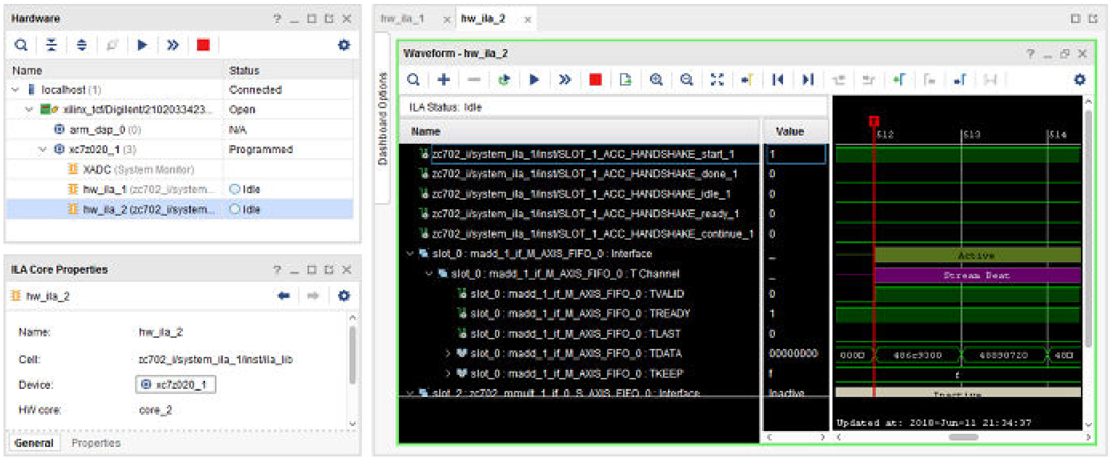
Capture status - Window 1 of 1

Window sample 0 of 1024

Trigger Setup - hw_ila_2

Press the + button to add probes.

8. The ChipScope tool can now be used to debug the interfaces where the Integrated Logic Analyzer was connected to running on the SDSoC platform. Refer to the *Vivado Design Suite User Guide: Programming and Debugging* (UG908) for more information on working with the tool.



Name	Status
localhost (1)	Connected
xilinx_tcf/Digilent/2102033423...	Open
arm_dap_0 (0)	N/A
xc7z020_1 (3)	Programmed
XADC (System Monitor)	
hw_ila_1 (zc702_i/system...	Idle
hw_ila_2 (zc702_i/system...	Idle

hw_ila_2

Name:	hw_ila_2
Cell:	zc702_i/system_ila_1/inst/ila_lib
Device:	xc7z020_1
HW core:	core_2

ILA Status: Idle

Name	Value
zc702_i/system_ila_1/inst/SLOT_1_ACC_HANDSHAKE_start_1	1
zc702_i/system_ila_1/inst/SLOT_1_ACC_HANDSHAKE_done_1	0
zc702_i/system_ila_1/inst/SLOT_1_ACC_HANDSHAKE_idle_1	0
zc702_i/system_ila_1/inst/SLOT_1_ACC_HANDSHAKE_ready_1	0
zc702_i/system_ila_1/inst/SLOT_1_ACC_HANDSHAKE_continue_1	0
slot_0 : madd_1_if_M_AXIS_FIFO_0 : Interface	_
slot_0 : madd_1_if_M_AXIS_FIFO_0 : T Channel	_
slot_0 : madd_1_if_M_AXIS_FIFO_0 : TVALID	0
slot_0 : madd_1_if_M_AXIS_FIFO_0 : TREADY	1
slot_0 : madd_1_if_M_AXIS_FIFO_0 : TLAST	0
slot_0 : madd_1_if_M_AXIS_FIFO_0 : TDATA	00000000
slot_0 : madd_1_if_M_AXIS_FIFO_0 : TKEEP	f
slot_2 : zc702_mmult_1_if_0_S_AXIS_FIFO_0 : Interface	Inactive

```
0000   406c9300   40890720   400
f
Updated at: 2018-Jun-11 21:34:37
```

Hardware/Software Event Tracing

Event tracing provides visibility into each phase of the hardware function execution, including the software setup for the accelerators and data transfers, as well as the hardware execution of the accelerators and data transfers. Tracing an application produces a log that records correlation between events for a duration of time. The goal of tracing is to help debug execution by observing what happened when, and how long events took.

Software event tracing automatically instruments the stub of the hardware function to capture software control events associated with a hardware function call. The event types that are recorded include the setup and initialization of the hardware accelerator, data transfers, and hardware-software synchronization events.

Hardware event tracing of accelerators with data transfers over AXI4-Stream connections can also be enabled through the use of the `-trace` option of the `sdscc/sds++` compilers . When the linker is invoked with the `-trace` option, it inserts hardware monitor IP cores into the RTL implementation of the hardware function to track the accelerator start and stop, and the duration of data transfers.

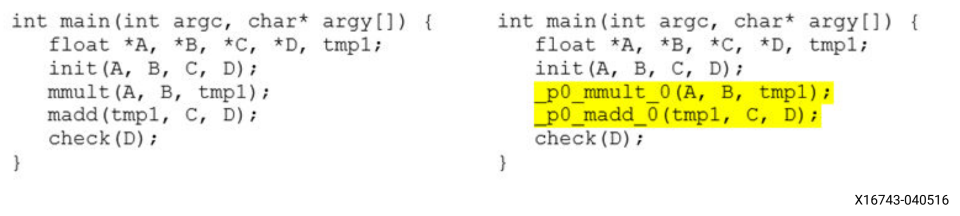
As with hardware debugging, event tracing requires you to connect the SDSoC platform to a host computer as described in Connecting to the Hardware. To run event tracing, execute the application using the SDSoC GUI from the host using a debug or release build configuration.

Hardware/Software System Runtime Operation

The SDSoC™ compilers implement hardware functions either by cross-compiling them into IP using the Vivado® HLS tool, or by linking them as C-Callable IP as described in the *SDSoC Environment Platform Development Guide* (UG1146).

Each hardware function callsite is rewritten to call a stub function that manages the execution of the hardware accelerator. The figure below shows an example of hardware function rewriting. The original user code is shown on the left. The code section on the right shows the hardware function calls rewritten with new function names.

Figure 13: **Hardware Function Call Site Rewriting**



```
int main(int argc, char* argy[]) {
   float *A, *B, *C, *D, tmp1;
   init(A, B, C, D);
   mmult(A, B, tmp1);
   madd(tmp1, C, D);
   check(D);
}
```

```
int main(int argc, char* argy[]) {
   float *A, *B, *C, *D, tmp1;
   init(A, B, C, D);
   _p0_mmult_0(A, B, tmp1);
   _p0_madd_0(tmp1, C, D);
   check(D);
}
```

X16743-040516

The stub function initializes the hardware accelerator, initiates any required data transfers for the function arguments, and then synchronizes hardware and software by waiting at an appropriate point in the program for the accelerator and all associated data transfers to complete. If, for example, the hardware function `foo()` is defined in `foo.cpp`, you can view the generated rewritten code in `_sds/swstubs/foo.cpp` for the project build configuration. As an example, the stub code below replaces a user function marked for hardware. This function starts the accelerator, starts data transfers to and from the accelerator, and waits for those transfers to complete.

```
void _p0_mmult0(float *A, float *B, float *C) {
   switch_to_next_partition(0);
   int start_seq[3];
   start_seq[0] = 0x00000f00;
   start_seq[1] = 0x00010100;
   start_seq[2] = 0x00020000;
   cf_send_i(cmd_addr,start_seq,cmd_handle);
   cf_wait(cmd_handle);
   cf_send_i(A_addr, A, A_handle);
   cf_send_i(B_addr, B, B_handle);
   cf_receive_i(C_addr, C, C_handle);
   cf_wait(A_handle);
   cf_wait(B_handle);
   cf_wait(C_handle);
```

Event tracing provides visibility into each phase of the hardware function execution, including the software setup for the accelerators and data transfers, as well as the hardware execution of the accelerators and data transfers. For example, the stub code below is instrumented for trace. Each command that starts the accelerator, starts a transfer, or waits for a transfer to complete is instrumented.

```
void_p0_mmult_0(float *A, float *B, float *C) {
   switch_to_next_partition(0);
   int start_seq[3];
   start_seq[0] = 0x00000f00;
   start_seq[1] = 0x00010100;
   start_seq[2] = 0x00020000;
```

```
    sds_trace(EVENT_START);
    cf_send_i(cmd_addr,start_seq,cmd_handle);
    sds_trace(EVENT_STOP);
    sds_trace(EVENT_START);
    cf_wait(cmd_handle);
    sds_trace(EVENT_STOP);
    sds_trace(EVENT_START);
    cf_send_i(A_addr, A, A_handle);
    sds_trace(EVENT_STOP);
    sds_trace(EVENT_START);
    cf_send_i(B_addr, B, B_handle);
    sds_trace(EVENT_STOP);
    sds_trace(EVENT_START);
    cf_receive_i(C_addr, C, C_handle);
    sds_trace(EVENT_STOP);
    sds_trace(EVENT_START);
    cf_wait(A_handle);
    sds_trace(EVENT_STOP);
    sds_trace(EVENT_START);
    cf_wait(B_handle);
    sds_trace(EVENT_STOP);
    sds_trace(EVENT_START);
    cf_wait(C_handle);
    sds_trace(EVENT_STOP);
```

Software Tracing

Event tracing automatically instruments the stub function to capture software control events associated with the implementation of a hardware function call. The event types include the following.

