# SDSoC Environment User Guide

UG1027 (v2017.2) August 16, 2017

XILINX
ALL PROGRAMMABLE™

# Revision History

The following table shows the revision history for this document.

| Date | Version | Revision |
|---|---|---|
| 08/16/2017 | 2017.2 | • Updated for SDx™ IDE 2017.2.<br>• Added Compiling Your OpenCL Kernel Using the Xilinx OpenCL Compiler (xocc). |
| 06/20/2017 | 2017.1 | • Updated for SDx IDE 2017.1.<br>• Added Getting Started with Examples. |

# Table of Contents

## Improving System Performance



## Debugging an Application



## Hardware/Software Event Tracing



## SDSoC Pragma Specification

## SDSCC/SDS++ Compiler Commands and Options



## Exporting a Library for GCC



## Compiling Your OpenCL Kernel Using the Xilinx OpenCL Compiler (xocc)



## SDSoC Environment API

## Additional Resources and Legal Notices

# The SDSoC Environment

The SDSoC™ (software-defined system-on-chip) 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 All Programmable SoCs and Zynq UltraScale+™ MPSoCs, as well as MicroBlaze™ processor-based applications on all Xilinx SoCs and FPGAs. 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, and other IPs, and an 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 allow you to iterate over design and architecture changes by refactoring at the program level, dramatically reducing the time needed to achieve working programs running on the target platform.

## Getting Started

Download and install the SDSoC™ environment according to the directions provided in *SDx Environments Release Notes, Installation, and Licensing Guide* (UG1238). This guide provides detailed instructions and hands-on tutorials to introduce the primary work flows for project creation, specifying functions to run in programmable logic, system compilation, debugging, and performance estimation. Working through these tutorials is the best way to get an overview of the SDSoC environment, and should be considered prerequisite to application development.

**NOTE:** *The SDSoC environment includes the entire tools stack to create a bitstream, object code, and executables. If you have installed the Xilinx® Vivado® Design Suite and Software Development Kit tools independently, you should not attempt to combine these installations with the SDSoC environment.*

## Feature Overview

The SDSoC™ environment inherits many of the tools in the Xilinx® Software Development Kit (SDK), including GNU toolchains and standard libraries (for example, glibc) as well as the Target Communication Framework (TCF) and GDB interactive debuggers, a performance analysis perspective within the Eclipse/CDT-based GUI, and command-line tools.

The SDSoC environment includes system compilers (`sdscc/sds++`) that generate 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 SDSoC system compilers employ 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 you employ with the SDSoC environment, using well established platform-based design methodologies. The SDSoC system compiler generates an application-specific system on chip by customizing a target platform. The SDSoC environment includes a number of platforms for application development and others are provided by Xilinx partners. The *SDSoC Environment Platform Development Guide* (UG1146) describes how to capture platform metadata so that a pre-existing design built using the Vivado Design Suite, and corresponding software run-time environment can be used to build an SDSoC platform and used in the SDSoC environment.

An SDSoC platform defines a base hardware and software architecture and application context, including processing system, external memory interfaces, custom input/output, and software run time including operating system (possibly "bare metal"), boot loaders, drivers for platform peripherals and root file system. Every project you create within the SDSoC environment targets a specific platform, and you employ the tools within the SDSoC 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.

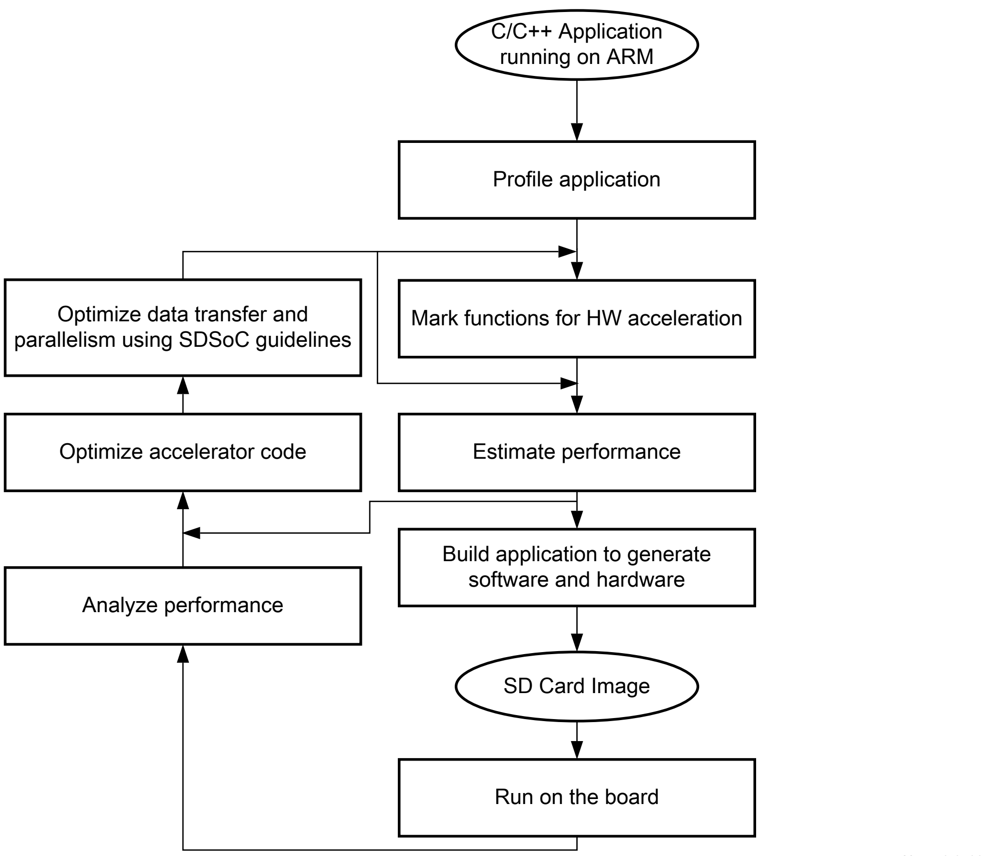
Chapter 2

# User Design Flows

The SDSoC environment is a tool suite for building efficient application-specific systems-on-chip, starting from a platform SoC that provides a base hardware and target software architecture including boot options.

The figure below shows a representative top-level user visible design flow that involves key components of the tool suite. For the purposes of exposition, the design flow proceeds linearly from one step to the next, but in practice you are free to choose other work flows with different entry and exit points. Starting with a software-only version of the application that has been cross-compiled for ARM CPUs, the primary goal is to identify portions of the program to move into programmable logic and to implement the application in hardware and software built upon a base platform.

***Figure 1:  User Design Flow***



The first step is to select a development platform, cross-compile the application, and ensure it runs properly on the platform. You then identify compute-intensive hot spots to migrate into programmable logic to improve system performance, and to isolate them into functions that can be compiled into hardware. You then invoke the SDSoC system compiler to generate a complete system-on-chip and SD card image for your application. You can instrument your code to analyze performance, and if necessary, optimize your system and hardware functions using a set of directives and tools within the SDSoC environment.

The system generation process is orchestrated by the `sdscc/sds++` system compilers through the SDSoC IDE or in an SDSoC terminal shell using the command line and makefiles. Using the SDSoC IDE or `sdscc` command line options, you select functions to run in hardware, specify accelerator and system clocks, and set properties on data transfers (for example, interrupt vs. polling for DMA transfers). You can insert pragmas into application source code to control the system mapping and generation flows, providing directives to the system compiler for implementing the accelerators and data motion networks.

Because a complete system compile can be time-consuming compared with an "object code" compile for a CPU, the SDSoC environment provides a faster performance estimation capability. The estimate allows you to approximate the expected speed-up over a software-only implementation for a given choice of hardware functions and can be functionally verified and analyzed through system emulation. The system emulation feature uses a QEMU model executing the software and RTL model of the hardware functions to enable fast and accurate analysis of the system.

As shown in the preceding figure (User Design Flow), the overall design process involves iterating the steps until the generated system achieves your performance and cost objectives.

It is assumed that you have already worked through the introductory tutorials (see *SDSoC Environment Tutorial: Introduction* (UG1028)) and are familiar with project creation, hardware function selection, compilation, and running a generated application on the target platform. If you have not done so, it is recommended you do so before continuing.

# Creating a Project for a Target Platform

In the SDSoC IDE, click on **File→New→Xilinx SDx Project** to create a new project and open up the **New Project** wizard. After entering the project name, the first step is to select a platform target for development from the **Choose Hardware Platform** window. The platform includes a base hardware system, software runtime (including operating system), boot loaders, and root file system. For an SDSoC environment project, the platform must be one of the hardware platforms from the Zynq-7000 or Zynq UltraScale+ families.

**NOTE:** *The hardware platform is fixed and the command line options are automatically inserted into every makefile. To retarget a project to a new platform, you must create a new project with the new platform and copy the source files from your current project into the new project.*

In addition to the available base platforms, you can manage other aspects of the hardware target from this window:

- Add Custom Platform: This allows you to add your own platform to the list of available platforms. Simply navigate to the top-level directory of the custom platform, select it and press **OK** to add the new platform. The custom platform is immediately available for selection from the list of available platforms. You can find sample platforms in the `<sds_root>/samples/platforms` directory.
- Manage Repositories: This allows you to both add or remove standard and custom platforms. If a custom platform is added, the path to the new platform is automatically added to the repositories. Removing any platform from the list of repositories removes the platform from the **Choose Hardware Platform** selection.
- Add Devices/Platforms: This allows you to manage which Xilinx devices (FPGAs) and platforms are installed. If a device or platform is not selected for inclusion during the installation process, the device will not be available for selection and any platform that uses the device will not be available for selection.

The menu option **Xilinx→Add Custom Platform** can be used at any time to directly add custom platforms and manage the repositories.

**NOTE:** *The clCreateBuffer flag option CL_MEM_USE_HOST_PTR is not supported. OpenCL is only supported in the Linux environment.*

In the Choose Software Platform and Target CPU window, select a System Configuration which defines the software environment that runs on the hardware platform, including the CPU and operating system (OS). For OpenCL support, the System Configuration must be set to A53 OpenCL Linux: the Runtime selection will automatically update to OpenCL. C/C++ is supported for all platforms.

Next, in the Templates window, select **Empty Application** to create a blank project into which you can add files or select from one of the available templates. Finally, review the application description to determine if it is a good starting point for your project, and click **Finish** to open the project.

In addition to the SDSoC IDE, a command line interface is provided.

- For C based projects this is invoked using `sdscc` command.
- For C++ projects this is invoked using the `sds++` command.
- For OpenCL projects this is invoked using the `xocc` command.
- The command line executables are located in `<sdx_root>/bin`.

If you are using the command line interface and writing makefiles outside of the SDSoC IDE, you must include the platform using the `-sds-pf` command line option on every call to `sdscc`. You can also specify the software platform, which includes the operating system that runs on the target CPU, using the `-sds-sys-config <system_configuration>` command line option.

```
sdscc -sds-pf <platform path name>
```

Here, the platform is either a file path or a named platform within the `<sdsoc_root>/platforms` directory. To view the available base platforms from the command line, run the following command.

```
sdscc -sds-pf-list
```

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 `sdscc` system compiler.

The hardware/software interface is defined implicitly in your application source code once you have selected a platform and a set of functions in the program to be implemented in hardware. The `sdscc/sds++` system compilers analyze the program data flow involving hardware functions, schedule each such function call, and generate a hardware accelerator and data motion network realizing the hardware functions in programmable logic. They do so not by implementing each function call on the stack through the standard ARM application binary interface, but instead by redefining hardware function calls as calls to function stubs having the same interface as the original hardware function. These stubs are implemented with low-level function calls to a `send/receive` middleware layer that efficiently transfers data between the platform memory and CPU and hardware accelerators, interfacing as needed to underlying kernel drivers.

The `send/receive` calls are implemented in hardware with data mover IP cores based on program properties like memory allocation of array arguments, payload size, the corresponding hardware interface for a function argument, as well as function properties such as memory access patterns and latency of the hardware function.

## Data Motion Network Clock

Every platform supports one or more clock sources, one of which is selected by default if you do not make an explicit choice. This default clock is defined by the platform provider, and is used for the data motion network (data mover IPs and control buses) generated by `sdscc` during system generation. You can view the platform clocks by selecting the **Platform** link in the **General** panel of the **SDx Project Settings** window. You can select a different platform clock frequency with the **Data Motion Network Clock Frequency** pull-down menu in the **Options** panel of the **SDx Project Settings** window, or on the command line with the `-dmclkid` option.

```
sdscc -sds-pf zc702 -dmclkid 1
```

To see the available clocks for a platform from the command line, execute the following:

```
$ sdscc -sds-pf-info zc702
Platform Information
====================
Name: zc702

Device
------
 Architecture: zynq
       Device: xc7z020
      Package: clg484
  Speed grade: -1

System Clocks
-------------
  Clock ID   Frequency
 ----------|------------
             666.666687
        0    166.666672
        1    142.857132
        2    100.000000
        3    200.000000

Platform:    zc702 (/opt/Xilinx/SDx/2017.2/platforms/zc702)
Description: Basic platform targeting the ZC702 board, which includes 1GB
of DDR3, 16MB Quad-SPI Flash and an SDIO card interface. More information
at https://www.xilinx.com/products/boards-and-kits/ek-z7-zc702-g.html
Available system configurations:
  linux (linux Linux OS on a9_0)
  standalone (standalone Standalone OS on a9_0)
  freertos (freertos FreeRTOS on a9_0)
  ocl (ocl Linux OS on a9_0)
```

## Compiling and Running Applications on an ARM Processor

A first step in application development is to cross-compile your application code to run on the target platform. Every platform included in the SDSoC environment includes a pre-built SD card image from which you can boot and run cross-compiled application code. When you do not select any functions for hardware in your project, this pre-built image is used.

When you make code changes, including changes to hardware functions, it is valuable to rerun a software-only compile to verify your changes did not adversely change your program. A software-only compile is much faster than a full system compile, and software-only debugging is a much quicker way to detect logical program errors than hardware/software debugging.

The SDSoC environment includes two distinct toolchains for the ARM® Cortex™-A9 CPU within Zynq®-7000 SoCs.

1. `arm-linux-gnueabihf` - for developing Linux applications
2. `arm-none-eabi` - for developing standalone ("bare-metal") and FreeRTOS applications

For the ARM Cortex-A53 CPUs within the Zynq UltraScale+™ MPSoCs, the SDSoC environment includes two toolchains:

- `aarch64-linux-gnu` - for developing Linux applications
- `aarch64-none-elf` - for developing standalone ("bare-metal") applications

For the ARM Cortex-R5 CPU provided in the Zynq UltraScale+ MPSoCs, the following toolchain is include in the SDSoC environment:

- `armr5-none-eabi` - for developing standalone ("bare-metal") applications

The underlying GNU toolchain is defined when you select the operating system during project creation. The SDSoC system compilers (`sdscc/sds++`) automatically invoke the corresponding toolchain when compiling code for the CPUs, including all source files not involved with hardware functions.