- Accelerator set up and initiation
- Data transfer setup
- Hardware/software synchronization barriers (“wait for event”)

Each of these events is independently traced, and results in a single AXI-Lite write into the programmable logic, where it receives a timestamp from the same global timer as hardware events.

Hardware Tracing

The SDSoC™ Environment supports hardware event tracing of accelerators cross-compiled using Vivado® HLS, and data transfers over AXI4-Stream connections. When `sds++` is invoked with the `-trace` option, it automatically inserts hardware monitor IP cores into the generated system to log these event types:

- Accelerator start and stop, defined by `ap_start` and `ap_done` signals.
- Data transfer start and stop, defined by AXI4-Stream handshake and `TLAST` signals.

Each of these events is independently monitored and receives a timestamp from the same global timer used for software events. If the hardware function explicitly declares an AXI4-Lite control interface using the following pragma, it cannot be traced because its `ap_start` and `ap_done` signals are not part of the IP interface:

```
#pragma HLS interface s_axilite port=foo
```

These debug cores will use some hardware resources, less than 0.1% of the hardware resources available on a ZC706 board.

The AXI4-Stream monitor core has two modes: basic and statistics. The basic mode does just the start/stop trace event generation. The statistics mode enables an AXI4-Lite interface to two 32-bit registers. The register at offset 0x0 presents the word count of the current, on-going transfer. The register at offset 0x4 presents the word count of the previous transfer. As soon as a transfer is complete, the current count is moved to the previous register. By default, the AXI4-Stream core is configured in the basic mode.

In addition to the hardware trace monitor cores, the output trace event signals are combined by a single integration core. This core has a parameterizeable number of ports (from 1–63), and can thus support up to 63 individual monitor cores (either accelerator or AXI4-Stream). The resource utilization of this core depends on the number of ports enabled, and thus the number of monitor cores inserted.

On a ZC706 platform, this can use between roughly 0.1-1.0 percent of the available hardware resources, and up to approximately 10% of the memories with the integration logic.

Implementation Flow

During the implementation flow, when tracing is enabled, tracing instrumentation is inserted into the software code and hardware monitors are inserted into the hardware system automatically. The hardware system (including the monitor cores) is then synthesized and implemented, producing the bitstream. The software tracing is compiled into the regular user program.

Hardware and software traces are timestamped in hardware and collected into a single trace stream that is buffered up in the programmable logic.

Debug Techniques

This chapter describes the different styles of debugging techniques applicable to SDSoC™ designs. It highlights different approaches for software-based debugging and hardware oriented techniques. In the software based approaches, the user is not required to actually understand the implementation of the design in the FPGA. However, this concept can only be extended to a certain degree at which point, the user will need to perform a more hardware based detailed analysis.

When debugging SDSoC applications, you can use the same methods and techniques as applications used for debugging standard C/C++. Most SDSoC applications consist of specific functions tagged for hardware acceleration and surrounded by standard C/C++ code.

When debugging an SDSoC application with a board attached to the debug Host machine, you can right-click a project and select the **Debug As → Launch on Hardware** option to begin a debug session.

Options different from the default settings can be set through the **Debug As → Debug Configurations** selection. As the debug environment is initialized Xilinx recommends that users switch to the Debug perspective when prompted by the SDx™ IDE. The debug perspective view provides the ability to debug the standard C/C++ portions of the application, by single-stepping code, setting and removing breakpoints, displaying variables, dumping registers, viewing memory, and controlling the code flow with "run until" and "jump to" type debugging directives. Inputs and outputs can be observed pre- and post- function call to determine correct behavior.

You can determine if a hardware accelerated application meets its real-time requirements by placing debug statements to start and stop a counter just before and just after a hardware accelerated function. The SDx environment provides the `sds_clock_counter()` function which is typically used to calculate the elapsed time for a hardware accelerated function.

You can also perform debugging without a target board connected to the debug host by building the SDx project for emulation. During emulation, you can control and observe the software and data just as before through the debug perspective view, but you can also view the hardware accelerated functions through a Vivado® simulator waveform viewer. You can observe accelerator signaling for conditions such as Accelerator start, Accelerator done and monitor data buses for inputs and outputs. Building a project for emulation also avoids a possibly long Vivado implementation step to generate an FPGA bitstream.

See the *SDSoC Environment Debugging Guide* (UG1282) for information on using the interactive debuggers in the SDx IDE.

Debugging System Hangs and Runtime Errors

Programs compiled using `sds++` can be debugged using the standard debuggers supplied with the SDSoC™ environment or Vivado® SDK. Typical runtime errors are incorrect results, premature program exits, and program "hangs." The first two kinds of errors are familiar to C/C++ programmers, and can be debugged by stepping through the code using a debugger.

Note: Applications might hang when you are running on the board. Hangs commonly happen due to a mismatch on data size between the producer and the consumer.

A program hang is a runtime error caused by specifying an incorrect amount of data to be transferred across a streaming connection created using `#pragma SDS data access_pattern(A:SEQUENTIAL)`, by specifying a streaming interface in a synthesizeable function within Vivado HLS, or by a C-Callable hardware function in a pre-built library that has streaming hardware interfaces. A program hangs when the consumer of a stream is waiting for more data from the producer but the producer has stopped sending data.

Consider the following code fragment that results in streaming input/output from a hardware function.

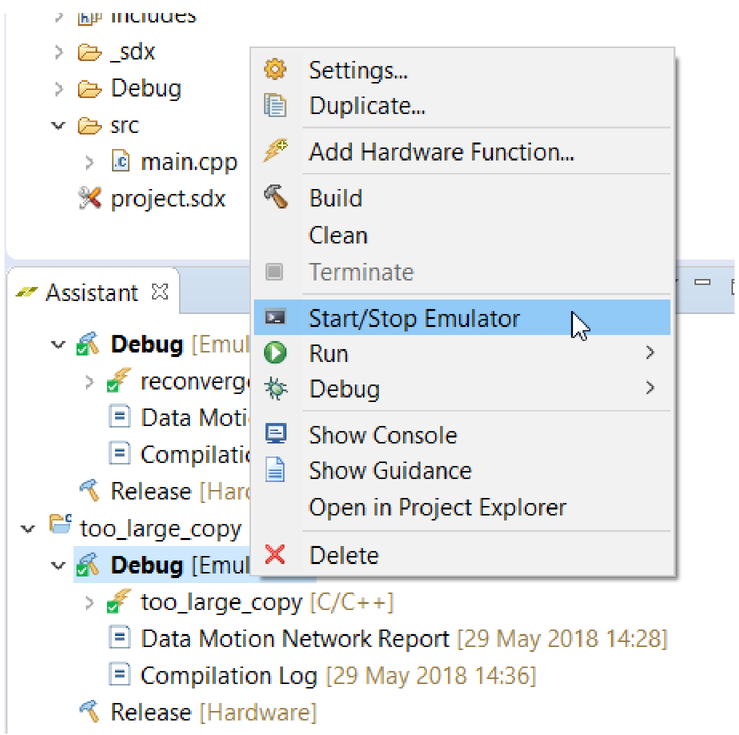
```
#pragma SDS data access_pattern(in_a:SEQENTIAL, out_b:SEQUENTIAL)
void f1(int in_a[20], int out_b[20]);      // declaration

void f1(int in_a[20], int out_b[20]) {     // definition
   int i;
   for (i=0; i < 19; i++) {
       out_b[i] = in_a[i];
   }
}
```

Notice that the loop reads the `in_a` stream 19 times but the size of `in_a[]` is 20, so the caller of `f1` would wait forever (or hang) if it waited for `f1` to consume all the data that was streamed to it. Similarly, the caller would wait forever if it waited for `f1` to send 20 int values because `f1` sends only 19. Program errors that lead to such "hangs" can be detected by using system emulation to review whether the data signals are static (review the associated protocol signals `TLAST`, `ap_ready`, `ap_done`, `TREADY`, etc.) or by instrumenting the code to flag streaming access errors such as non-sequential access or incorrect access counts within a function and running in software. Streaming access issues are typically flagged as `improper streaming` `access` warnings in the log file, and it is left to the user to determine if these are actual errors. Running your application on the SDSoC emulator is a good way to gain visibility of data transfers with a debugger. You will be able to see where in software the system is hanging (often within a `cf_wait()` call), and can then inspect associated data transfers in the simulation waveform view, which gives you access to signals on the hardware blocks associated with the data transfer.

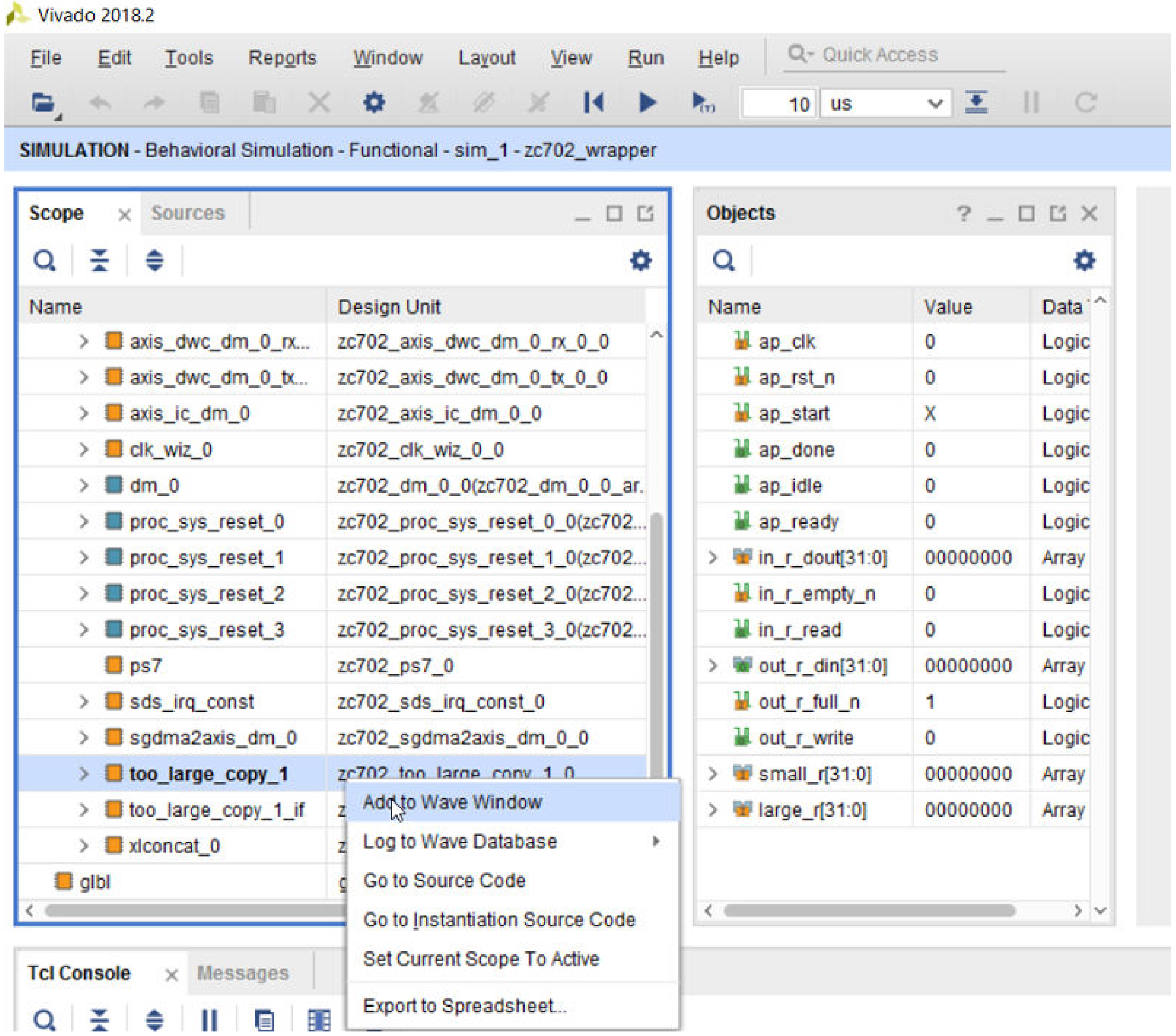
As another example, consider the following code that results in streaming input/output from the hardware function.

```
#pragma SDS data access_pattern(in:SEQUENTIAL, out:SEQUENTIAL)
#pragma SDS data copy(in[0:large], out[0:small])
void too_large_copy(int* in, int* out, int small, int large)
{
    for(int i = 0; i < small; i++) {out[i] = in[i];}
}


int main()
{
    int* temp_var1 = new int[1024 * 1024];
    int* temp_var2 = new int[1024 * 1024];

    too_large_copy(temp_var1, temp_var2, 1024, 1024 * 1024); //hangs
because the input DMA continues to try to feed data to a halted HLS core

}
```

In this case, the DMA continues to try to send data to the hardware function, whereas the hardware function is already done and is not accepting any data. This results in a system "hang". To debug this type of issue, build the code for emulation on the base platform. Once the application is compiled, start the emulator by selecting **Xilinx**→**Start/Stop Emulator**. Alternatively, you can start the emulator from the **Assistant** window as shown below. Right-click the **Active build configuration** for the application and select **Start/Stop Emulator**.



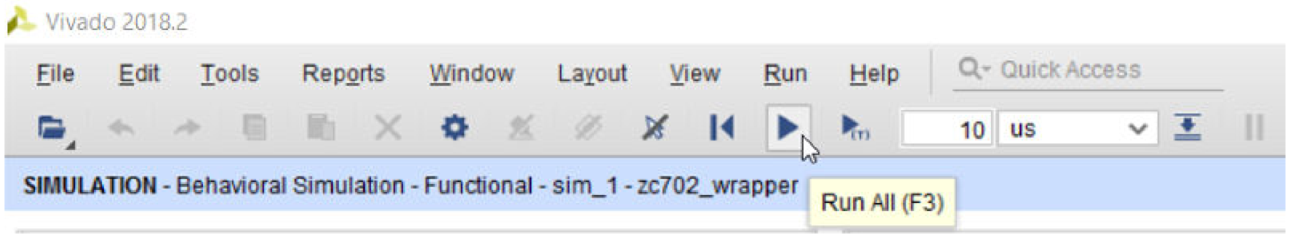
In the **Emulation** dialog box ensure that the **Show Waveform (Programmable Logic only)** checkbox is checked. This brings up the Vivado Simulator where the state of different interfaces can be viewed in the Waveform window. To monitor the interfaces of the hardware function, right-click on the function and select **Add to Wave window**.