The SDSoC system compilers generate an SD card image by default in a project subdirectory named `sd_card`. For Linux applications, this directory includes the following files:

- `README.TXT`- contains brief instructions on how to run the application
- `BOOT.BIN` - the boot image contains first stage boot loader (FSBL), boot program (U-Boot), and the FPGA bitstream
- `image.ub` - contains the Linux boot image (platforms can be created that include `uImage`, `devicetree.dtb`, and `uramdisk.image.gz` files)
- `<app>.elf` - the application binary executable

To run the application, copy the contents of `sd_card` directory onto an SD card and insert into the target board. Open a serial terminal connection to the target and power up the board (for more information see *SDSoC Environment Tutorial: Introduction* (UG1028)). Linux boots, automatically logs you in as root, and enters a bash shell. The SD card is mounted at `/mnt`, and from that directory you can run `<app>.elf`.

For standalone applications, the ELF, bitstream, and board support package (BSP) are contained within `BOOT.BIN`, which automatically runs the application after the system boots.

## Compiling and Running Applications on a MicroBlaze Processor

The SDSoC environment includes the standard SDK toolchain for MicroBlaze processors, including `microblaze-xilinx-elf` for developing standalone ("bare-metal") and FreeRTOS applications. A MicroBlaze platform in SDSoC is a standard MicroBlaze processor system built using the Vivado tools and SDK that must be a self-contained system with a local memory bus (LMB) memory, MicroBlaze Debug Module (MDM), UART, and AXI timer. By default, the SDSoC system compilers do not generate an SD card image for projects targeting a MicroBlaze platform. A user can package the bitstream and corresponding ELF executable as needed for their application. To run an application, the bitstream must be programmed onto the device before the ELF can be downloaded to a MicroBlaze core. The SDSoC environment includes Vivado tools and SDK facilities to create MCS files, insert an ELF file into a bitstream, and boot the system from an SD card.

# Profiling and Instrumenting Code to Measure Performance

The first major task in creating a software-defined SoC is to identify portions of application code that are suitable for implementation in hardware, and that significantly improve overall performance when run in hardware. Program hot-spots that are compute-intensive are good candidates for hardware acceleration, especially when it is possible to stream data between hardware and the CPU and memory to overlap the computation with the communication. Software profiling is a standard way to identify the most CPU-intensive portions of your program.

The SDSoC environment includes all performance and profiling capabilities that are included in the Xilinx SDK, including `gprof`, the non-intrusive Target Communication Framework (TCF) Profiler, and the Performance Analysis perspective within Eclipse.

To run the TCF Profiler for a standalone application, run the following steps:

1. Set the active build configuration to **Debug** by right-clicking on the project in the Project Explorer and selecting **Build Configurations→Set Active→Debug**.
2. In the **SDSoC Project Overview** window, click on **Debug application**.

   **NOTE:** *The board must be connected to your computer and powered on. The application automatically breaks at the entry to* `main()`.
3. Launch the TCF Profiler by selecting **Window→Show View→Other→Debug→TCF Profiler**.
4. Start the TCF Profiler by clicking on the green **Start** button at the top of the **TCF Profiler** tab. Enable **Aggregate per function** in the **Profiler Configuration** dialog box.
5. Start the profiling by clicking on the **Resume** button. The program runs to completion and breaks at the `exit()` function.
6. View the results in the **TCF Profiler** tab.

Profiling provides a statistical method for finding hot spots based on sampling the CPU program counter and correlating to the program in execution. Another way to measure program performance is to instrument the application to determine the actual duration between different parts of a program in execution.

The `sds_lib` library included in the SDSoC environment provides a simple, source code annotation based time-stamping API that can be used to measure application performance.

```
/*
 * @return value of free-running 64-bit Zynq(TM) global counter
 */
unsigned long long sds_clock_counter(void);
```

By using this API to collect timestamps and differences between them, you can determine duration of key parts of your program. For example, you can measure data transfer or overall round trip execution time for hardware functions as shown in the following code snippet:

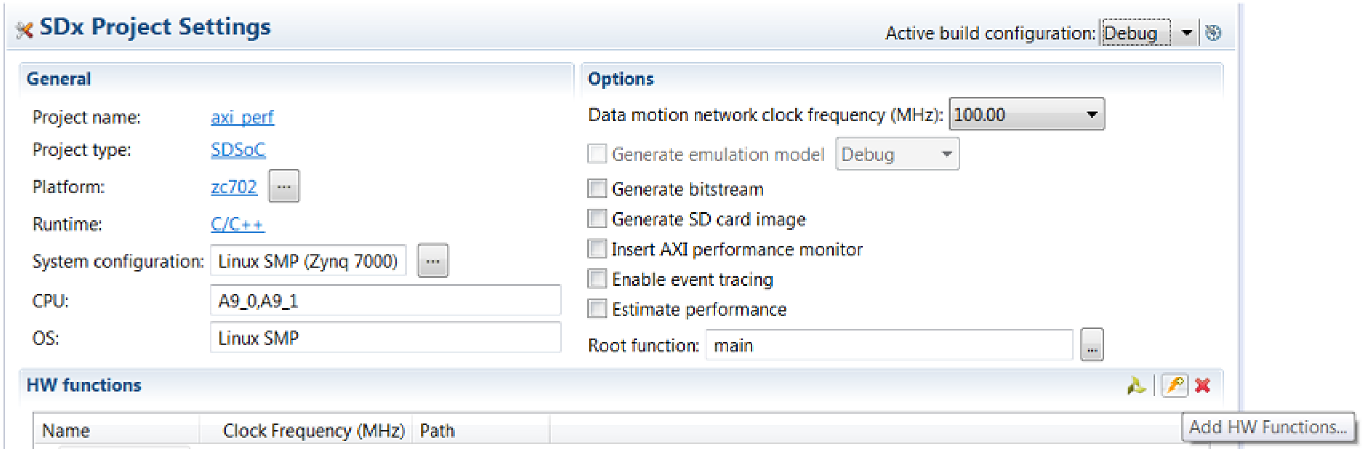
```
class perf_counter
{
public:
     uint64_t tot, cnt, calls;
     perf_counter() : tot(0), cnt(0), calls(0) {};
     inline void reset() { tot = cnt = calls = 0; }
     inline void start() { cnt = sds_clock_counter(); calls++; };
     inline void stop() { tot += (sds_clock_counter() - cnt); };
     inline uint64_t avg_cpu_cycles() { return (tot / calls); };
};

extern void f();
void measure_f_runtime()
{
     perf_counter f_ctr;
     f_ctr.start();
     f()
     f_ctr.stop();
     std::cout << "Cpu cycles f(): " << f_ctr.avg_cpu_cycles()
                    << std::endl;
}
```

The performance estimation feature within the SDSoC environment employs this API by automatically instrumenting functions selected for hardware implementation, measuring actual run-times by running the application on the target, and then comparing actual times with estimated times for the hardware functions.

**NOTE:** *While off-loading CPU-intensive functions is probably the most reliable heuristic to partition your application, it is not guaranteed to improve system performance without algorithmic modification to optimize memory accesses. A CPU almost always has much faster random access to external memory than you can achieve from programmable logic, due to multi-level caching and a faster clock speed (typically 2x to 8x faster than programmable logic). Extensive manipulation of pointer variables over a large address range, for example, a sort routine that sorts indices over a large index set, while very well-suited for a CPU, may become a liability when moving a function into programmable logic. This does not mean that such compute functions are not good candidates for hardware, only that code or algorithm restructuring may be required. This issue is also well-known for DSP and GPU coprocessors.*

# Moving Functions into Programmable Logic

When you have created a new project, you can open up the **SDSoC Project Overview** by double-clicking on the `project.sdsoc` file in the **Project Explorer**.




Click on the symbol in the **Hardware Functions** panel to display the list of candidate functions within your program. The list of Hardware Functions consists of functions in the call graph rooted at the `Root Function` as defined in the **General** panel as shown above, and is set to `main` by default. The `Root Function` can be changed by clicking on the **...** button and selecting an alternative function root.

From within the popup window, you can select one or more functions for hardware acceleration and click **OK**. The selected functions appear in the list box. Note that the Eclipse CDT indexing mechanism is not foolproof, and you might need to close and reopen the selection popup to view available functions. If a function does not appear in the list, you can navigate to its containing file in the **Project Explorer**, expand the contents, right-click on the function prototype, and select **Toggle HW/SW**.

From the command line, select a function `foo` in the file `foo_src.c` for hardware with the following `sdscc` command line option.

```
-sds-hw foo foo_src.c -sds-end
```

If `foo` invokes sub-functions contained in files `foo_sub0.c` and `foo_sub1.c`, use the `-files` option.

```
-sds-hw foo foo_src.c -files foo_sub0.c,foo_sub1.c -sds-end
```

Although the control buses and data mover IPs within the data motion network run off of a single clock, it is possible to run hardware functions and zero_copy data buses at different clock rates to achieve higher performance. In the **Hardware Functions** panel, select functions from the list and use the **Clock Frequency** pull-down menu to choose their clocks. Be aware that it might not be possible to implement the hardware system with some clock selections.

To set a clock on the command-line, determine the corresponding clock id using `sdscc -sds-pf-info <platform>` and use the `-clkid` option.

```
-sds-hw foo foo_src.c -clkid 1 -sds-end
```

When moving a function optimized for CPU execution into programmable logic, you usually need to revise the code to achieve best performance. See Improving Hardware Function Parallelism and Coding Guidelines for programming guidelines.

For xFAST libraries, right-click and select **Toggle Hardware** from the associated header files in the project includes in project explorer. Also add the function chooser GUI.

# System Emulation

After the hardware functions are identified, the logic can be compiled into hardware and the entire system (PS and PL) verified using emulation. This provides the same level of accuracy as the final implementation without the need to compile the system into a bitstream and program the FPGA on the board.

Within the SDx Project Settings, select **Generate Emulation Model** to enable system emulation. Because emulation does not require a full system compile, you might be asked to disable Generate Bitstream and you are encouraged to do so to improve run time. The bitstream generation takes more time to complete than any other part of the development flow. System emulation allows you to verify and debug the system with the same level of accuracy as a full bitstream compilation.

To capture waveform data from the PL hardware emulation for viewing and debugging, select the **Debug** pull-down menu option. For faster emulation without capturing this hardware debug information, select the **Optimized** pull-down menu option. Use the **Build** toolbar button to compile the system for emulation after selecting **Debug** or **Optimized** mode. Once the system is compiled for emulation, the system emulator is invoked using **Xilinx Tools > Start/Stop Emulator**. When the Emulation window opens you can choose to run the emulation with or without waveforms.

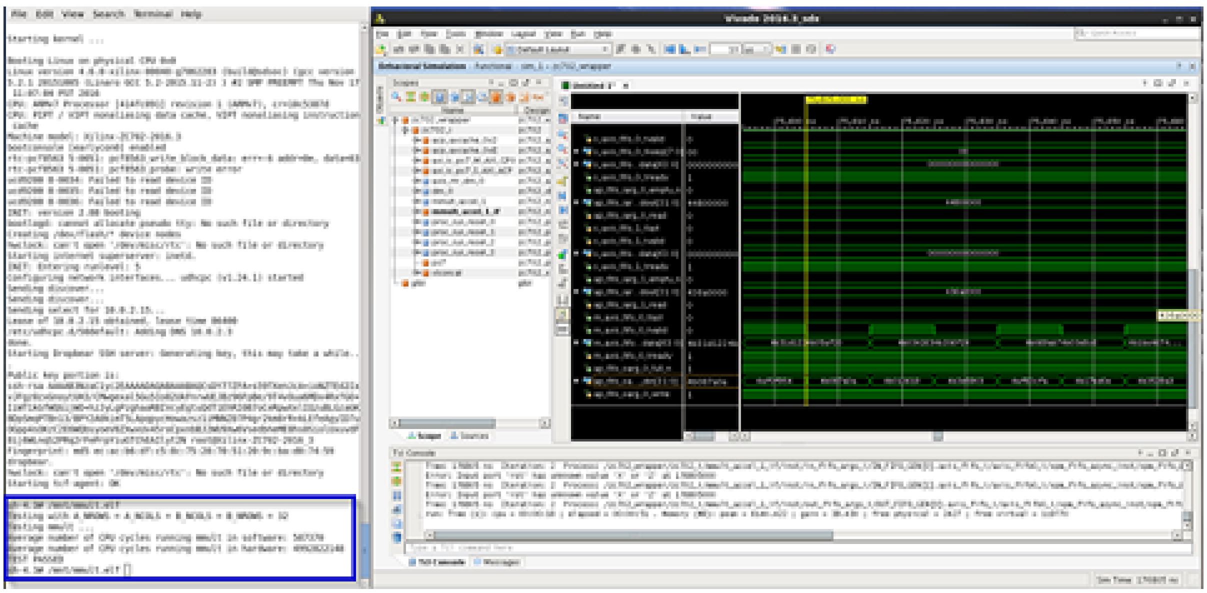
Leaving the **Show Waveform** option unselected allows you to run emulation with output directed solely to the console pane. The console pane shows all system messages including the results of any print statements in the source code. Some of these statements might include the values transferred to and from the hardware functions, if desired, or simply a statement that the application has completed successfully, which would verify that the source code running on the PL and the compiled hardware functions running in the PS are functionally correct.

Selecting the **Show Waveform** option in the Emulation windows provides the same functionality in the console window plus an RTL waveform window. The RTL waveform window allows you to see the value of any signal in the hardware functions over time. When using this option, signals should be manually added to the waveform window before starting the emulation. Use the Scopes pane to navigate the design hierarchy, then select the signals in the Object pane you wish to monitor and use right-click to add the signals to the waveform pane. Press the **Run All** toolbar button to start updates to the waveform window.

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

The system emulation is started by selecting the active project in the Project Navigator and right-clicking to select the menu options **Run→Run As→Launch** on the **Emulator** menu or **Debug→Debug As→Launch** on the **Emulator** menu. You will see the program output in the console tab, and if the **Show Waveform** option was selected, you will also see any appropriate response in the hardware functions in the RTL waveform. With the system emulation running, it can be paused by breakpoints in Debug mode and analysis performed in the debug perspective. 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. The emulation can be stopped at any time using the menu option **Xilinx Tools→Start/Stop Emulator** and selecting **Stop**. For an example suitable for emulation, create a 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.

A system emulation session run from the command-line is shown in the following figure, with the QEMU console shown at left and the PL waveform shown on the right.



## 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.

## 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 reported by tools in the SDSoC environment chain.
  - Check whether the corresponding code adheres to Coding Guidelines.
  - 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.
  - 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 Improving Hardware Function Parallelism for more information on how to do this.
- Vivado tools cannot meet timing.
  - 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).
  - 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% or so. See the next item for ways to reduce the design size.
- Design too large to fit.
  - 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 Improving Hardware Function Parallelism.
  - 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 System Hangs and Runtime Errors

Programs compiled using `sdscc/sds++` can be debugged using the standard debuggers supplied with the SDSoC™ environment or Xilinx® 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.

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:SEQENTIAL)
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.

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.

## 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, such as the accelerated code, and the non-accelerated code take to execute.

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.