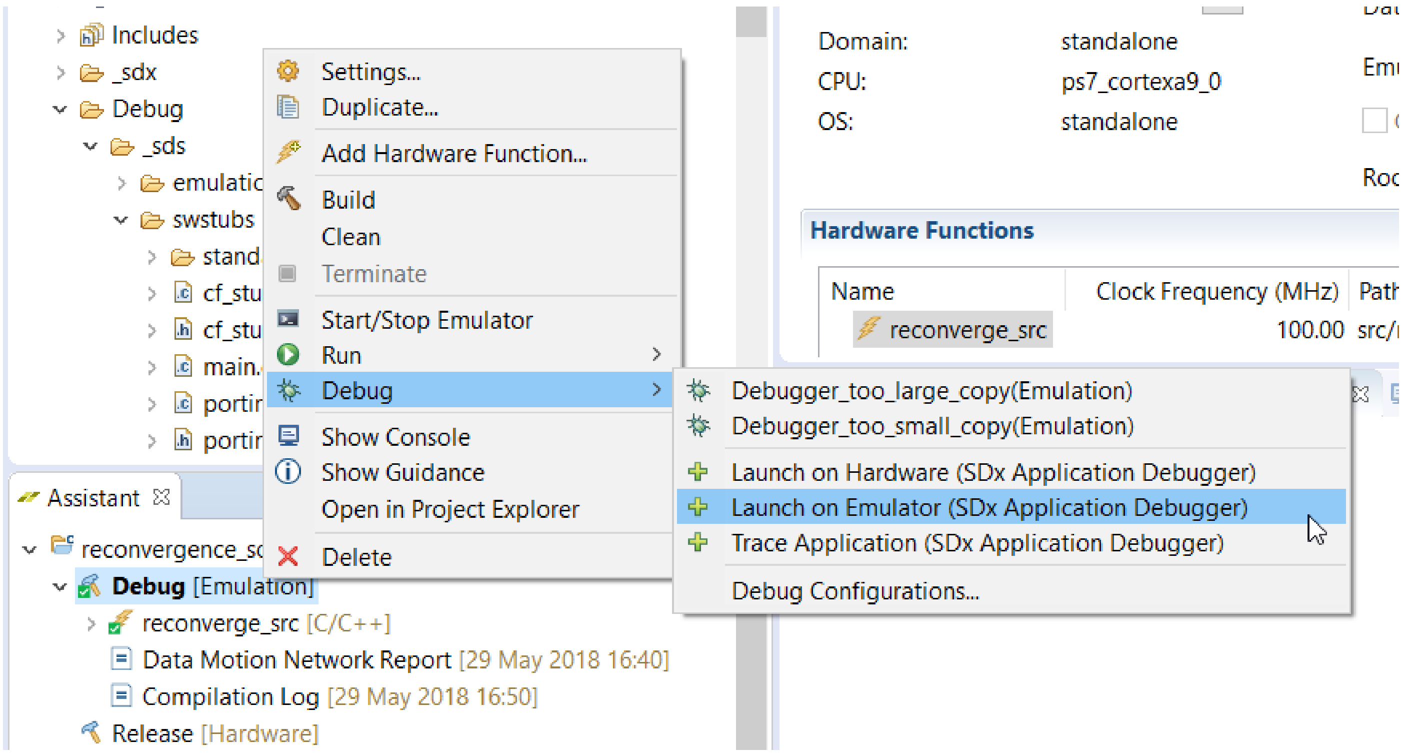
Name	Design Unit
axis_dwc_dm_0_rx...	zc702_axis_dwc_dm_0_rx_0_0
axis_dwc_dm_0_tx...	zc702_axis_dwc_dm_0_tx_0_0
axis_ic_dm_0	zc702_axis_ic_dm_0_0
clk_wiz_0	zc702_clk_wiz_0_0
dm_0	zc702_dm_0_0(zc702_dm_0_0_ar.
proc_sys_reset_0	zc702_proc_sys_reset_0_0(zc702...
proc_sys_reset_1	zc702_proc_sys_reset_1_0(zc702...
proc_sys_reset_2	zc702_proc_sys_reset_2_0(zc702...
proc_sys_reset_3	zc702_proc_sys_reset_3_0(zc702...
ps7	zc702_ps7_0
sds_irq_const	zc702_sds_irq_const_0
sgdma2axis_dm_0	zc702_sgdma2axis_dm_0_0
too_large_copy_1	zc702_too_large_copy_1_0
too_large_copy_1_if	z
xlconcat_0	z
glbl	g

Name	Value	Data
ap_clk	0	Logic
ap_rst_n	0	Logic
ap_start	X	Logic
ap_done	0	Logic
ap_idle	0	Logic
ap_ready	0	Logic
in_r_dout[31:0]	00000000	Array
in_r_empty_n	0	Logic
in_r_read	0	Logic
out_r_din[31:0]	00000000	Array
out_r_full_n	1	Logic
out_r_write	0	Logic
small_r[31:0]	00000000	Array
large_r[31:0]	00000000	Array

This adds all the I/O ports of the selected function to the Waveform window. Start the simulator by clicking the **Run All** icon in the toolbar.



Go back to SDx™ IDE and launch the application on the debugger. To do this, select the application to be debugged, right-click and select **Launch on Emulator (SDx Application Debugger)**.

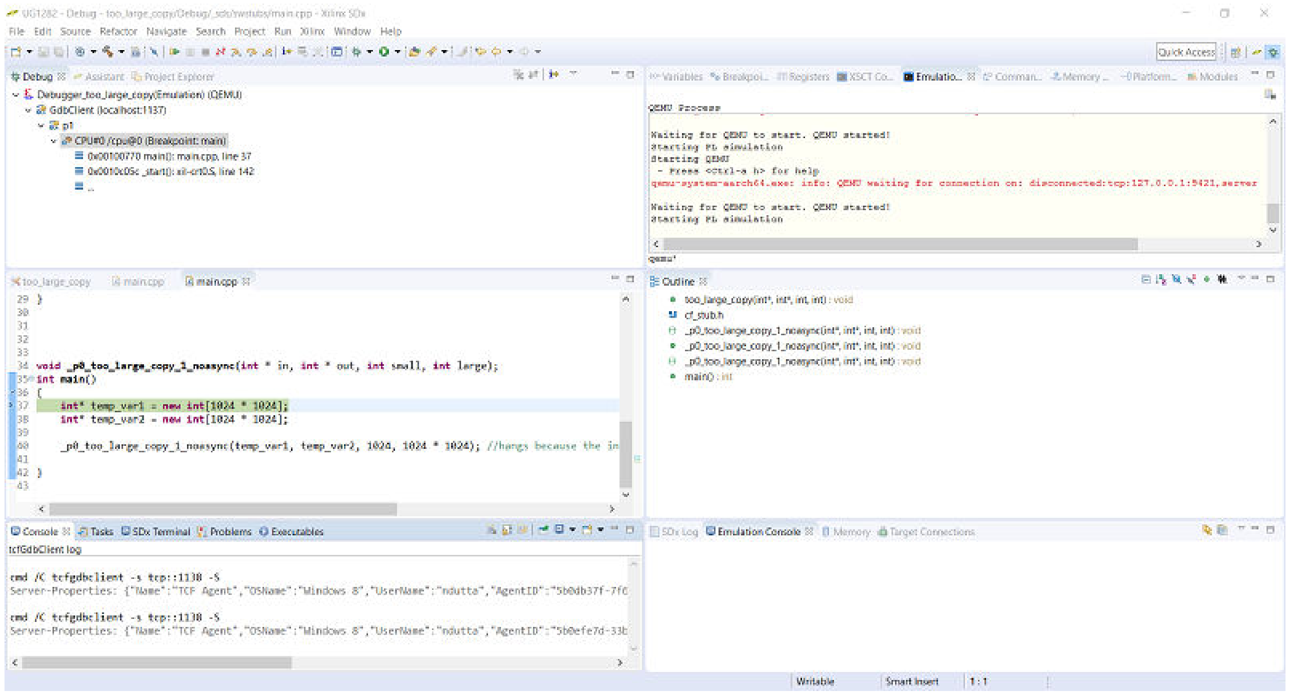


Domain:	standalone
CPU:	ps7_cortexa9_0
OS:	standalone

Hardware Functions

Name	Clock Frequency (MHz)	Path
reconverge_src	100.00	src/

On the **Confirm Perspective Switch** dialog box, click **Yes**. The Debug Perspective opens with the application running on the hardware. The code execution stops at the main program entry.



```
Debugger_too_large_copy(Emulation) (QEMU)
  GdbClient (localhost:1137)
    p1
      CPU#0 /cpu@0 (Breakpoint: main)
        0x00100770 main(): main.cpp, line 37
        0x0010c05c _start(): xil-crt0.S, line 142
        ..
```

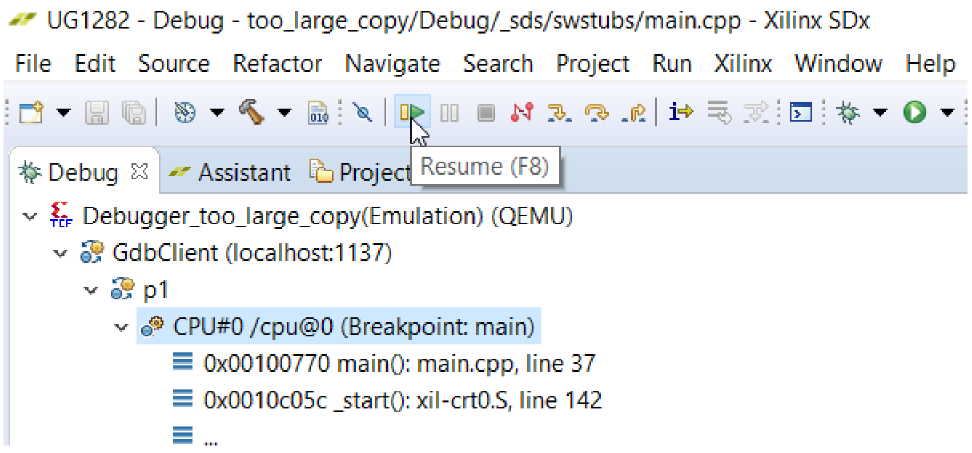
```
QEMU Process

Waiting for QEMU to start. QEMU started!
Starting PL simulation
Starting QEMU
 - Press <ctrl-a h> for help
qemu-system-aarch64.exe: info: QEMU waiting for connection on: disconnected:tcp:127.0.0.1:5421,server

Waiting for QEMU to start. QEMU started!
Starting PL simulation
qemu'
```

```
}




void _p0_too_large_copy_1_noasync(int * in, int * out, int small, int large);
int main()
{
    int* temp_var1 = new int[1024 * 1024];
    int* temp_var2 = new int[1024 * 1024];

    _p0_too_large_copy_1_noasync(temp_var1, temp_var2, 1024, 1024 * 1024); //hangs because the in
}
```

```
too_large_copy(int*, int*, int, int) : void
cf_stub.h
_p0_too_large_copy_1_noasync(int*, int*, int, int) : void
_p0_too_large_copy_1_noasync(int*, int*, int, int) : void
_p0_too_large_copy_1_noasync(int*, int*, int, int) : void
main() : int
```

```
tcfGdbClient log

cmd /C tcfgdbclient -s tcp::1138 -S
Server-Properties: {"Name":"TCF Agent","OSName":"Windows 8","UserName":"ndutta","AgentID":"5b0db37f-7fc

cmd /C tcfgdbclient -s tcp::1138 -S
Server-Properties: {"Name":"TCF Agent","OSName":"Windows 8","UserName":"ndutta","AgentID":"5b0efe7d-33b
```

Click the **Resume** button on the toolbar to execute the application.


UG1282 - Debug - too_large_copy/Debug/_sds/swstubs/main.cpp - Xilinx SDx

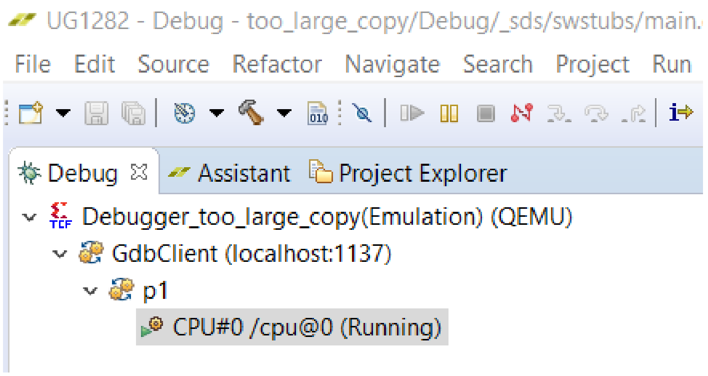
File Edit Source Refactor Navigate Search Project Run Xilinx Window Help

Resume (F8)

Debug | Assistant | Project

```
Debugger_too_large_copy(Emulation) (QEMU)
  GdbClient (localhost:1137)
    p1
      CPU#0 /cpu@0 (Breakpoint: main)
        0x00100770 main(): main.cpp, line 37
        0x0010c05c _start(): xil-crt0.S, line 142
        ...
```

Notice that the application is now "stuck". In other words a system hang has been encountered.

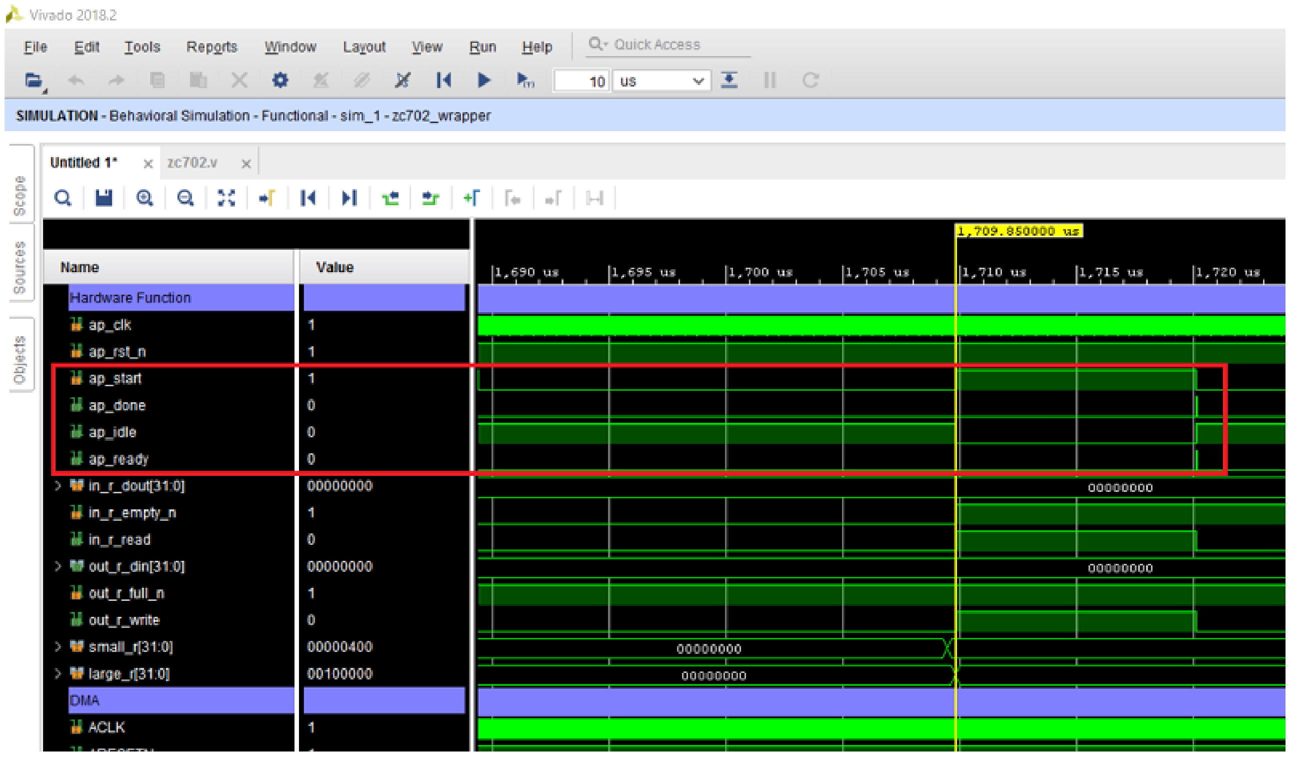

UG1282 - Debug - too_large_copy/Debug/_sds/swstubs/main.

File Edit Source Refactor Navigate Search Project Run

Debug | Assistant | Project Explorer

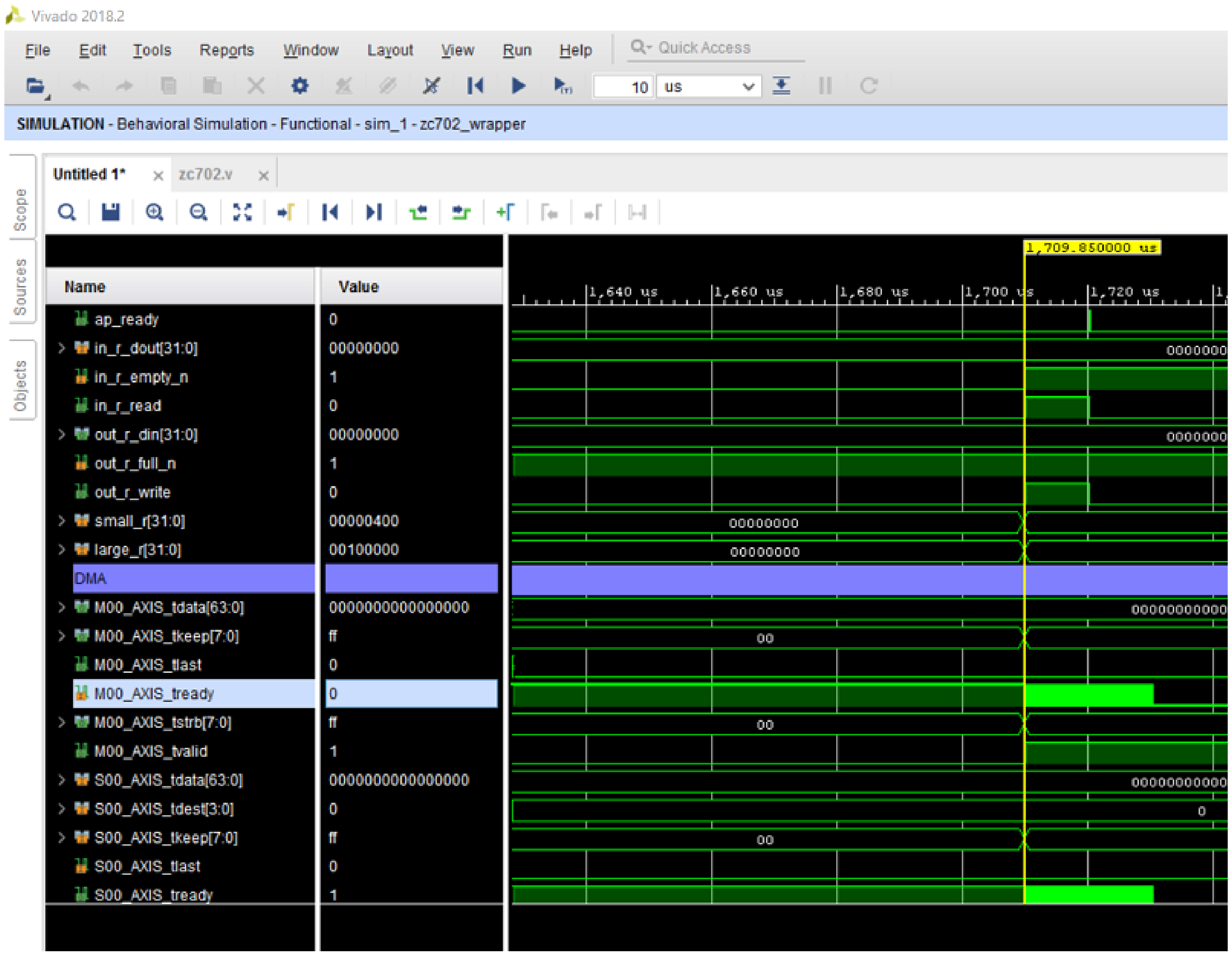
```
Debugger_too_large_copy(Emulation) (QEMU)
  GdbClient (localhost:1137)
    p1
      CPU#0 /cpu@0 (Running)
```