## Debugging an Application

The SDSoC™ environment allows projects to be created and debugged using the SDSoC IDE. Projects can also be created outside the SDSoC IDE (user-defined makefiles) and debugged either on the command line or using the SDSoC IDE.

See *SDSoC Environment Tutorial: Introduction* (UG1028) for information on using the interactive debuggers in the SDSoC IDE.

*Chapter 3*

# Coding Guidelines

This contains general coding guidelines for application programming using the SDSoC system compilers, with the assumption of starting from application code that has already been cross-compiled for the ARM CPU within the Zynq® device, using the GNU toolchain included as part of the SDSoC environment.

## Guidelines for Invoking SDSCC/SDS++

The SDSoC IDE automatically generates makefiles that invoke `sds++` for all C++ files and `sdscc` for all C files, but the only source files that must be compiled with `sdscc/sds++` are those containing code that:

- Define a hardware function
- Call a hardware function
- Use `sds_lib` functions, for example, to allocate or memory map buffers that are sent to hardware functions
- Files that contain functions in the transitive closure of the downward call graph of the above

All other source files can safely be compiled with the ARM GNU toolchain.

A large software project may include many files and libraries that are unrelated to the hardware accelerator and data motion networks generated by `sdscc`. If the `sdscc` compiler issues errors on source files unrelated to the generated hardware system (for example, from an OpenCV library), you can compile these files through `GCC` instead of `sdscc` by right-clicking on the file (or folder) **Properties→C/C++ Build→Settings** and setting the **Command** to `GCC`.

## Makefile Guidelines

The makefiles provided with the designs in `<sdsoc_root>/samples` consolidate all `sdscc` hardware function options into a single command line. This is not required, but has the benefit of preserving the overall control structure and dependencies within a makefile without requiring change to the makefile actions for files containing a hardware function.

- You can define make variables to capture the entire SDSoC environment command line, for example: `CC = sds++ ${SDSFLAGS}` for C++ files, invoking `sdscc` for C files. In this way, all SDSoC environment options are consolidated in the *`${CC}`* variable. Define the platform and target OS once in this variable.

- There must be a separate `-sds-hw/-sds-end` clause in the command line for each file that contains a hardware function. For example:

```
-sds-hw foo foo.cpp -clkid 1 -sds-end
```

  For the list of the SDSoC compiler and linker options, see SDSCC/SDS++ Compiler Commands and Options or use `sdscc --help`.

## General C/C++ Guidelines

- Hardware functions can execute concurrently under the control of a master thread. Multiple master threads are supported.
- A top-level hardware function must be a global function, not a class method, and it cannot be an overloaded function.
- There is no support for exception handling in hardware functions.
- It is an error to refer to a global variable within a hardware function or any of its sub-functions when this global variable is also referenced by other functions running in software.
- Hardware functions support scalar types up to 1024 bits, including double, long long, packed structs, etc.
- A hardware function must have at least one argument.
- An output or inout scalar argument to a hardware function can be assigned multiple times, but only the last written value will be read upon function exit.
- Use predefined macros to guard code with `#ifdef` and `#ifndef` preprocessor statements; the macro names begin and end with two underscore characters '_'. For examples, see SDSCC/SDS++ Compiler Commands and Options.
    - The `__SDSCC__` macro is defined and passed as a `-D` option to sub-tools whenever `sdscc` or `sds++` is used to compile source files, and can be used to guard code dependent on whether it is compiled by `sdscc`/`sds++` or by another compiler, for example a GNU host compiler.
    - When `sdscc` or `sds++` compiles source files targeted for hardware acceleration using Vivado HLS, the `__SDSVHLS__` macro is defined and passed as a `-D` option, and can be used to guard code dependent on whether high-level synthesis is run or not.
    - In 2017.2 running on the Windows operating system, you will typically have to use these macros to guard code that will be synthesized within SDx with type long long (which should be 64 bits).
    - Vivado HLS employs some 32-bit libraries irrespective of the host machine. Furthermore, the tool does not provide a true cross-compilation.

All object code for the ARM CPUs is generated with the GNU toolchains, but the `sdscc` (and `sds++`) compiler, built upon Clang/LLVM frameworks, is generally less forgiving of C/C++ language violations than the GNU compilers. As a result, you might find that some libraries needed for your application cause front-end compiler errors when using `sdscc`. In such cases, compile the source files directly through the GNU toolchain rather than through `sdscc`, either in your makefiles or by setting the compiler to `arm-linux-gnueabihf-g++` by right-clicking on the file (or folder) in the **Project Explorer** and selecting **C/C++ Build→Settings→SDSCC/SDS++ Compiler**.

## Hardware Function Argument Types

The SDSoC™ environment `sdscc/sds++` system compilers support hardware function arguments with types that resolve to a single or array of C99 basic arithmetic type (scalar), a `struct` or `class`whose members flatten to a single or array of C99 basic arithmetic type (hierarchical structs are supported), an array of `struct` whose members flatten to a single C99 basic arithmetic type. Scalar arguments must fit in a 1024-bit container. The SDSoC™ environment automatically infers hardware interface types for each hardware function argument based on the argument type and the following pragmas:

```
#pragma SDS data copy|zero_copy

#pragma SDS data access_pattern
```

To avoid interface incompatibilities, you should only incorporate Vivado® HLS interface type directives and pragmas in your source code when `sdscc`/`sds++` fails to generate a suitable hardware interface directive.

- Vivado® HLS provides arbitrary precision types `ap_fixed<int>`, `ap_int<int>`, and an `hls::stream` class. In the SDSoC environment, `ap_fixed<int>` types must be specified as having widths greater than 7 but less than 1025 (7 < width < 1025). The `hls::stream` data type is not supported as the function argument to any hardware function.
- By default, an array argument to a hardware function is transferred by copying the data, that is, it is equivalent to using `#pragma SDS data copy`. As a consequence, an array argument must be either used as an input or produced as an output, but not both. For an array that is both read and written by the hardware function, you must use `#pragma SDS data zero_copy` to tell the compiler that the array should be kept in the shared memory and not copied.
- To ensure alignment across the hardware/software interface, do not use hardware function arguments that are an array of `bool`.

**IMPORTANT:** *Pointer arguments for a hardware function require special consideration. Hardware functions operate on physical addresses, which typically are not available to userspace programs, so pointers cannot be embedded in data structures passed to hardware functions.*

**IMPORTANT:**

*By default, in the absence of any pragmas, a pointer argument is taken to be a scalar parameter, even though in C/C++ it might denote a one-dimensional array type. The following are the permitted pragmas.*

- *This pragma provides pointer semantics using shared memory.*

```
#pragma SDS data zero_copy
```

- *This pragma maps the argument onto a stream, and requires that array elements are accessed in index order. The data copy pragma is only required when the* `sdscc` *system compiler is unable to determine the data transfer size and issues an error.*

```
#pragma SDS data copy(p[0:<p_size>])

#pragma SDS data access_pattern(p:SEQUENTIAL)
```

**IMPORTANT:** *When you require non-sequential access to the array in the hardware function, you should change the pointer argument to an array with an explicit declaration of its dimensions, for example,* `A[1024]`.

## Hardware Function Call Guidelines

- Stub functions generated in the SDSoC™ environment transfer the exact number of bytes according the compile-time determinable array bound of the corresponding argument in the hardware function declaration. If a hardware function admits a variable data size, you can use the following pragma to direct the SDSoC environment to generate code to transfer data whose size is defined by an arithmetic expression:

```
#pragma SDS data copy|zero_copy(arg[0:<C_size_expr>])
#pragma SDS data zero_copy(arg[0:<C_size_expr>])
```

  where the `<C_size_expr>` must compile in the scope of the function declaration.

  The `zero_copy` pragma directs the SDSoC environment to map the argument into shared memory.

  **IMPORTANT:** *Be aware that mismatches between intended and actual data transfer sizes can cause the system to hang at runtime, requiring laborious hardware debugging.*

- Align arrays transferred by DMAs on cache-line boundaries (for L1 and L2 caches). Use the `sds_alloc()` API provided with the SDSoC environment instead of `malloc()` to allocate these arrays.
- Align arrays to page boundaries to minimize the number of pages transferred with the scatter-gather DMA, for example, for arrays allocated with `malloc`.
- You must use `sds_alloc` to allocate an array for the following two cases:
  1. You are using zero-copy pragma for the array.
  2. You are using pragmas to explicitly direct the system compiler to use Simple-DMA.

  Note that in order to use `sds_alloc()` from `sds_lib.h`, it is necessary to include `stdlib.h` before including `sds_lib.h`. `stdlib.h` is included to provide the `size_t` type.

Chapter 4

# Getting Started with Examples

All Xilinx SDx™ Environments are provided with examples designs:

- To help you quickly get started.
- To demonstrate useful coding styles.
- To highlight important optimization techniques.

Example designs are provided with the tool installation and additional examples may be downloaded from the Xilinx® GitHub repository.

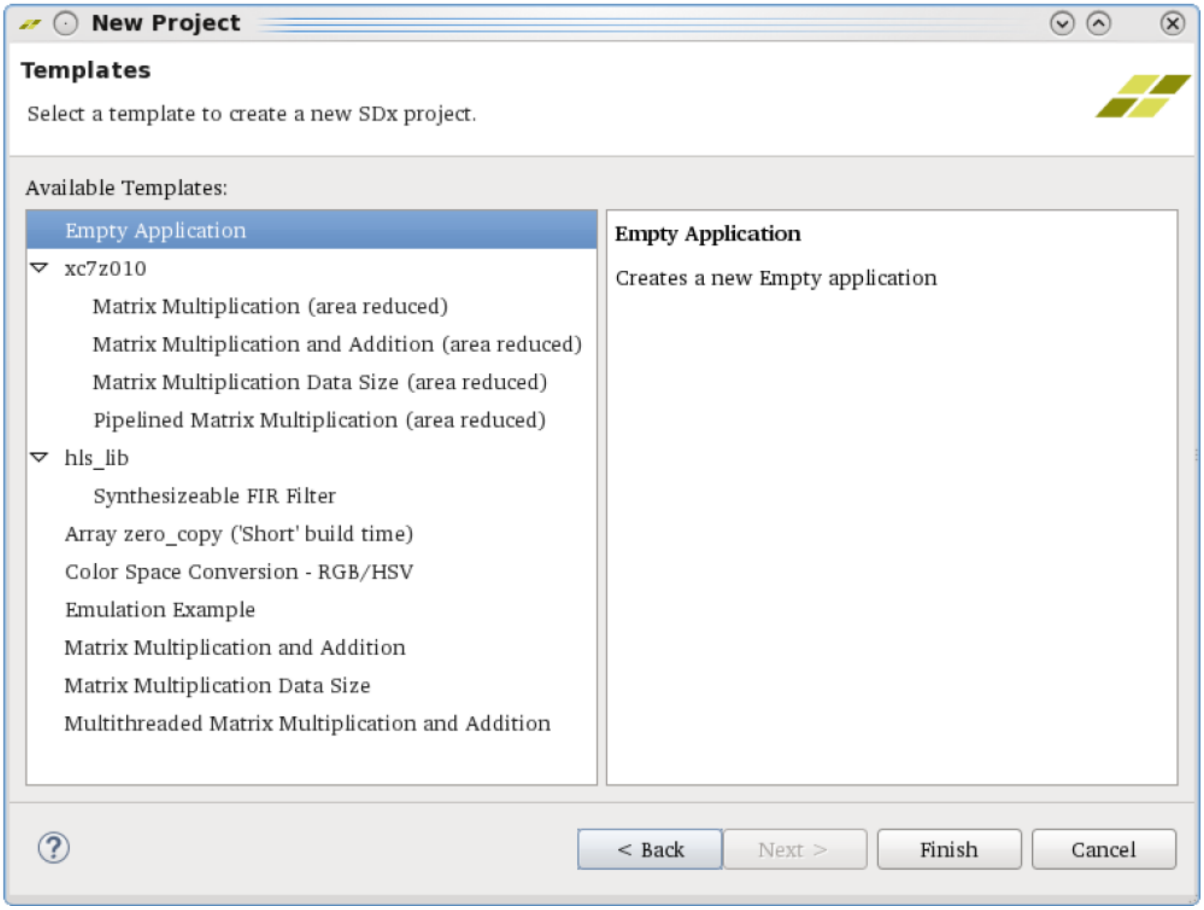
## Installed Examples

The installed examples are provided through the Create SDx™ Project wizard. Select **Create SDx Project** from the SDx Development Environment Welcome page to open the new project wizard. After selecting your hardware platform and software platform, the final page of the wizard lists the available templates.

**NOTE:** *Not all available platforms have an installed example.*

You may select examples from the Templates page, as shown below.

***Figure 2: Templates Page***



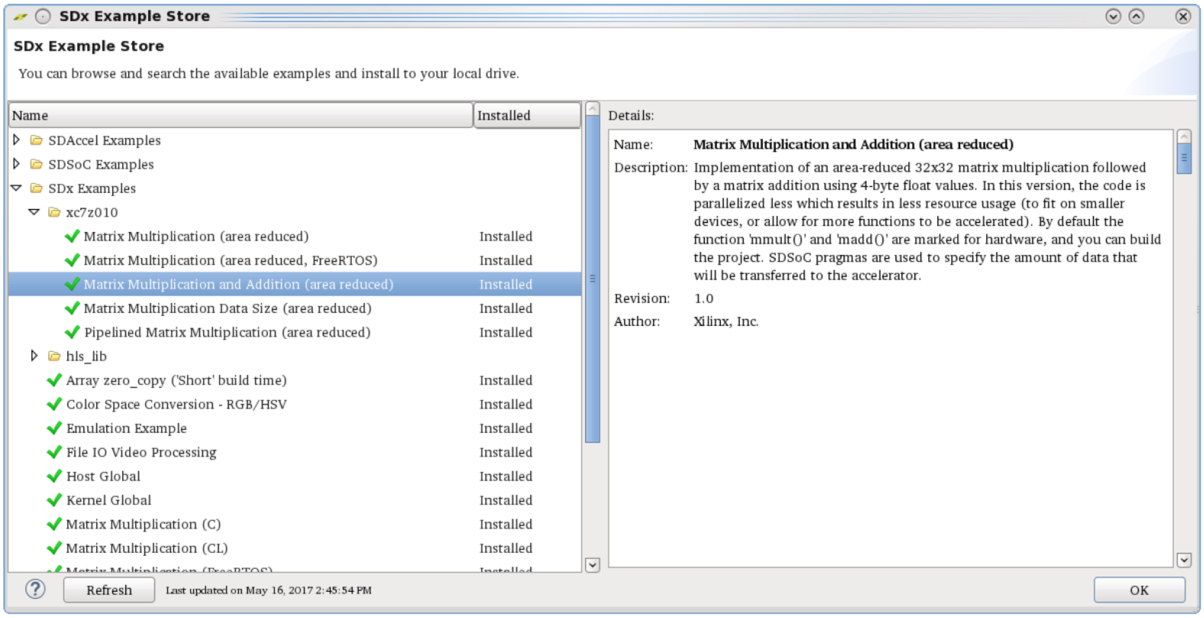

After selecting **Finish**, the example is copied into the local workspace and can be used.

# GitHub Examples

The GitHub examples may be accessed from the menu **Xilinx** > **Open SDx Example Store**. When the **SDx Example Store** dialog box opens it lists all the available examples and indicates if the examples are already installed or not.

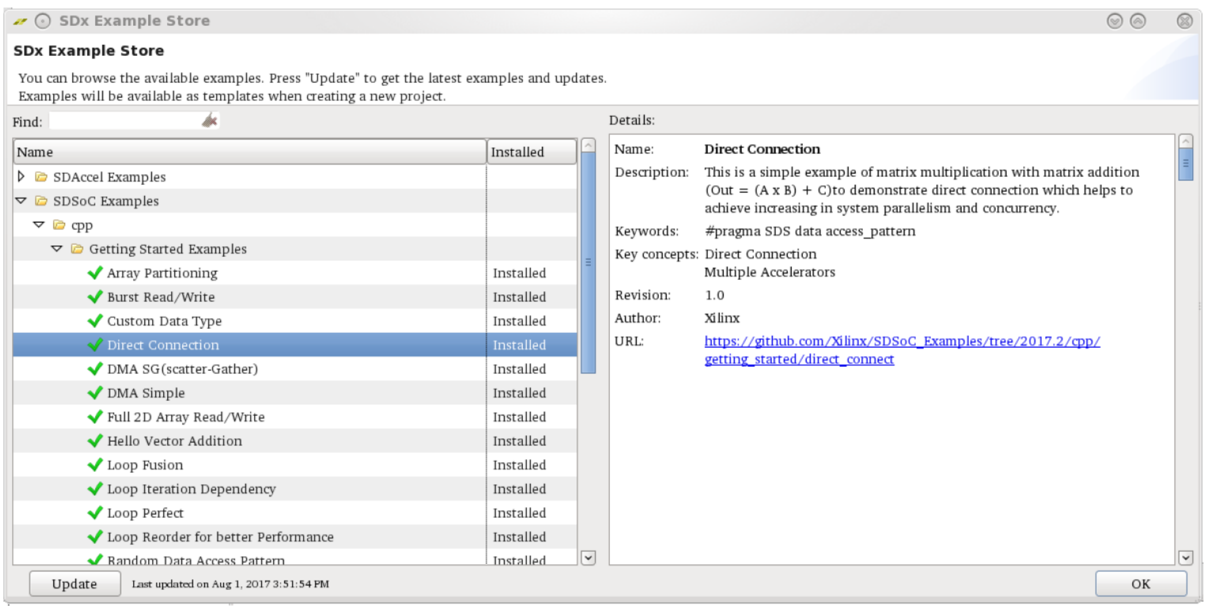
The SDx Examples folder lists all installed examples and shows they are already installed.

*Figure 3:* ***SDx Examples Folder***



The **SDx Example Store** dialog also lists the GitHub exmaples provided for specific SDx environments. For example, the **SDSoC** folder lists all GitHub examples for the SDSoC environment.

***Figure 4: SDx Examples for SDSoC***



The GitHub examples also indicate if they are installed or not. Use the **Refresh** button to ensure you have the latest update from the repository. Click **Install** to download and install the example design.

Once the example design has been installed, it may be accessed during new project creation in the same manner as the installed examples.

## Synthesizeable FIR Filter

Many of the functions in the Vivado HLS source code libraries included in the SDSoC environment do not comply with the SDSoC environment Coding Guidelines. To use these libraries in the SDSoC environment, you typically have to wrap the functions to insulate the SDSoC system compilers from non-portable data types or unsupported language constructs.

The Synthesizeable FIR Filter example demonstrates a standard idiom to use such a library function that in this case, computes a finite-impulse response digital filter. This example uses a filter class constructor and operator to create and perform sample-based filtering. To use this class within the SDSoC environment, the example wraps within a function wrapper as follows.

```
void cpp_FIR(data_t x, data_t *ret)
{
    static CF<coef_t, data_t, acc_t> fir1;
    *ret = fir1(x);
}
```

This wrapper function becomes the top-level hardware function that can be invoked from application code.

## Matrix Multiplication

Matrix multiplication is a common compute-intensive operation for many application domains. The SDSoC IDE provides template examples for all base platforms, and the code for these provide instructive use of SDSoC environment system optimizations for memory allocation and memory access described in Improving System Performance, and Vivado HLS optimizations like function inlining, loop unrolling and pipelining, and array partitioning, described in Optimization Guidelines.

## Using a C-Callable RTL Library

The SDSoC system compilers can incorporate libraries with hardware functions that are implemented using IP blocks written in register transfer level (RTL) in a hardware description language (HDL) like VHDL or Verilog. The process of creating such a library is described in Using C-Callable IP Libraries. This example demonstrates how to incorporate the library in an SDSoC project.

To build this example in the SDSoC IDE, create a new SDSoC project and select the C-callable RTL Library template. As described in `src/SDSoC_project_readme.txt`, you must first build the library from an SDSoC terminal window at the command line.

To use the library and build the application, you must add the `-l` and `-L` linker options as described in Using C-Callable IP Libraries. Right-click on the project in the **Project Explorer** and select **C/C++ Build Settings→SDS++ Linker→Libraries**, to add the `-lrtl_arraycopy` and `-L<path to project>` options.

# C++ Design Libraries

A number of design libraries are provided with the SDSoC installation. The C libraries allow common hardware design constructs and functions to be easily modeled in C and synthesized to RTL. The following C libraries are provided:

- reVISION and Machine Learning ibraries
- Arbitrary Precision Data Types Library
- HLS Stream Library
- HLS Math Library
- HLS Video Library
- HLS IP Library
- HLS Linear Algebra Library
- HLS DSP Library

You can use each of the C libraries in your design by including the library header file. These header files are located in the `include` directory in the SDSoC Environment installation area (`$HOME_SDSOC/Vivado_HLS/include`).

**IMPORTANT:** *The header files for the Vivado HLS C libraries do not have to be in the include path if the C++ code is used in the SDSoC environment.*

*Chapter 5*

# Using C-Callable IP Libraries

Using a C-callable library is similar to using any software library. You `#include` header files for the library in appropriate source files and use the `sdscc -I<path>` option to compile your source, for example

```
> sdscc -c -I<path to header> -o main.o main.c
```

When you are using the SDSoC IDE, you add these `sdscc` options by right-clicking on your project, selecting `C/C++ Build Settings->SDSCC Compiler->Directories` (or `SDS++ Compiler->Directories` for C++ compilation).

To link the library into your application, you use the -L<path> and -l<lib> options.

```
> sdscc -sds-pf zc702 ${OBJECTS} -L<path to library> -l<library_name> -o
myApp.elf
```

As with the standard GNU linkers, for a library called `libMyLib.a`, you use `-lMyLib`.

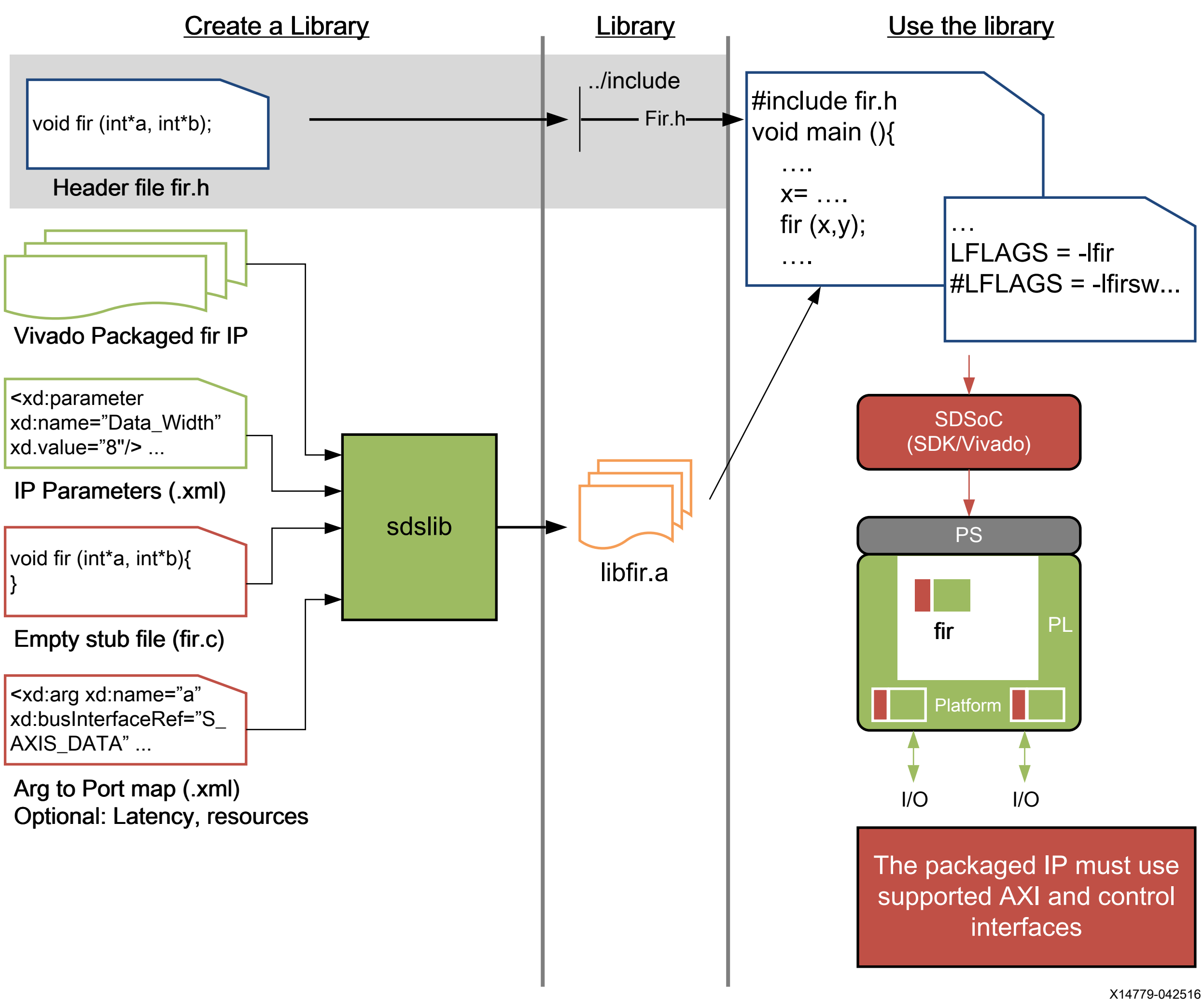
When you are using the SDSoC IDE, you add these `sdscc` options by right-clicking on your project, selecting **C/C++ Build Settings→SDS++ Linker→Libraries**.

You can find code examples that employ C-callable libraries in the SDSoC™ environment installation under the `samples/fir_lib/use` and `samples/rtl_lib/arraycopy/use` directories.

# C-Callable Libraries

This section describes how to create a C-callable library for IP blocks written in a hardware description language like VHDL or Verilog. User applications can statically link with such libraries using the SDSoC system compilers, and the IP blocks will be instantiated into the generated hardware system. A C-callable library can also provide `sdscc`-compiled applications access to IP blocks within a platform (see Creating a Library).

***Figure 5: Create and Use a C-Callable Library***




The following is the list of elements that are part of an SDSoC platform software callable library:

- Header File
  - Function prototype

- Static Library
    - Function definition
    - IP core
    - IP configuration parameters
    - Function argument mapping

## Header File

A library must declare function prototypes that map onto the IP block in a header file that can be #included in user application source files. These functions define the function call interface for accessing the IP through software application code.

For example:

```
 // FILE: fir.h
#define N 256
void fir(signed char X[N], short Y[N]);
```

## Static Library

An SDSoC environment static library contains several elements that allow a software function to be executed on programmable resources.

### *Function Definition*

The function interface defines the entry points into the library, as a function or set of functions that can be called in user code to target the IP. The function definitions can contain empty function bodies since the SDSoC compilers will replace them with API calls to execute data transfers to/from the IP block. The implementation of these calls depend upon the data motion network created by the SDSoC system compilers. The function definition must #include stdlib.h and stdio.h, which are used when the function body is replaced and compiled.

For example:

```
// FILE: fir.c
#include "fir.h"
#include <stdlib.h>
#include <stdio.h>
void fir(signed char X[N], short Y[N])
{
    // SDSoC replaces function body with API calls for data transfer
}
```

**NOTE:** *Application code that links to the library must also #include stdlib.h and stdio.h, which are required by the API calls in the stubs generated by the SDSoC system compilers.*

### *IP Core*

An HDL IP core for a C-callable library must be packaged using the Vivado® tools. This IP core can be located in the Vivado tools IP repository or in any other location. When the library is used, the corresponding IP core is instantiated in the hardware system.

You must package the IP for the Vivado Design Suite as described in the *Vivado Design Suite User Guide: Designing with IP* (UG896). The Vivado IP Packager tool creates a directory structure for the HDL and other source files, and an IP Definition file (`component.xml`) that conforms to the IEEE-1685 IP-XACT standard. In addition, the packager creates an archive zip file that contains the directory and its contents required by Vivado Design Suite.

The IP can export AXI4, AXI4-Lite, and AXI4 Stream interfaces. The IP control register must exist at address offset `0x0`, and can support two different task protocols:

1. '`none`' - in this mode, the control register must be tied to a constant value 0x6. The core then is assumed to run continuously upon power up, with all data synchronized through AXI4 stream interfaces or through asynchronous read or writes to memory-mapped registers via an axilite bus.
2. '`axilite`' - in this mode, the control register must conform to the following specification, which coincides with the `axilite` control interface for an IP generated by Vivado HLS.

The control signals are generally self-explanatory. The `ap_start` signal initiates the IP execution, `ap_done` indicates IP task completion, and `ap_ready` indicates that the IP is can be started. For more details, see the Vivado HLS documentation for the `ap_ctrl_hs` bus definition.

```
// 0x00 : Control signals
// bit 0 - ap_start (Read/Write/COH)
// bit 1 - ap_done (Read/COR)
// bit 2 - ap_idle (Read)
// bit 3 - ap_ready (Read)
// bit 7 - auto_restart (Read/Write)
// others - reserved
// (COR = Clear on Read, COH = Clear on Handshake)
```

**IMPORTANT:** *For details on how to integrate HDL IP into the Vivado Design Suite, see Vivado Design Suite User Guide: Creating and Packaging Custom IP (UG1118).*

### *IP Configuration Parameters*

Most HDL IP cores are customizable at synthesis time. This customization is done through IP parameters that define the IP core’s behavior. The SDSoC environment uses this information at the time the core is instantiated in a generated system. This information is captured in an XML file.

The `xd:component` name is the same as the `spirit:component` name, and each `xd:parameter` name must be a parameter name for the IP. To view the parameter names in IP Integrator, right-click on the block and select **Edit IP Meta Data** to access the IP Customization Parameters.