To determine why the system has hung up, go back to Vivado Design Suite. Look at the state of the ap_done, ap_start, ap_idle and ap_ready signals for the function. The state of these signals indicate that a transaction was started at the instance when the ap_start signal went high, followed by the transaction ending when the ap_done signal went low. The ap_ready and ap_idle signals likewise indicate the state of the function.



Name	Value
Hardware Function	
ap_clk	1
ap_rst_n	1
ap_start	1
ap_done	0
ap_idle	0
ap_ready	0
in_r_dout[31:0]	00000000
in_r_empty_n	1
in_r_read	0
out_r_din[31:0]	00000000
out_r_full_n	1
out_r_write	0
small_r[31:0]	00000400
large_r[31:0]	00100000
DMA	
ACLK	1

Analyzing the state of DMA at the same point of time, you will notice that while the hardware function is finished accepting data, the DMA is still writing to it, as indicated by the M00_AXIS_tready and the M00_AXIS_tvalid signals.



SIMULATION - Behavioral Simulation - Functional - sim_1 - zc702_wrapper

Name	Value
ap_ready	0
in_r_dout[31:0]	00000000
in_r_empty_n	1
in_r_read	0
out_r_din[31:0]	00000000
out_r_full_n	1
out_r_write	0
small_r[31:0]	00000400
large_r[31:0]	00100000
DMA	
M00_AXIS_tdata[63:0]	0000000000000000
M00_AXIS_tkeep[7:0]	ff
M00_AXIS_tlast	0
M00_AXIS_tready	0
M00_AXIS_tstrb[7:0]	ff
M00_AXIS_tvalid	1
S00_AXIS_tdata[63:0]	0000000000000000
S00_AXIS_tdest[3:0]	0
S00_AXIS_tkeep[7:0]	ff
S00_AXIS_tlast	0
S00_AXIS_tready	1

Now that you know the cause of the system hang, you can go back to the hardware function code and fix any outstanding issues.

There are other situations where a system hang can occur as listed below:

1. If you can **Ctrl+C** out of the application, there was probably not enough data from the accelerator. The Arm® is expecting more data than the accelerator is sending. Review latencies if there is more than one path from a producer to a consumer. Designs where there are multiple paths with equal latencies between two accelerators (A -> B ... -> Z while there is also A -> Z Direct) need to be fixed at the design level equalizing the branches.
2. If **Ctrl+C** does not work, but you can `ping` or `ssh` into the board there is not enough data in a Scatter Gather DMA (SGDMA) operation. Review data movers (copy or zero-copy) and access pattern.
3. If you can not `ping` the board and it has hard locked, only coming back to life after a power cycle, common causes are interaction between the following:
    a. The SDSoC design and IP on the platform. Debug with ChipScope and peeking and poking registers.

b. The SDSoC design and C-Callable IP libraries. Debug with ChipScope and peeking and poking of registers.

c. The RTL or the SW driver generated in the SDSoC flow. If you have enough Vivado Design Suite or C driver experience you might be able to debug this; otherwise contact the forums.

The following list shows other sources of run-time errors:

- Improper placement of `wait()` statements could result in:
  - Software reading invalid data before a hardware accelerator has written the correct value.
  - A blocking `wait()` being called before a related accelerator is started, resulting in a system hang.
- Inconsistent use of memory consistency #pragma `SDS data mem_attribute` can result in incorrect results.

Unexpected Data Values

When the application is running, it is possible to get unexpected data. The hardware function may not be returning the expected data, or it may be returning expected data at the wrong time. This can be caused by hardware and/or software issues.

If hardware is suspect, check data inputs to your board using ChipScope if needed.

If software is suspect:

1. Go back to software debug and confirm that your software is good.
2. If the software debug is good, you need to visually inspect the code. Two common causes for unexpected data are from the use of the #SDS data or the #SDS zero copy pragmas.

   a. If using #SDS data pragmas the tools trust what you write. Confirm that the data access pattern int the code matches data access pattern specified by the pragma.

   b. A mis-sized (normally too large) #SDS zero copy can pull invalid data from cache. This is seen in hardware. Emulation is likely to pass as there is no cache controller in software.

Peeking and Poking IP Registers

The Xilinx® System Debugger tool (XSDB) is most useful for understanding what is going on with IP blocks included with the platform or various C-callable IP blocks.

For details on XSBD see *SDK Online Help* (UG782)

CAUTION!: *Trying to access an address that is not mapped reports a BUS ERROR; addresses that are mapped but lack proper backing result in a system hang.*

Event Tracing

This section describes how traces are collected and displayed in the SDSoC™ Environment.

Runtime Trace Collection

Software traces are inserted into the same storage path as the hardware traces and receive a timestamp using the same timer/counter as hardware traces. This single trace data stream is buffered in the hardware system and accessed over JTAG by the host PC.

In the SDSoC™ environment, traces are read back constantly as the program executes attempting to empty the hardware buffer as quickly as possible and prevent buffer overflow. However, trace data only displays when the application is finished.

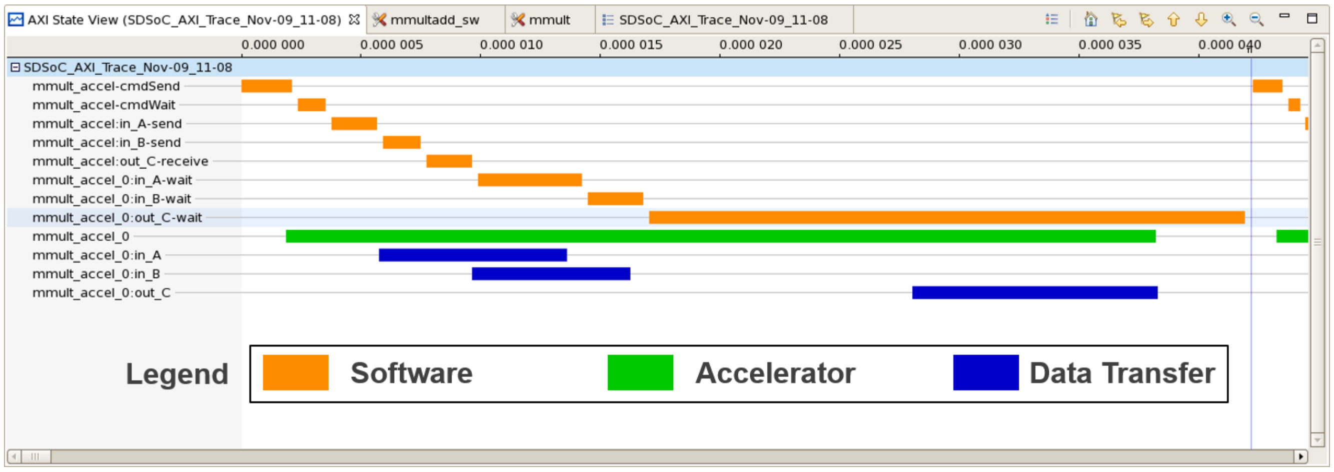
Trace data is collected in real time when you are running on the hardware.

For information about connecting to the hardware, refer to Connecting to the Hardware.

Trace Visualization

The SDSoC™ environment GUI provides a graphical rendering of the hardware and software trace stream. Each trace point in the user application is given a unique name, and its own axis/swimlane on the timeline. In general, a trace point can create multiple trace events throughout the execution of the application, for example, if the same block of code is executed in a loop or if an accelerator is invoked more than once.