For example:

```
<!-- FILE: fir.params.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<xd:component xmlns:xd="http://www.xilinx.com/xd" xd:name="fir_compiler">
  <xd:parameter xd:name="DATA_Has_TLAST" xd:value="Packet_Framing"/>
  <xd:parameter xd:name="M_DATA_Has_TREADY" xd:value="true"/>
  <xd:parameter xd:name="Coefficient_Width" xd:value="8"/>
  <xd:parameter xd:name="Data_Width" xd:value="8"/>
  <xd:parameter xd:name="Quantization" xd:value="Integer_Coefficients"/>
  <xd:parameter xd:name="Output_Rounding_Mode" xd:value="Full_Precision"/>
  <xd:parameter xd:name="CoefficientVector"
 xd:value="6,0,-4,-3,5,6,-6,-13,7,44,64,44,7,-13,-6,6,5,-3,-4,0,6"/>
</xd:component>
```

### *Function Argument Map*

The SDSoC system compiler requires a mapping from any function prototypes in the library onto the hardware interface defined by the IP block that implements the function. This information is captured in a "function map" XML file. XML attribute literals, for example array sizes, must be constants and not macros (the SDSoC environment does not use macros in header files to resolve literals in the XML file).

The information includes the following.

- XML namespace - the namespace must be defined as `xmlns:xd="https://www.xilinx.com/xd"`
- Function name – the name of the function mapped onto a component
- Component reference – the IP type name from the IP-XACT Vendor-Name-Library-Version identifier.
  - If the function is associated with a platform, then the component reference is the platform name. For example, see *SDSoC Environment Platform Development Guide* (UG1146).
- C argument name – an address expression for a function argument, for example `x` (pass scalar by value) or `*p` (pass by pointer).

  **NOTE:** *argument names in the function map must be identical to the argument in the function definition, and they must occur in precisely the same order.*
- Function argument direction – either `in` (an input argument to the function) or `out` (an output argument to the function). Currently the SDSoC environment does not support `inout` function arguments.
- Bus interface – the name of the IP port corresponding to a function argument. For a platform component, this name is the platform interface `xd:name`, not the actual port name on the corresponding platform IP.
- Port interface type – the corresponding IP port interface type, which currently must be either `aximm` (slave only), `axis`.
- Address offset – hex address, for example, 0x40, required for arguments mapping onto `aximm` slave ports (this must be a constant).
- Data width – number of bits per datum (this must be a constant).
- Array size – number of elements in an array argument (this must be a constant).

The function mapping for a configuration of the Vivado FIR Filter Compiler IP from `samples/fir_lib/build` is shown below.

```
<!-- FILE: fir.fcnmap.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<xd:repository xmlns:xd="https://www.xilinx.com/xd">
    <xd:fcnMap xd:fcnName="fir" xd:componentRef="fir_compiler">
        <xd:arg xd:name="X"
            xd:direction="in"
            xd:portInterfaceType="axis"
            xd:dataWidth="8"
            xd:busInterfaceRef="S_AXIS_DATA"
            xd:arraySize="32"/>
        <xd:arg xd:name="Y"
            xd:direction="out"
            xd:portInterfaceType="axis"
            xd:dataWidth="16"
            xd:busInterfaceRef="M_AXIS_DATA"
            xd:arraySize="32"/>
        <xd:latencyEstimates xd:worst-case="20" xd:average-case="20"
xd:best-case="20"/>
        <xd:resourceEstimates xd:BRAM="0" xd:DSP="1 xd:FF="200"
xd:LUT="200"/>
    </xd:fcnMap>
</xd:repository>
```

## Creating a Library

Xilinx provides a utility called `sdslib` that allows the creation of SDSoC libraries.

### *Usage*

```
sdslib [arguments] [options]
```

### *Arguments (mandatory)*

| Argument | Description |
|---|---|
| `-lib <libname>` | Library name to create or append to |
| `<function_name file_name>+` | One or more `<function, file>` pairs.<br><br>For example: `fir fir.c` |
| `-vlnv <v>:<l>:<n>:<v>` | Use IP core specified by this vlnv. For example, `-vlnv xilinx.com:ip:fir_compiler:7.1` |
| `-ip-map <file>` | Use specified <file> as IP function map |
| `-ip-params <file>` | Use specified <file> as IP parameters |
| `-pfunc` | IP core is a platform function |

| Option | Description |
|---|---|
| `-ip-repo <path>` | Add HDL IP repository search path |
| `-target-os <name>` | Specify target Operating System<br>• linux (default)<br>• standalone (bare-metal) |
| `--help` | Display this information |
| `-target-cpu <name>` | Specify target CPU<br>• cortex-a9 (default)<br>• cortex-a53<br>• cortex-r5<br>• microblaze |

As an example, to create an SDSoC library for a `fir filter` IP core, call:

```
> sdslib -lib libfir.a \
    fir fir.c \
    fir_reload fir_reload.c \
    fir_config fir_config.c \
    -vlnv xilinx.com:ip:fir_compiler:7.1 \
    -ip-map fir_compiler.fcnmap.xml \
    -ip-params fir_compiler.params.xml
```

In the above example, `sdslib` uses the functions `fir` (in file `fir.c`), `fir_reload` (in file `fir_reload.c`) and `fir_config` (in file `fir_config.c`) and archives them into the `libfir.a` static library. The `fir_compiler` IP core is specified using `-vlnv` and the function map and IP parameters are specified with `-ip-map` and `-ip-params` respectively.

## Testing a Library

To test a library, create a program that uses the library. Include the appropriate header file in your source code. When compiling the code that calls a library function, provide the path to the header file using the `-I` switch.

```
> sdscc -c -I<path to header> -o main.o main.c
```

To link against a library, use the `-L` and `-l` switches.

```
> sdscc -sds-pf zc702 ${OBJECTS} -L<path to library> -lfir -o
 fir.elf
```

In the example above, the compiler uses the library `libfir.a` located at `<path to library>`.

## C-Callable Library Example: Vivado FIR Compiler IP

You can find an example on how to build a library in the SDSoC environment installation under the `samples/fir_lib/build` directory. This example employs a single-channel reloadable filter configuration of the FIR Compiler IP within the Vivado® Design Suite. Consistent with the design of the IP, all communication and control is accomplished over AXI4-Stream channels.

You can also find an example on how to use a library in the SDSoC environment installation under the `samples/fir_lib/use` directory.

## C-Callable Library Example: HDL IP

You can find an example of a Vivado tools-packaged RTL IP in the `samples/rtl_lib/arraycopy/build` directory. This example includes two IP cores, each of which copies M elements of an array from its input to its output, where M is a scalar parameter that can vary with each function call.

- `arraycopy_aximm` - array transfers using an AXI master interface in the IP.
- `arraycopy_axis` - array transfers using AXI4-Stream interfaces.

The register mappings for the IPs are as follows.

```
// arraycopy_aximm
// 0x00 : Control signals
// bit 0 - ap_start (Read/Write/COH)
// bit 1 - ap_done (Read/COR)
// bit 2 - ap_idle (Read)
// bit 3 - ap_ready (Read)
// bit 7 - auto_restart (Read/Write)
// others - reserved
// 0x10 : Data signal of ap_return
// bit 31~0 - ap_return[31:0] (Read)
// 0x18 : Data signal of a
// bit 31~0 - a[31:0] (Read/Write)
// 0x1c : reserved
// 0x20 : Data signal of b
// bit 31~0 - b[31:0] (Read/Write)
// 0x24 : reserved
// 0x28 : Data signal of M
// bit 31~0 - M[31:0] (Read/Write)
// 0x2c : reserved
// (SC = Self Clear, COR = Clear on Read, TOW = Toggle on Write, COH =
Clear on Handshake)

// arraycopy_axis
// 0x00 : Control signals
// bit 0 - ap_start (Read/Write/COH)
// bit 1 - ap_done (Read/COR)
// bit 2 - ap_idle (Read)
// bit 3 - ap_ready (Read)
// bit 7 - auto_restart (Read/Write)
```

```
// others - reserved
// 0x10 : Data signal of ap_return
// bit 31~0 - ap_return[31:0] (Read)
// 0x18 : Data signal of M
// bit 31~0 - M[31:0] (Read/Write)
// 0x1c : reserved
// (SC = Self Clear, COR = Clear on Read, TOW = Toggle on Write, COH =
Clear on Handshake)
```

The makefile indicates how to use `stdlib` to create the library. To build the library, open a terminal shell in the SDSoC IDE, and from within the build directory, run

- `make librtl_arraycopy.a` - to build a library for Linux applications
- `make standalone/lib_rtl_arraycopy.a` - to build a library for standalone applications

A simple test example that employs both IPs is available in the `samples/rtl_lib/arraycopy/use` directory. In an SDSoC terminal shell, run `make` to create a Linux application that exercises both hardware functions.

Chapter 6

# SDSCC/SDS++ Performance Estimation Flow Options

A full bitstream compile can take much more time than a software compile, so `sdscc` provides performance estimation options to compute the estimated run-time improvement for a set of hardware function calls. In the SDSoC environment Project Overview window, invoke the estimator by clicking on **Estimate Performance**, which enables performance estimation for the current build configuration and builds the project.

Estimating the speed-up is a two phase process. First, the SDSoC environment compiles the hardware functions and generates the system. Instead of synthesizing the system to bitstream, `sdscc` computes an estimate of the performance based on estimated latencies for the hardware functions and data transfer time estimates for the callers of hardware functions. In the generated Performance report, select **Click Here** to run an instrumented version of the software on the target to determine a performance baseline and the performance estimate (see *SDSoC Environment Tutorial: Introduction* (UG1028) for more information).

You can also generate a performance estimate from the command line. As a first pass to gather data about software runtime, you use the `-perf-funcs` option to specify functions to profile and `-perf-root` to specify the root function encompassing calls to the profiled functions. The `sdscc` compiler then automatically instruments these functions to collect run-time data when the application is run on a board. When you run an "instrumented" application on the target, the program creates a file on the SD card called `swdata.xml`, which contains the run-time performance data for the run.

Copy `swdata.xml` to the host and run a build that estimates the performance gain on a per hardware function caller basis and for the top-level function specified by the `-perf-root` function in the first pass run. Use the `-perf-est` option to specify `swdata.xml` as input data for this build.

The following table specifies the `sdscc` options normally used to build an application.

| Option | Description |
|---|---|
| `-perf-funcs function_name_list` | Specify a comma separated list of all functions to be profiled in the instrumented software application. |
| `-perf-root function_name` | Specify the root function encompassing all calls to the profiled functions. The default is the function main. |
| `-perf-est data_file` | Specify the file contain runtime data generated by the instrumented software application when run on the target. Estimate performance gains for hardware accelerated functions. The default name for this file is `swdata.xml`. |

| Option | Description |
|---|---|
| `-perf-est-hw-only` | Run the estimation flow without running the first pass to collect software run data. Using this option provides hardware latency and resource estimates without providing a comparison against baseline. |

**CAUTION!**

*After running the sd_card image on the board for collecting profile data, run `cd /; sync; umount /mnt;`. This ensures that the `swdata.xml` file is written out to the SD card.*

*A complete example of the makefile-based flow for performance estimation can be found in `<sdsoc_root>/samples/mmult_performance_estimation`.*

*Chapter 7*

# Improving System Performance

There are many factors that affect overall system performance. A well-designed system generally balances computation and communication so that all hardware components remain occupied doing meaningful work. Some applications will be compute-bound, and for these, you should concentrate on maximizing throughput and minimizing latency in hardware accelerators. Others may be memory-bound, in which case you might need to restructure algorithms to increase temporal and spatial locality in the hardware, for example, by adding copy-loops or memcopy to pull blocks of data into hardware rather than making random array accesses to external memory.

This chapter describes underlying principles and inference rules within the SDSoC system compiler to assist the programmer in controlling the compiler to improve overall system performance through

- An understanding of the data motion network: default behavior and user specification
- Increased system parallelisim and concurrency
- Improved access to external memory from programmable logic
- Increased parallelism in the hardware function

Control over the various aspects of optimization is provided through the use of pragmas in the code. A complete description of the pragmas discussed in this chapter is located in SDSoC Pragma Specification.

## Data Motion Network Generation in SDSoC

This section describes the components that make up the data motion network in the SDSoC™ environment. It helps the user understand the data motion network generated by the SDSoC compiler. The section also provides guidelines to help you guide the data motion network generation by using appropriate SDSoC pragmas.

Every transfer between the software program and a hardware function requires a data mover, which consists of a hardware component that moves the data, and an operating system-specific library function. The following table lists supported data movers and various properties for each.

*Figure 6: SDSoC Data Movers Table*

| SDSoC Data Mover | Vivado IP Data Mover | Accelerator IP Port Types | Transfer Size | Contiguous Memory Only |
|---|---|---|---|---|
| axi_lite | processing_system7 | register, axilite | | |
| axi_dma_simple | axi_dma | bram, ap_fifo, axis | < 8 MB | ✓ |
| axi_dma_sg | axi_dma | bram, ap_fifo, axis | | |
| axi_fifo | axi_fifo_mm_s | bram, ap_fifo, axis | (≤ 300 B) | |
| zero_copy | accelerator IP | aximm master | | ✓ |

X14762-052217

Scalar variables are always transferred over an AXI4-Lite bus interface with the `axi_lite` data mover. For array arguments, the data mover inference is based on transfer size, hardware function port mapping, and function call site information. The selection of data movers is a trade off between performance and resource, for example:

- The `axi_dma_simple` data mover is the most efficient bulk transfer engine, but only supports up to 8 MB transfers, so it is only for larger transfers.
- The `axi_fifo` data mover does not require as many hardware resources as the DMA, but due to its slower transfer rates, is preferred only for payloads of up to 300 bytes.
- The `axi_dma_sg` (scatter-gather DMA) data mover provides slower DMA performance and consumes more hardware resources but has fewer limitations, and in the absence of any pragma directives, is often the best default data mover.

,

You can override the data mover selection by inserting a pragma into program source immediately before the function declaration, for example,

```
#pragma SDS data data_mover(A:AXIDMA_SIMPLE)
```

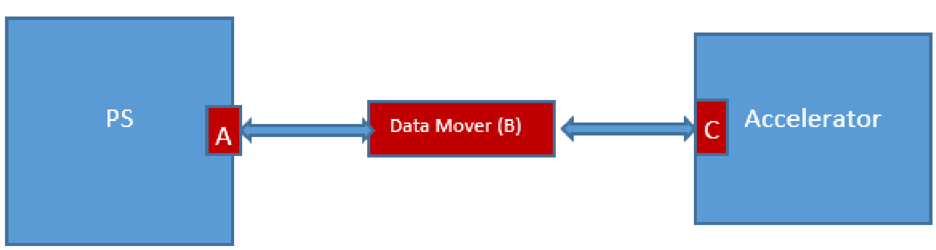
**NOTE:** *`#pragma SDS` is always treated as a rule, not a hint, so you must ensure that their use conforms with the data mover requirements in the preceding figure (SDSoC Data Movers Table).*

The data motion network in the SDSoC environment is made up of three components:

- The memory system ports on the PS (A)
- Data movers between the PS and accelerators as well as among accelerators (B)
- The hardware interface on an accelerator (C)

The following figure illustrates these three components.

*Figure 7: Data Motion Network Components*



Without any SDS pragma, the SDSoC environment generates the data motion network based on an analysis of the source code. However, the SDSoC environment also provides pragmas for you to guide the data motion network generation.

## System Port

A system port connects a data mover to the PS. It can be an ACP, AFI (corresponding to high-performance ports), or MIG (corresponding to a PL-based DDR memory controller) port on the Zynq Ultrascale+ MPSoC or Zynq-7000 AP SoC. The ACP port is a cache-coherent port and the cache coherency is maintained by the hardware. The AFI port is a non-cache-coherent port. Cache coherency (i.e. cache flushing and cache invalidation) is maintained by software if needed. Selecting between the ACP port versus the AFI port depends on the cache requirement of the transferred data.

The system port choice is dependent on the data's cache attribute and data size. If the data is allocated with `sds_alloc_non_cacheable()` or `sds_register_dmabuf()`, it is better to connect to the AFI port to avoid cache flushing/invalidation. If the data is allocated in other ways, it is better to connect to the ACP port for fast cache flushing/invalidation.

SDSoC compiler analyzes these memory attributes for the data transferred to and received from the accelerator, and connects data movers to the appropriate system port. However, if the user would like to override the compiler decision, or in some cases, the compiler is not able to do such analysis, the user can use the following pragma to specify the system port.

```
#pragma SDS data sys_port(arg:port)
```

Where `port` can be either `ACP`, `AFI`, or `MIG`.

The data size pragmas (`#pragma SDS data copy` and `#pragma SDS data zero_copy`) have been discussed previously. Notice the user must make sure the specified pragma is correct.

## Data Mover

The data mover transfers data between PS and accelerators, and among accelerators. SDSoC™ can generate various types of data movers based on the properties and size of the data being transferred.

**Scalar**

Scalar data is always transferred by the AXI_LITE data mover.

**Array**

SDSoC can generate AXIDMA_SG, AXIDMA_SIMPLE, AXIFIFO, zero_copy (accelerator-mastered AXI4 bus), or AXI_LITE data movers, depending on the memory attributes and data size of the array. For example, if the array is allocated using `malloc()`, the memory is not physically contiguous, and SDSoC generates a scatter-gather DMA (AXI_DMA_SG). However, if the data size is less than 300 bytes, AXI_FIFO is generated instead because the data transfer time is less than AXI_DMA_SG, and it occupies much less PL resource.

**Struct or Class**

The implementation of structs depends on how the struct is passed to the hardware—passed by value, passed by reference, or as an arrays of structs—and the type of datamover selected. The following table shows the various implementations.

**Table 1: Struct Implementations**

| Struct Pass Method | Default (no pragma) | #pragma SDS data zero_copy (arg) | #pragma SDS data zero_copy (arg[0:SIZE]) | #pragma SDS data copy (arg) | #pragma SDS data copy (arg[0:SIZE]) |
|---|---|---|---|---|---|
| pass by value (struct RGB arg) | Each field is flattened and passed individually as a scalar or an array. | This is not supported and will result in an error. | This is not supported and will result in an error. | The struct is packed into a single wide scalar. | Each field is flattened and passed individually as a scalar or an array.<br>The value of SIZE is ignored. |
| pass by pointer (struct RGB *arg) or reference (struct RGB &arg) | Each field is flattened and passed individually as a scalar or an array. | The struct is packed into a single wide scalar and transferred as a single value.<br>The data is transferred to the hardware accelerator via an AXI4 bus. | The struct is packed into a single wide scalar.<br>The number of data values transferred to the hardware accelerator via an AXI4 bus is defined by the value of SIZE. | The struct is packed into a single wide scalar. | The struct is packed into a single wide scalar.<br>The number of data values transferred to the hardware accelerator via an AXIDMA_SG or AXIDMA_SIMPLE is defined by the value of SIZE. |
| array of struct<br>(struct RGB arg[1024]) | Each struct element of the array is packed into a single wide scalar. | Each struct element of the array is packed into a single wide scalar.<br>The data is transferred to the hardware accelerator via an AXI4 bus. | Each struct element of the array is packed into a single wide scalar.<br>The data is transferred to the hardware accelerator via an AXI4 bus. | Each struct element of the array is packed into a single wide scalar.<br>The data is transferred to the hardware accelerator via a data mover such as AXIDMA_SG or AXIDMA_SIMPLE. | Each struct element of the array is packed into a single wide scalar.<br>The data is transferred to the hardware accelerator via a data mover such as AXIDMA_SG or AXIDMA_SIMPLE. |

| Struct Pass Method | Default (no pragma) | #pragma SDS data zero_copy (arg) | #pragma SDS data zero_copy (arg[0:SIZE]) | #pragma SDS data copy (arg) | #pragma SDS data copy (arg[0:SIZE]) |
|---|---|---|---|---|---|
| | | | The value of SIZE overrides the array size and determines the number of data values transferred to the accelerator. | | The value of SIZE overrides the array size and determines the number of data values transferred to the accelerator. |

The selection of which data mover to use for transferring an array is dependent on two attributes of the array: data size and physical memory contiguity. For example, if the memory size is 1 MB and not physically contiguous (allocated by `malloc()`), you should use AXIDMA_SG. The following table shows the applicability of these data movers.

**Table 2: Data Mover Selection**

| Data Mover | Physical Memory Contiguity | Data Size (bytes) |
|---|---|---|
| AXIDMA_SG | Either | > 300 |
| AXIDMA_Simple | Contiguous | < 8M |
| AXIFIFO | Non-contiguous | < 300 |

Normally, the SDSoC™ compiler analyzes the array that is transferred to the hardware accelerator for these two attributes, and selects the appropriate data mover accordingly. However, there are cases where such analysis is not possible. At that time, SDSoC issues a warning message and asks you to specify the memory attributes via SDS pragmas. An example of the message:

```
WARNING: [DMAnalysis 83-4492] Unable to determine the memory attributes
passed to rgb_data_in of function img_process at
C:/simple_sobel/src/main_app.c:84
```

The pragma to specify the memory attributes is:

```
#pragma SDS data mem_attribute(arg:contiguity)
```

Where `contiguity` can be either `PHYSICAL_CONTIGUOUS` or `NON_PHYSICAL_CONTIGUOUS`. The pragma to specify the data size is:

```
#pragma SDS data copy(arg[offset:size])
```

Where `size` can be a number or an arbitrary expression.

### *Zero Copy Data Mover*

As mentioned previously, the zero copy data mover is a special one because it covers both the accelerator interface and the data mover. The syntax of this pragma is:

```
#pragma SDS data zero_copy(arg[offset:size])
```

Where `[offset:size]` is optional, and only needed if data transfer size for an array cannot be determined at compile time.

By default, SDSoC assumes `copy` semantics for an array argument, meaning the data is explicitly copied from the PS to the accelerator via a data mover. When this `zero_copy` pragma is specified, SDSoC generates an AXI-Master interface for the specified argument on the accelerator, which grabs the data from the PS as specified in the accelerator code.

To use the `zero_copy` pragma, the memory corresponding to the array has to be physically contiguous, that is allocated with `sds_alloc`.

## Accelerator Interface

The accelerator interface generated in SDSoC™ depends on the data type of the argument.

**Scalar**

For a scalar argument, the register interface is generated to pass in and/or out of the accelerator.

**Arrays**

The hardware interface on an accelerator for transferring an array can be either a RAM interface or a streaming interface, depending on how the accelerator accesses the data in the array.

The RAM interface allows the data to be accessed randomly within the accelerator; however, it requires the entire array to be transferred to the accelerator before any memory accesses can happen within the accelerator. Moreover, the use of this interface requires BRAM resources on the accelerator side to store the array.

The streaming interface, on the other hand, does not require memory to store the whole array, it allows the accelerator to pipeline the processing of array elements, i.e., the accelerator can start processing a new array element while the previous ones are still being processed. However, the streaming interface requires the accelerator to access the array in a strict sequential order, and the amount of data transferred must be the same as the accelerator expects.

SDSoC, by default, will generate the RAM interface for an array; however, SDSoC provides pragmas to direct it to generate the streaming interface.

**struct or class**

The implementation of structs depends on how the struct is passed to the hardware—passed by value, passed by reference, or as an arrays of structs—and the type of datamover selected. The Struct Implementations table in Data Mover shows the various implementations.

The following SDS pragma can be used to guide the interface generation for the accelerator.

```
#pragma SDS data access_pattern(arg:pattern)
```

Where "pattern" can be either "RANDOM" or "SEQUENTIAL", and "arg" can be an array argument name of the accelerator function.

If an array argument's access pattern is specified as "RANDOM", a RAM interface is generated. If it is specified as “SEQUENTIAL”, a streaming interface is generated. Several notes regarding this pragma:

- The default access pattern for an array argument is “RANDOM”.
- The specified access pattern must be consistent with the accelerator function's behavior. For “SEQUENTIAL” access patterns, the function must access every array element in a strict sequential order.
- This pragma only applies to arguments without the “zero_copy” pragma. This will be detailed later.

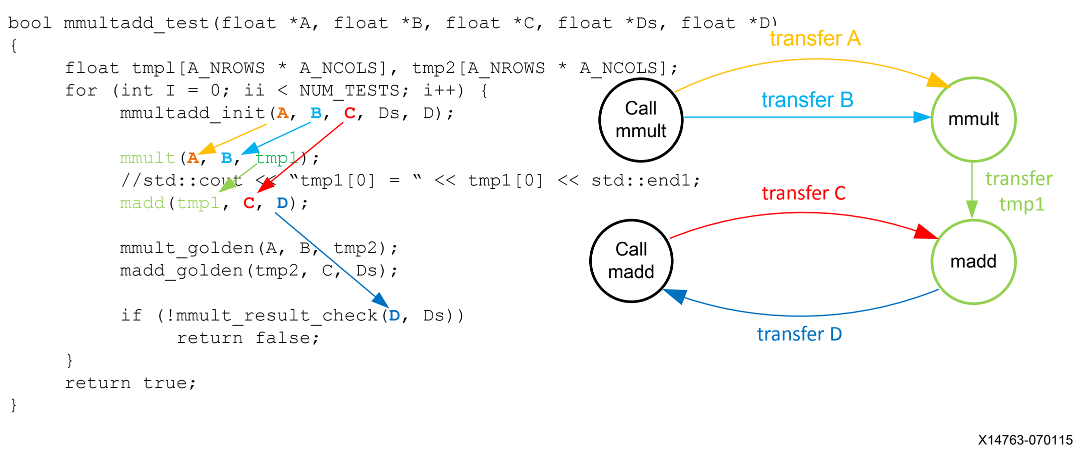
# Increasing System Parallelism and Concurrency

Increasing the level of concurrent execution is a standard way to increase overall system performance, and increasing the level of parallel execution is a standard way to increase concurrency. Programmable logic is well-suited to implement architectures with application-specific accelerators that run concurrently, especially communicating through flow-controlled streams that synchronize between data producers and consumers.

In the SDSoC environment, you influence the macro-architecture parallelism at the function and data mover level, and the micro-architecture parallelism within hardware accelerators. By understanding how the `sdscc` system compiler infers system connectivity and data movers, you can structure application code and apply pragmas as needed to control hardware connectivity between accelerators and software, data mover selection, number of accelerator instances for a given hardware function, and task level software control. You can control the micro-architecture parallelism, concurrency, and throughput for hardware functions within Vivado HLS or within the IPs you incorporate as C-callable/linkable libraries.

At the system level, the `sdscc` compiler chains together hardware functions when the data flow between them does not require transferring arguments out of programmable logic and back to system memory. For example, consider the code in the following figure, where `mmult` and `madd` functions have been selected for hardware.

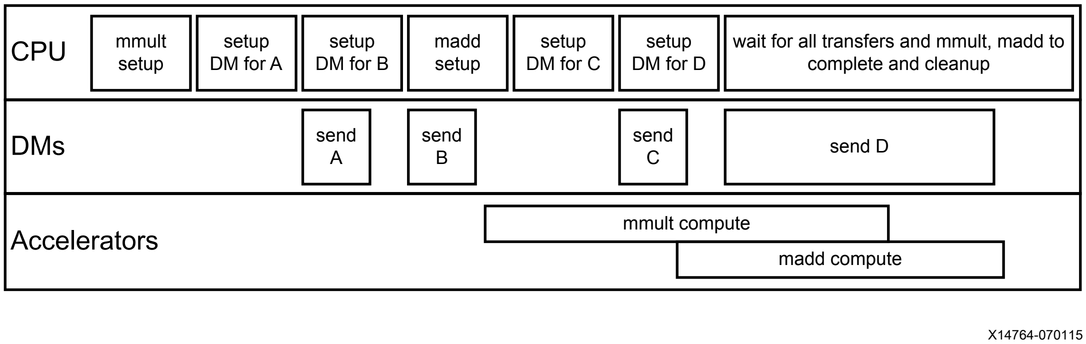
***Figure 8: Hardware /Software Connectivity with Direct Connection***




Because the intermediate array variable `tmp1` is used only to pass data between the two hardware functions, the `sdscc` system compiler chains the two functions together in hardware with a direct connection between them.

It is instructive to consider a time line for the calls to hardware as shown in the following figure.

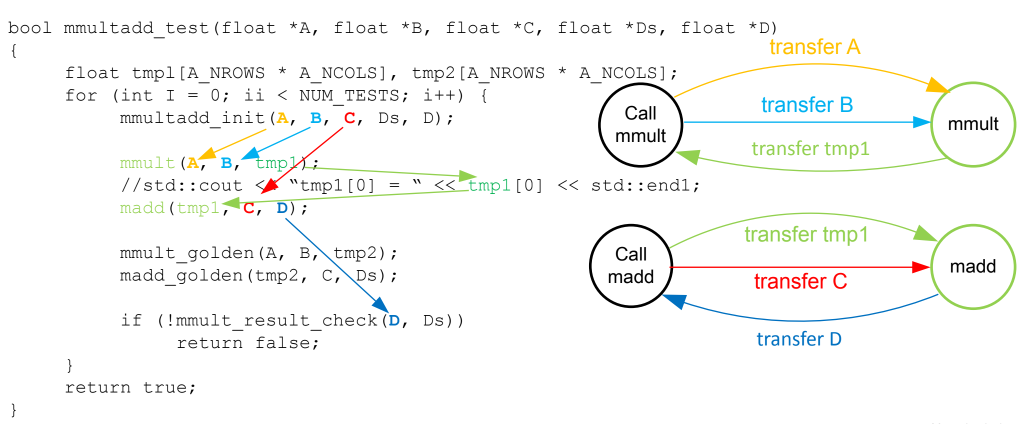
***Figure 9: Timeline for mmult/madd Function Calls***




The program preserves the original program semantics, but instead of the standard ARM procedure calling sequence, each hardware function call is broken into multiple phases involving setup, execution, and cleanup, both for the data movers (DM) and the accelerators. The CPU in turn sets up each hardware function (that is, the underlying IP control interface) and the data transfers for the function call with non-blocking APIs, and then waits for all calls and transfers to complete. In the example shown in the diagram, the mmult and madd functions run concurrently whenever their inputs become available. The ensemble of function calls is orchestrated in the compiled program by control code automatically generated by `sdscc` according to the program, data mover, and accelerator structure.