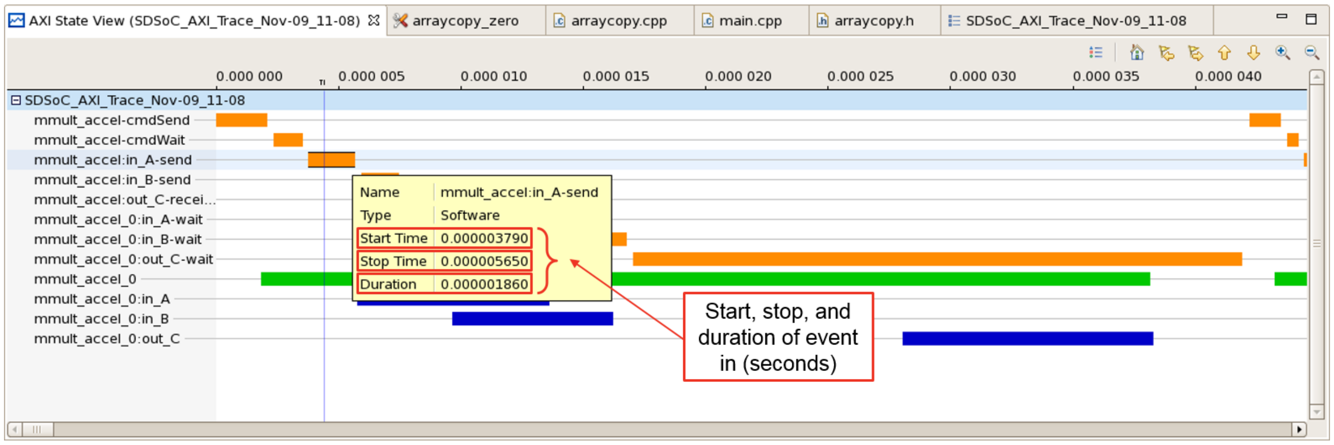
Figure 14: **Example Trace Visualization Highlighting the Different Types of Events**



X16913-050216

Each trace event has a few different attributes: name, type, start time, stop time, and duration. This data is shown as a tool-tip when the curser hovers above one of the event rectangles in the view.

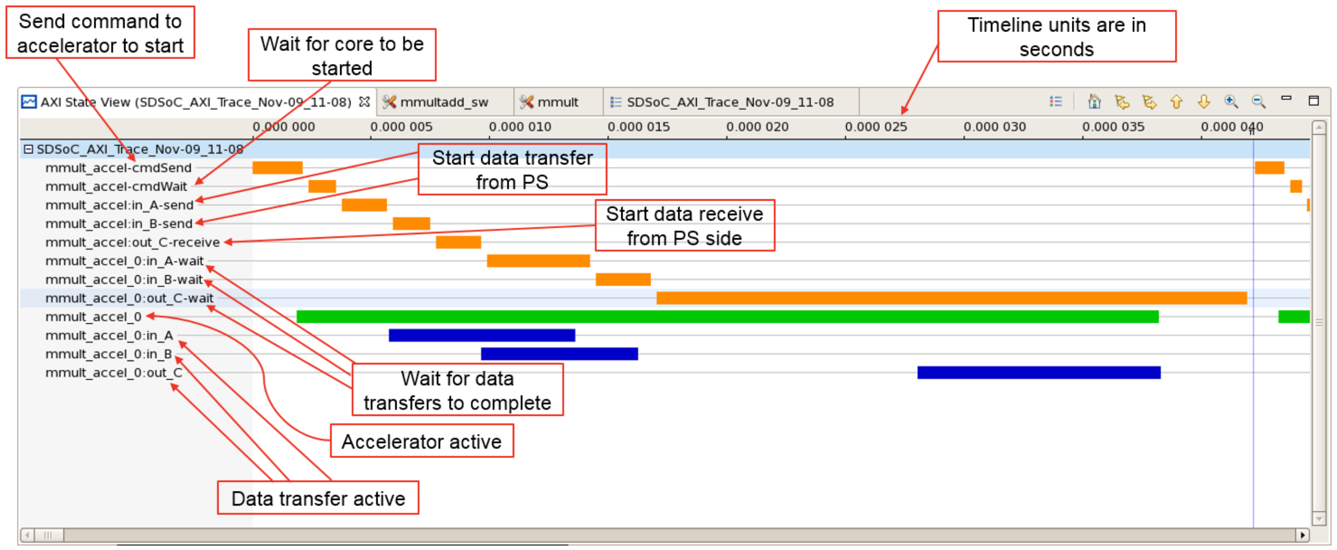
Figure 15: **Example Trace Visualization Highlighting the Detailed Information Available for Each Event**



Name	mmult_accel:in_A-send
Type	Software
Start Time	0.000003790
Stop Time	0.000005650
Duration	0.000001860

X16912-050216

Figure 16: **Example Trace Visualization Highlighting the Event Names and Correlation to the User Program**



X16914-050216

Troubleshooting

1. Incremental build flow: The SDSoC™ Environment does not support any incremental build flow using the trace feature. To ensure the correct build of your application and correct trace collection, be sure to do a project clean first, followed by a build after making any changes to your source code. Even if the source code you change does not relate to or impact any function marked for hardware, you can see incorrect results.
2. Programming and bitstream: The trace functionality is a "one-shot" type of analysis. The timer used for timestamping events is not started until the first event occurs and runs forever afterwards. If you run your software application once after programming the bitstream, the timer will be in an unknown state after your program is finished running. Running your software for a second time will result in incorrect timestamps for events. Be sure to program the bitstream first, followed by downloading your software application, each and every time you run your application to take advantage of the trace feature. Your application will run correctly a second time, but the trace data will not be correct. For Linux, you need to reboot because the bitstream is loaded during boot time by U-Boot.
3. Buffering up traces: In the SDSoC Environment, traces are buffered up and read out in real-time as the application executes (although at a slower speed than they are created on the device), but are displayed after the application finishes in a post-processing fashion. This relies on having enough buffer space to store traces until they can be read out by the host PC. By default, there is enough buffer space for 1024 traces. After the buffer fills up, subsequent traces that are produced are dropped and lost. An error condition is set when the buffer overflows. Any traces created after the buffer overflows are not collected, and traces just prior to the overflow might be displayed incorrectly.

4. Errors: In the SDSoC Environment, traces are buffered up in hardware before being read out over JTAG by the host PC. If traces are produced faster than they are consumed, a buffer overflow event might occur. The trace infrastructure is cognizant of this and will set an error flag that is detected during the collection on the host PC. After the error flag is parsed during trace data collection, collection is halted and the trace data that was read successfully is prepared for display. However, some data read successfully just prior to the buffer overflow might appear incorrectly in the visualization.

After an overflow occurs, an error file is created in the `<build_config>/_sds/trace` directory with the name in the following format: `archive_DAY_MON_DD_HH_MM_SS_-GMT_YEAR_ERROR`. You must reprogram the device (reboot Linux, etc.) prior to running the application and collecting trace data again. The only way to reset the trace hardware in the design is with reprogramming.

Debugging with Software/Hardware Cross Probing

Once an SDx™ application has been created and functions are marked for hardware acceleration, you build the design with the appropriate settings. You then must connect to the target board (see Connecting to the Hardware).

Setting Debug Configurations

To set up the debug configuration:

1. In the Project Explorer view, click the ELF (`.elf`) file in the `Debug` folder in the project.
2. In the toolbar, click the **Debug** icon or use the **Debug** icon pull-down menu to select **Debug As→Launch on Hardware (SDx™ Application Debugger)**.

Alternatively, right-click the project and select **Debug As→Launch on Hardware (SDx Application Debugger)**. The **Confirm Perspective Switch** dialog box appears.

Ensure that the board is switched on before debugging the project. Click **Yes** to switch to the debug perspective. You are now in the **Debug Perspective** of the SDx IDE.

Note that the debugger resets the system, programs and initializes the device, then breaks at the main function. The source code is shown in the center panel, local variables in the top right corner panel and the SDx log at the bottom right panel shows the Debug Configuration log.

Before you start running your application, connect a serial terminal to the board so you can see the output from your program. As an example the following settings can be used:

Connection Type: Serial

Port: COM<n>

Baud Rate: 115200 baud

Running the Application

Click the **Resume** icon to run your application, and observe the output in the terminal window. The source code window shows the `_exit` function, and the **Terminal** tab shows the output from the application.

Tips for Debugging Performance

The SDSoC™ Environment provides some basic performance monitoring capabilities with the following functions:

- `sds_clock_counter()`: Use this function to determine how much time different code sections, such as the accelerated code and the non-accelerated code, take to execute.
- `sds_clock_frequency()`: This function returns the number of CPU cycles per second.

You can estimate the actual hardware acceleration time by looking at the latency numbers in the Vivado® Design Suite HLS report files (`_sds/vhls/…/*.rpt`) or in the GUI under **Reports→ HLS Report**. The latency of X accelerator clock cycles equals `X * (processor_clock_freq/ accelerator_clock_freq)processor clock cycles`. Compare this with the time spent on the actual function call to determine the overhead of setup and data transfers.