In general, it is impossible for the `sdscc` compiler to determine side-effects of function calls in your application code (for example, `sdscc` may have no access to source code for functions within linked libraries), so any intermediate access of a variable occurring lexically between hardware function calls requires the compiler to transfer data back to memory. So for example, an injudicious simple change to uncomment the debug print statement (in the "wrong place") as shown in the figure below, can result in a significantly different data transfer graph and consequently, an entirely different generated system and application performance.

***Figure 10: Hardware/Software Connectivity with Broken Direct Connection***



A program can invoke a single hardware function from multiple call sites. In this case, the `sdscc` compiler behaves as follows. If any of the function calls results in "direct connection" data flow, then `sdscc` creates an instance of the hardware function that services every similar direct connection, and an instance of the hardware function that services the remaining calls between memory ("software") and programmable logic.

Structuring your application code with "direct connection" data flow between hardware functions is one of the best ways to achieve high performance in programmable logic. You can create deep pipelines of accelerators connected with data streams, increasing the opportunity for concurrent execution.

There is another way in which you can increase parallelism and concurrency using the `sdscc` compiler. You can direct the compiler to create multiple instances of a hardware function by inserting the following pragma immediately preceding a call to the function.

```
#pragma SDS resource(<id>) // <id> a non-negative integer
```

This pragma creates a hardware instance that is referenced by `<id>`.

A simple code snippet that creates two instances of a hardware function `mmult` is as follows.

```
{
#pragma SDS resource(1)
 mmult(A, B, C); // instance 1
#pragma SDS resource(2)
 mmult(D, E, F); // instance 2
}
```

The `async` mechanism gives the programmer ability to handle the "hardware threads" explicitly to achieve very high levels of parallelism and concurrency, but like any explicit multi-threaded programming model, requires careful attention to synchronization details to avoid non-deterministic behavior or deadlocks.

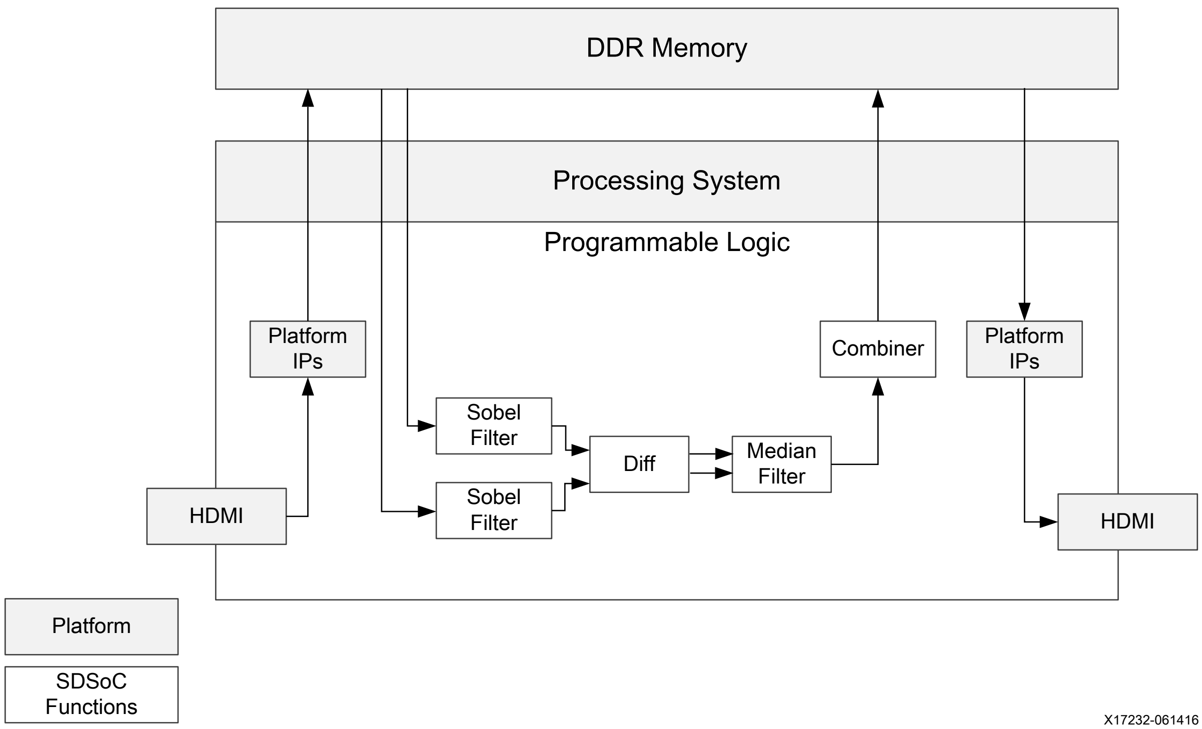
# Using External I/O

Hardware accelerators generated in the SDSoC™ environment can communicate with system inputs and outputs either directly through hardware connections, or though memory buffers (e.g., a frame buffer). Examples of system I/O include analog-to-digital and digital-to-analog converters, image, radar, LiDAR, and ultrasonic sensors, and HDMI™ multimedia streams. A platform exports stream connections in hardware that are accessed in software by calling plaform library functions as described in the following sections. Direct hardware connections are implemented over AXI4-Stream channels, and connections to memory buffers are realized through function calls implemented by the standard data movers supported in the SDSoC Environment. For information and examples that show how to create SDSoC platforms, refer to *SDSoC Environment Platform Development Guide* (UG1146).

## Accessing External I/O via Memory Buffers

This section uses the motion-detect ZC702 + HDMI IO FMC or ZC706 + HDMI IO FMC platform found on the SDSoC Downloads Page. The figure below shows how the design example is configured. The preconfigured SDSoC platform is responsible for the HDMI data transfer to external memory. The application must call the platform interfaces to process the data from the frame buffer in DDR memory.

***Figure 11: Motion Detect Design Configuration***




The SDSoC environment accesses the external frame buffer through an accelerator interface to the platform. The `zc702_hdmi` platform provides a software interface to access the video frame buffer through the Video4Linux2 (V4L2) API. The V4L2 framework provides an API accessing a collection of device drivers supporting real-time video capture in Linux. For the application developer, this API is the platform I/O entry point. In the `motion_demo_processing` example, the following code snippet from `m2m_sw_pipeline.c` demonstrates the function call interface.

```
extern void motion_demo_processing(unsigned short int *prev_buffer,
            unsigned short int *in_buffer,
            unsigned short int *out_buffer,
            int fps_enable,
            int height, int width, int stride);
.
.
.
unsigned short *out_ptr = v_pipe->drm.d_buff[buf_next->index].drm_buff;
unsigned short *in_ptr1 = buf_prev->v412_buff;
unsigned short *in_ptr2 = buf_next->v412_buff;
v_pipe->events[PROCESS_IN].counter_val++;
```

```
motion_demo_processing(in_ptr1, in_ptr2, out_ptr,
                        v_pipe->fps_enable,
                        (int)m2m_sw_stream_handle.video_in.format.height,
                        (int)m2m_sw_stream_handle.video_in.format.width,

(int)m2m_sw_stream_handle.video_in.format.bytesperline/2);
```

The application accesses this API in `motion_detect.c`, where `motion_demo_procesing` is defined and called by the `img_process` function.

```
void motion_demo_processing(unsigned short int *prev_buffer,
                             unsigned short int *in_buffer,
                             unsigned short int *out_buffer,
                             int fps_enable,
                             int height, int width, int stride)
{
      int param0=0, param1=1, param2=2;

      img_process(prev_buffer, in_buffer, out_buffer, height, width,
stride);
}
```

Finally, `img_process` calls the various filters and transforms to process the data.

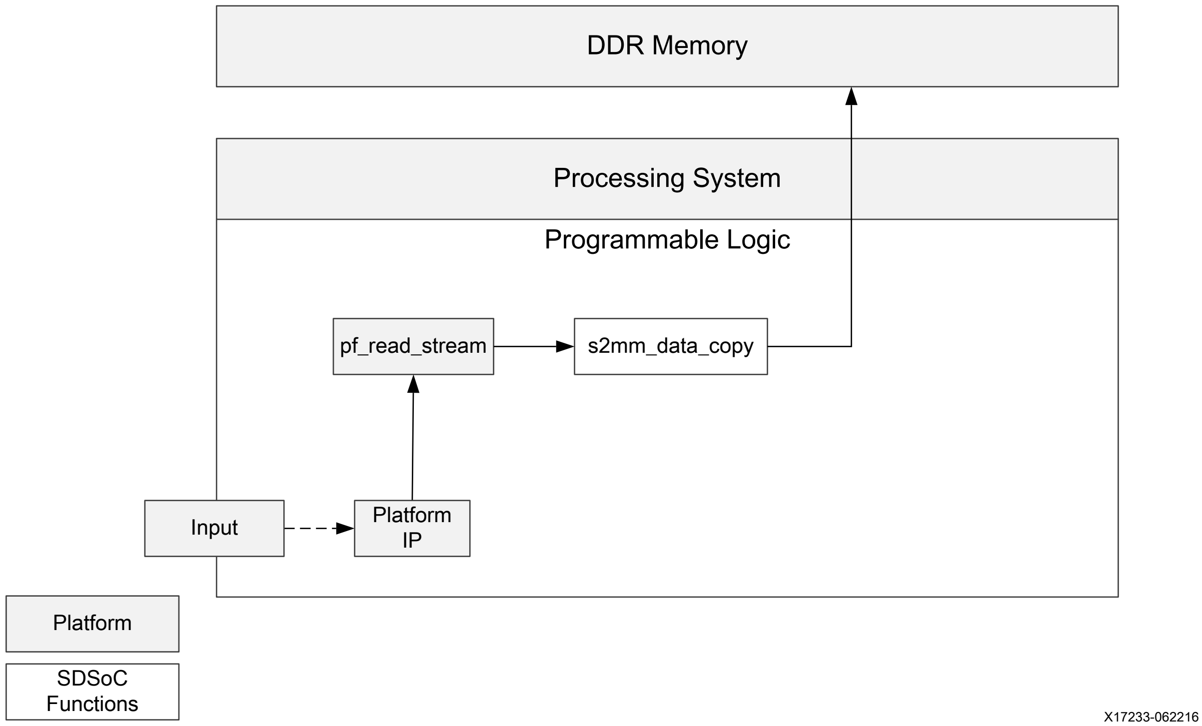
```
void img_process(unsigned short int *frame_prev,
                  unsigned short int *frame_curr,
                  unsigned short int *frame_out,
                  int param0, int param1, int param2)
{
...
}
```

By using a platform API to access the frame buffers, the application developer does not program at the driver level to process the video frames. You can find the platform used for the code snippets on the SDSoC Downloads Page with the name ZC702[ZC706] + HDMI IO FMC. To access the project in the SDSoC environment, create a new project, name the project, and select **Add Custom Platform**. From the **Target Platform** menu, select the downloaded platform named **zc702[zc706]_trd**, click **Next**, and use the template named Motion Detect.

## Accessing External I/O via Direct Hardware Connections

Whereas the previous example demonstrated how applications can access system I/O through memory buffers, a platform can also provide direct hardware connectivity to hardware accelerators within an application generated in the SDSoC Environment. The following figure shows how a function `s2mm_data_copy` communicates to the platform via an AXI4-Stream channel, and writes to DDR memory using the zero_copy datamover (implemented as an AXI4 master interface). This design template is called `Unpacketized AXI4-Stream to DDR` design example in the `samples/platforms/zc702_axis_io` platform (a similar design for the Zybo board is in `samples/xc7z010/zybo_axis_io`).

*Figure 12: Unpacketized AXI-MM DataMover Design*




In this example, the `zc702_axis_io` platform proxies actual I/O by providing a free-running binary counter (labeled Platform IP in the diagram) running at 50 MHz, connected to an AXI4-Stream Data FIFO IP block that exports an AXI4-Stream master interface to the platform clocked at the data motion clock (which might differ from the 50 MHz input clock).

The direct I/O software interface can be found in the `zc702_axis_io.h` header file located in the SDSoC install directory under `samples/platforms/zc702/aarch32-linux/include`.

```
#pragma SDS data access_pattern(rbuf:SEQUENTIAL)
void pf_read_stream(unsigned *rbuf);
```

In the code snippet below, the application defines a direct signal path from the platform input to a hardware function before transferring the output to memory.

```
// This function's data flow defines the accelerator network
void s2mm_data_copy_wrapper(unsigned *buf)
{
     unsigned rbuf0[1];
     pf_read_stream(rbuf0);
     s2mm_data_copy(rbuf0,buf);
}
```

The platform library provides the `pf_read_stream` function that the sds++ linker maps onto the hardware stream port. Because the only use of the `rbuf0` output is the input to the `s2mm_data_copy` function, the linker creates a direct hardware connection over an AXI4-Stream channel. Because the `s2mm_data_copy` function transfers `buf` using the zero_copy data mover, the buffer must be allocated in physically contiguous memory using `sds_alloc`, and released using `sds_free`.

```
int main()
{
     unsigned *bufs[NUM_BUFFERS];

     for(int i=0; i<NUM_BUFFERS; i++) {
          bufs[i] = (unsigned*) sds_alloc(BUF_SIZE * sizeof(unsigned));
     }
     // call accelerator data path and check result
     for(int i=0; i<NUM_BUFFERS; i++) {
          sds_free(bufs[i]);
     }
     return 0;
}
```

A tutorial of how to use this example design is provided in *SDSoC Environment Tutorial: Introduction* (UG1028).

A detailed tutorial on creating a platform using AXI4-Stream to write memory directly can be found in *SDSoC Environment Platform Development Guide* (UG1146).

# Improving Hardware Function Parallelism

This section provides a concise introduction to writing efficient code that can be cross-compiled into programmable logic.

The SDSoC environment employs Vivado HLS as a programmable logic cross-compiler to transform C/C++ functions into hardware. By applying the principles described in this section, you can dramatically increase the performance of the synthesized functions, which can lead to significant increases in overall system performance for your application.

## Top-Level Hardware Function Guidelines

This section describes coding guidelines to ensure that a Vivado HLS hardware function has a consistent interface with object code generated by the ARM GNU toolchain.

### *Use Standard C99 Data Types for Top-Level Hardware Function Arguments*

1. Avoid using arrays of `bool`. An array of `bool` has different memory layout between ARM `GCC` and Vivado® HLS.
2. Avoid using `hls::stream` at the hardware function top-level interface. This data type helps the HLS compiler synthesize efficient logic within a hardware function but does not make sense for application software.

### *Omit HLS Interface Directives for Top-Level Hardware Function Arguments*

Although supported, a top-level hardware function should not in general contain `HLS interface` pragmas. The `sds++` compiler automatically generates appropriate HLS interface directives. There are two SDSoC environment pragmas you can specify for a top-level hardware function to guide the SDSoC environment to generate the desired HLS interface directives.

`#pragma SDS data zero_copy()` can be used to generate a shared memory interface implemented as an AXI master interface in hardware.
`#pragma SDS data access_pattern(argument:SEQUENTIAL)` can be used to generate a streaming interface implemented as a FIFO interface in hardware.