For best performance improvement, the time required for executing the accelerated function must be much smaller than the time required for executing the original software function. If this is not true, try to run the accelerator at a higher frequency by selecting a different `clkid` on the `sds++` command line. If that does not work, try to determine whether the data transfer overhead is a significant part of the accelerated function execution time, and reduce the data transfer overhead. Note that the default `clkid` is 100 MHz for all platforms. More details about the `clkid` values for the given platform can be obtained by running `sdscc -sds-pf-info <platform name>`.

If the data transfer overhead is large, the following changes might help:

- Move more code into the accelerated function so that the computation time increases, and the ratio of computation to data transfer time is improved.
- Reduce the amount of data to be transferred by modifying the code or using pragmas to transfer only the required data.
- Sequentialize the access pattern as observed from the accelerator code, as it is more efficient to burst transfers than to make a series of unrelated random accesses.

- Assure that data transfers make use of system ports that are appropriate for the cache-ability of the data being transfered. Cache flushing can be an expensive procedure, and using coherent ports to access coherent data, and noncoherent ports to access non-coherent ports makes a big difference.

  Use `sds_alloc()` instead of `malloc`, where possible. The memory that `sds_alloc()` issues is physically contiguous, and enables the use of datamovers that are faster to configure that require physically contiguous memory. Also, pinning virtual pages, which is necessary when transfering data issue by `malloc()` data, is very costly.

Troubleshooting Compile and Link Time Errors

Typical compile/link time errors are indicated by error messages issued when running `make`. To probe further, look at the log files and `rpt` files in the `_sds/reports` subdirectory created by the SDSoC™ Environment in the build directory. The most recently generated log file usually indicates the cause of the error, such as a syntax error in the corresponding input file, or an error generated by the tool chain while synthesizing accelerator hardware or the data motion network.

Some tips for dealing with SDSoC Environment specific errors follow.

Tool Errors Are Reported by Tools in the SDSoC Environment Chain

Try the following troubleshooting steps:

- Check whether the corresponding code adheres to the Coding Guidelines found in *SDSoC Environment Programmers Guide* (UG1278).
- Check the syntax of pragmas.
- Check for typos in pragmas that might prevent them from being applied to the correct function.

Vivado® Design Suite High-Level Synthesis (HLS) Cannot Meet Timing Requirement

Try the following troubleshooting steps:

- Select a slower clock frequency for the accelerator in the SDSoC IDE (or with the `sdscc/sds++` command line parameter).
- Modify the code structure to allow HLS to generate a faster implementation. See the Improving Hardware Function Parallelism section in *SDSoC Environment Profiling and Optimization Guide* (UG1235) for more information on how to do this.

Vivado Tools Cannot Meet Timing

Try the following troubleshooting steps:

- In the SDSoC IDE, select a slower clock frequency for the data motion network or accelerator, or both (from the command line, use `sdscc/sds++` command line parameters).
- Provide an example/resource to help the user synthesize the HLS block to a higher clock frequency so that the synthesis/implementation tools have a bigger margin.
- Modify the C/C++ code passed to HLS, or add more HLS directives to make the HLS block go faster.
- Reduce the size of the design in case the resource usage (see the Vivado tools report in `_sds/p0/_vpl/ipi/*.log` and other log files in the subdirectories there) exceeds 80%. See the next item for ways to reduce the design size.

The Design Is Too Large to Fit

Try the following troubleshooting steps:

- Reduce the number of accelerated functions.
- Change the coding style for an accelerator function to produce a more compact accelerator. You can reduce the amount of parallelism using the mechanisms described in the Improving Hardware Function Parallelism section in *SDSoC Environment Profiling and Optimization Guide* (UG1235).
- Modify pragmas and coding styles (pipelining) that cause multiple instances of accelerators to be created.
- Use pragmas to select smaller data movers such as AXIFIFO instead of AXIDMA_SG.
- Rewrite hardware functions to have fewer input and output parameters/arguments, especially in cases where the inputs/outputs are continuous stream (sequential access array argument) types that prevent sharing of data mover hardware.

Troubleshooting Performance Issues

The SDSoC™ environment provides some basic performance monitoring capabilities in the form of the `sds_clock_counter()` function described earlier. Use this to determine how much time different code sections take to execute, such as the accelerated code, and the non-accelerated code.

Estimate the actual hardware acceleration time by looking at the latency numbers in the Vivado® HLS report files (`_sds/vhls/…/*.rpt`). In the SDSoC IDE Project Platform Details tab, you can determine the CPU clock frequency, and in the Project Overview you can determine the clock frequency for a hardware function. A latency of X accelerator clock cycles is equal to `X * (processor_clock_freq/accelerator_clock_freq)` processor clock cycles. Compare this with the time spent on the actual function call to determine the data transfer overhead.

For best performance improvement, the time required for executing the accelerated function must be much smaller than the time required for executing the original software function. If this is not true, try to run the accelerator at a higher frequency by selecting a different clkid on the `sdscc/sds++` command line. If that does not work, try to determine whether the data transfer overhead is a significant part of the accelerated function execution time, and reduce the data transfer overhead. Note that the default `clkid` is 100 MHz for all platforms. More details about the `clkid` values for the given platform can be obtained by running `sdscc -sds-pf-info <platform name>`.

If the data transfer overhead is large, the following changes might help:

- Move more code into the accelerated function so that the computation time increases, and the ratio of computation to data transfer time is improved.
- Reduce the amount of data to be transferred by modifying the code or using pragmas to transfer only the required data.

SDSoC Environment Troubleshooting

There are several common types of issues you might encounter using the SDSoC™ Environment flow.

- Compile/link time errors can be the result of typical software syntax errors caught by software compilers, or errors specific to the SDSoC Environment flow, such as the design being too large to fit on the target platform.
- Runtime errors can be the result of general software issues such as null-pointer access, or SDSoC Environment-specific issues such as incorrect data being transferred to/from accelerators.
- Performance issues are related to the choice of the algorithms used for acceleration, the time taken for transferring the data to/from the accelerator, and the actual speed at which the accelerators and the data motion network operate.
- Incorrect program behavior can be the result of logical errors in code that fails to implement algorithmic intent.

Additional Resources and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see Xilinx Support.

Solution Centers

See the Xilinx Solution Centers for support on devices, software tools, and intellectual property at all stages of the design cycle. Topics include design assistance, advisories, and troubleshooting tips

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see Xilinx Support.

Documentation Navigator and Design Hubs

Xilinx® Documentation Navigator provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. To open the Xilinx Documentation Navigator (DocNav):

- From the Vivado® IDE, select **Help → Documentation and Tutorials**.
- On Windows, select **Start → All Programs → Xilinx Design Tools → DocNav**.
- At the Linux command prompt, enter `docnav`.

Xilinx Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In the Xilinx Documentation Navigator, click the **Design Hubs View** tab.
- On the Xilinx website, see the Design Hubs page.

***Note*:** For more information on Documentation Navigator, see the Documentation Navigator page on the Xilinx website.

References

These documents provide supplemental material useful with this guide:

1. *SDSoC Environments Release Notes, Installation, and Licensing Guide* (UG1294)
2. *SDSoC Environment User Guide* (UG1027)
3. *SDSoC Environment Getting Started Tutorial* (UG1028)
4. *SDSoC Environment Platform Creation Tutorial* (UG1236)
5. *SDSoC Environment Platform Development Guide* (UG1146)
6. *SDSoC Environment Profiling and Optimization Guide* (UG1235)
7. *SDx Command and Utility Reference Guide* (UG1279)
8. *SDSoC Environment Programmers Guide* (UG1278)
9. *SDSoC Environment Debugging Guide* (UG1282)
10. *SDx Pragma Reference Guide* (UG1253)
11. *UltraFast Embedded Design Methodology Guide* (UG1046)
12. *Zynq-7000 SoC Software Developers Guide* (UG821)
13. *Zynq UltraScale+ MPSoC: Software Developers Guide* (UG1137)
14. *ZC702 Evaluation Board for the Zynq-7000 XC7Z020 SoC User Guide* (UG850)
15. *ZCU102 Evaluation Board User Guide* (UG1182)
16. *Vivado Design Suite User Guide: High-Level Synthesis* (UG902)
17. *Vivado Design Suite: Creating and Packaging Custom IP* (UG1118)
18. SDSoC Development Environment web page
19. Vivado® Design Suite Documentation

Training Resources

1. SDSoC Development Environment and Methodology
2. Advanced SDSoC Development Environment and Methodology

Please Read: Important Legal Notices

AUTOMOTIVE APPLICATIONS DISCLAIMER

Copyright