---

**IMPORTANT:** *If you specify the interface using `#pragma HLS interface` for a top-level function argument, the SDSoC environment does not generate a HLS interface directive for that argument, and it is your responsibility to ensure that the generated hardware interface is consistent with all other function argument hardware interfaces. Because a function with incompatible HLS interface types can result in cryptic `sdscc` error messages, it is strongly recommended (though not absolutely mandatory) that you omit `HLS interface` pragmas.*

---

## Optimization Guidelines

This section documents several fundamental HLS optimization techniques to enhance hardware function performance. These techniques are: function inlining, loop and function pipelining, loop unrolling, increasing local memory bandwidth and streaming data flow between loops and functions. For more information, see *SDSoC Environment Optimization Guide* (UG1235).

- Function Inlining
- Loop Pipelining and Loop Unrolling
- Increasing Local Memory Bandwidth
- Data Flow Pipelining

### *Function Inlining*

Similar to function inlining of software functions, it can be beneficial to inline hardware functions.

Function inlining replaces a function call by substituting a copy of the function body after resolving the actual and formal arguments. After that, the inlined function is dissolved and no longer appears as a separate level of hierarchy. Function inlining allows operations within the inlined function be optimized more effectively with surrounding operations, thus improving the overall latency or the initiation interval for a loop.

To inline a function, put `#pragma HLS inline` at the beginning of the body of the desired function. The following code snippet directs Vivado HLS to inline the `mmult_kernel` function:

```
void mmult_kernel(float in_A[A_NROWS][A_NCOLS],
                  float in_B[A_NCOLS][B_NCOLS],
                  float out_C[A_NROWS][B_NCOLS])
{
#pragma HLS INLINE
     int index_a, index_b, index_d;
     // rest of code body omitted
}
```

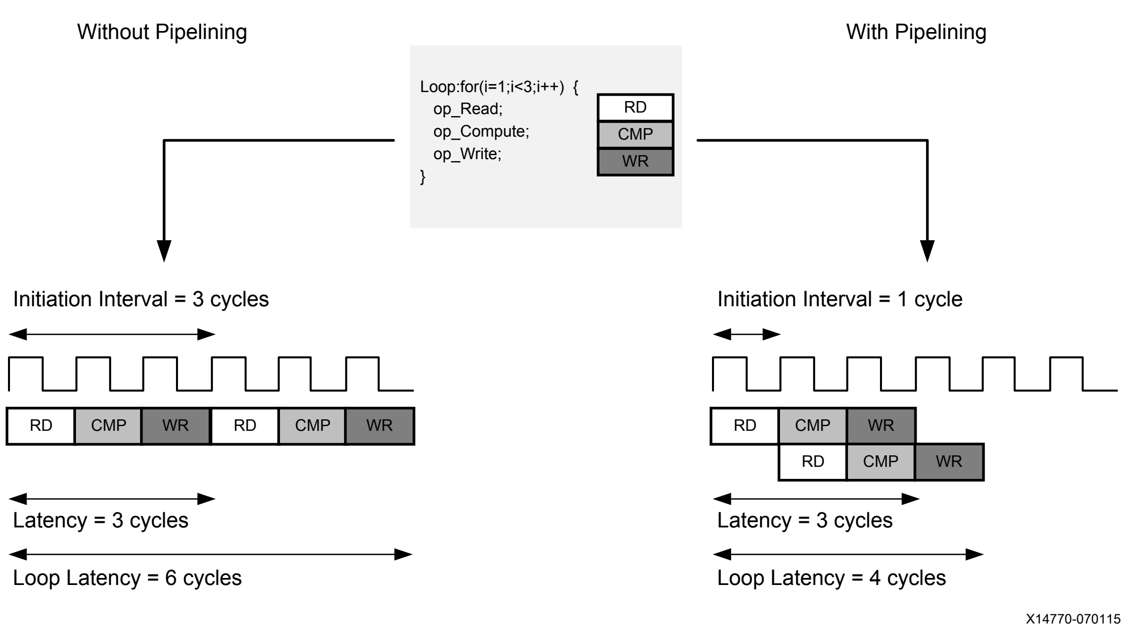
### *Loop Pipelining and Loop Unrolling*

Both loop pipelining and loop unrolling improve the hardware function's performance by exploiting the parallelism between loop iterations. The basic concepts of loop pipelining and loop unrolling and example codes to apply these techniques are shown and the limiting factors to achieve optimal performance using these techniques are discussed.

### *Loop Pipelining*

In sequential languages such as C/C++, the operations in a loop are executed sequentially and the next iteration of the loop can only begin when the last operation in the current loop iteration is complete. Loop pipelining allows the operations in a loop to be implemented in a concurrent manner as shown in the following figure.

***Figure 13: Loop Pipelining***



As shown in the above figure, without pipelining, there are three clock cycles between the two `RD` operations and it requires six clock cycles for the entire loop to finish. However, with pipelining, there is only one clock cycle between the two `RD` operations and it requires four clock cycles for the entire loop to finish, that is, the next iteration of the loop can start before the current iteration is finished.

An important term for loop pipelining is called *Initiation Interval (II)*, which is the number of clock cycles between the start times of consecutive loop iterations. In the above figure, the Initiation Interval (II) is one because there is only one clock cycle between the start times of consecutive loop iterations.

To pipeline a loop, put `#pragma HLS pipeline` at the beginning of the loop body, as illustrated in the following code snippet. Vivado HLS tries to pipeline the loop with minimum *Initiation Interval*.

```
for (index_a = 0; index_a < A_NROWS; index_a++) {
     for (index_b = 0; index_b < B_NCOLS; index_b++) {
#pragma HLS PIPELINE II=1
          float result = 0;
          for (index_d = 0; index_d < A_NCOLS; index_d++) {
              float product_term = in_A[index_a][index_d] *
in_B[index_d][index_b];
              result += product_term;
          }
          out_C[index_a * B_NCOLS + index_b] = result;
     }
}
```

### *Loop Unrolling*

Loop unrolling is another technique to exploit parallelism between loop iterations. It creates multiple copies of the loop body and adjust the loop iteration counter accordingly. The following code snippet shows a normal rolled loop:

```
int sum = 0;
for(int i = 0; i < 10; i++) {
    sum += a[i];
}
```

After the loop is unrolled by a factor of 2, the loop becomes:

```
int sum = 0;
for(int i = 0; i < 10; i+=2) {
    sum += a[i];
    sum += a[i+1];
}
```

So unrolling a loop by a factor of N basically creates N copies of the loop body, the loop variable referenced by each copy is updated accordingly ( such as the `a[i+1]` in the above code snippet ), and the loop iteration counter is also updated accordingly ( such as the `i+=2` in the above code snippet ).

Loop unrolling creates more operations in each loop iteration, so that Vivado HLS can exploit more parallelism among these operations. More parallelism means more throughput and higher system performance. If the factor N is less than the total number of loop iterations (10 in the example above), it is called a "partial unroll". If the factor N is the same as the number of loop iterations, it is called a "full unroll". Obviously, "full unroll" requires the loop bounds be known at compile time but exposes the most parallelism.

To unroll a loop, simply put `#pragma HLS unroll [factor=N]` at the beginning of the desired loop. Without the optional `factor=N`, the loop will be fully unrolled.

```
int sum = 0;
for(int i = 0; i < 10; i++) {
#pragma HLS unroll factor=2
    sum += a[i];
}
```

### *Factors Limiting the Parallelism Achieved by Loop Pipelining and Loop Unrolling*

Both loop pipelining and loop unrolling exploit the parallelism between loop iterations. However, parallelism between loop iterations is limited by two main factors: one is the data dependencies between loop iterations, the other is the number of available hardware resources.

A data dependence from an operation in one iteration to another operation in a subsequent iteration is called a loop-carried dependence. It implies that the operation in the subsequent iteration cannot start until the operation in the current iteration has finished computing the data input for the operation in subsequent iteration. Loop-carried dependencies fundamentally limit the initiation interval that can be achieved using loop pipelining and the parallelism that can be exploited using loop unrolling.

The following example demonstrates loop-carried dependencies among operations producing and consuming variables a and b.

```
while (a != b) {
    if (a > b)
        a -= b;
    else
        b -= a;
}
```

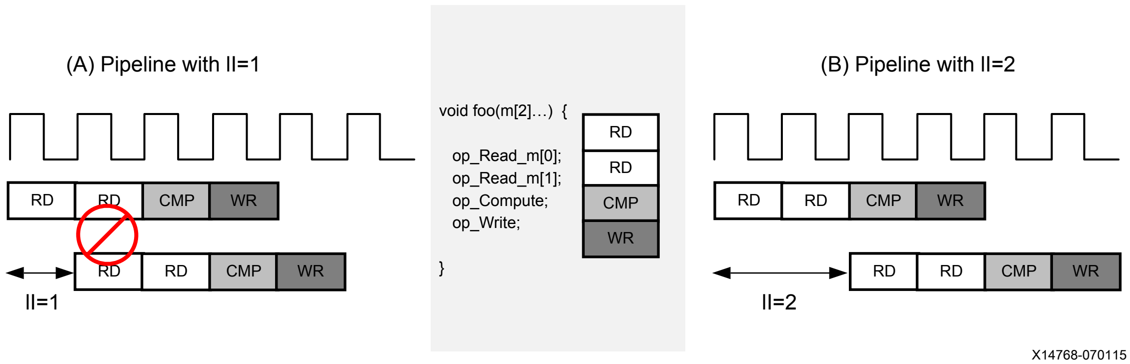
Obviously, operations in the next iteration of this loop can not start until the current iteration has calculated and updated the values of a and b. Array accesses are a common source of loop-carried dependencies, as shown in the following example:

```
for (i = 1; i < N; i++)
    mem[i] = mem[i-1] + i;
```

In this case, the next iteration of the loop must wait until the current iteration updates the content of the array. In case of loop pipelining, the minimum Initiation Interval is the total number of clock cycles required for the memory read, the add operation, and the memory write.

Another performance limiting factor for loop pipelining and loop unrolling is the number of available hardware resources. The following figure shows an example the issues created by resource limitations, which in this case prevents the loop to be pipelined with an initiation interval of 1.

***Figure 14: Resource Contention***




In this example, if the loop is pipelined with an initiation interval of one, there are two read operations. If the memory has only a single port, then the two read operations cannot be executed simultaneously and must be executed in two cycles. So the minimal initiation interval can only be two, as shown in part (B) of the figure. The same can happen with other hardware resources. For example, if the `op_compute` is implemented with a DSP core which cannot accept new inputs every cycle, and there is only one such DSP core. Then `op_compute` cannot be issued to the DSP core each cycle, and an initiation interval of one is not possible.

### *Increasing Local Memory Bandwidth*

This section shows several ways provided by Vivado HLS to increase local memory bandwidth, which can be used together with loop pipelining and loop unrolling to improve system performance.

Arrays are intuitive and useful constructs in C/C++ programs. They allow the algorithm be easily captured and understood. In Vivado HLS, each array is by default implemented with a single port memory resource. However, such memory implementation may not be the most ideal memory architecture for performance oriented programs. At the end of Loop Pipelining and Loop Unrolling, an example of resource contention caused by limited memory ports is shown.

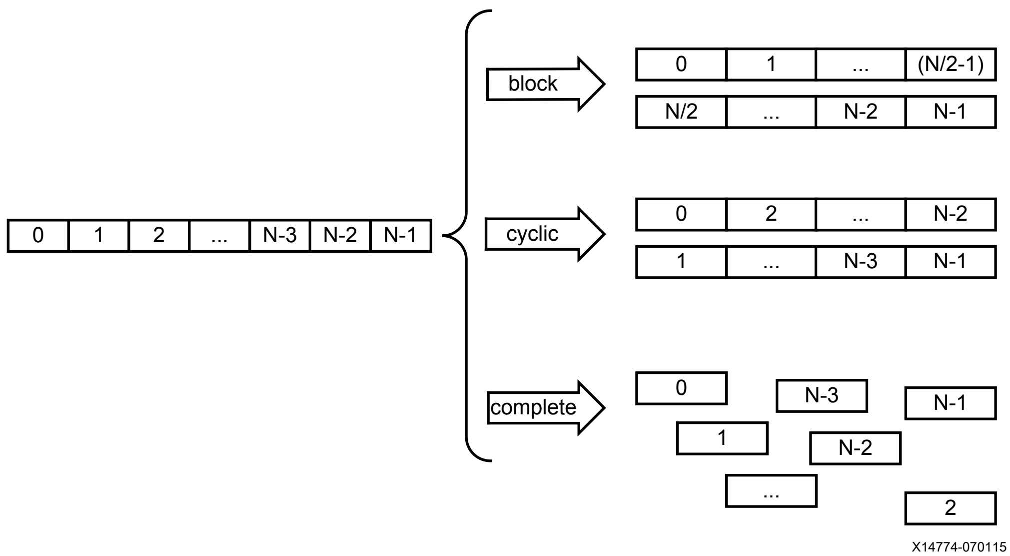
### *Array Partitioning*

Arrays can be partitioned into smaller arrays. Physical implementation of memories have only a limited number of read ports and write ports, which can limit the throughput of a load/store intensive algorithm. The memory bandwidth can sometimes be improved by splitting up the original array (implemented as a single memory resource) into multiple smaller arrays (implemented as multiple memories), effectively increasing the number of load/store ports.

Vivado HLS provides three types of array partitioning, as shown in the following figure.

1. *block*: The original array is split into equally sized blocks of consecutive elements of the original array.
2. *cyclic*: The original array is split into equally sized blocks interleaving the elements of the original array.
3. *complete*: The default operation is to split the array into its individual elements. This corresponds to implementing an array as a collection of registers rather than as a memory.

***Figure 15: Array Partitioning***



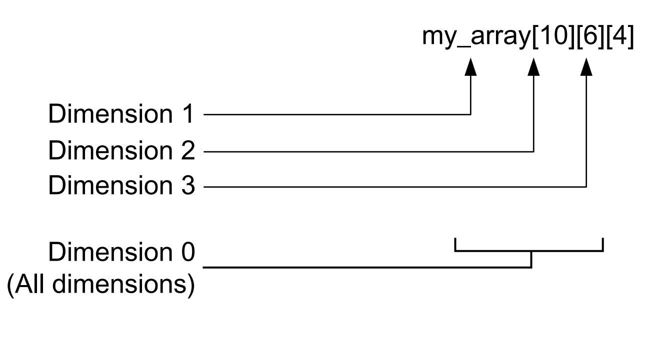

To partition an array in Vivado HLS, insert this in the hardware function source code:

```
#pragma HLS array_partition variable=<variable> <block, cyclic, complete>
factor=<int> dim=<int>
```

For *block* and *cyclic* partitioning, the `factor` option can be used to specify the number of arrays which are created. In the figure above, a factor of two is used, dividing the array into two smaller arrays. If the number of elements in the array is not an integer multiple of the factor, the last array will have fewer than average elements.

When partitioning multi-dimensional arrays, the `dim` option can be used to specify which dimension is partitioned. The following figure shows an example of partitioning different dimensions of a multi-dimensional array.

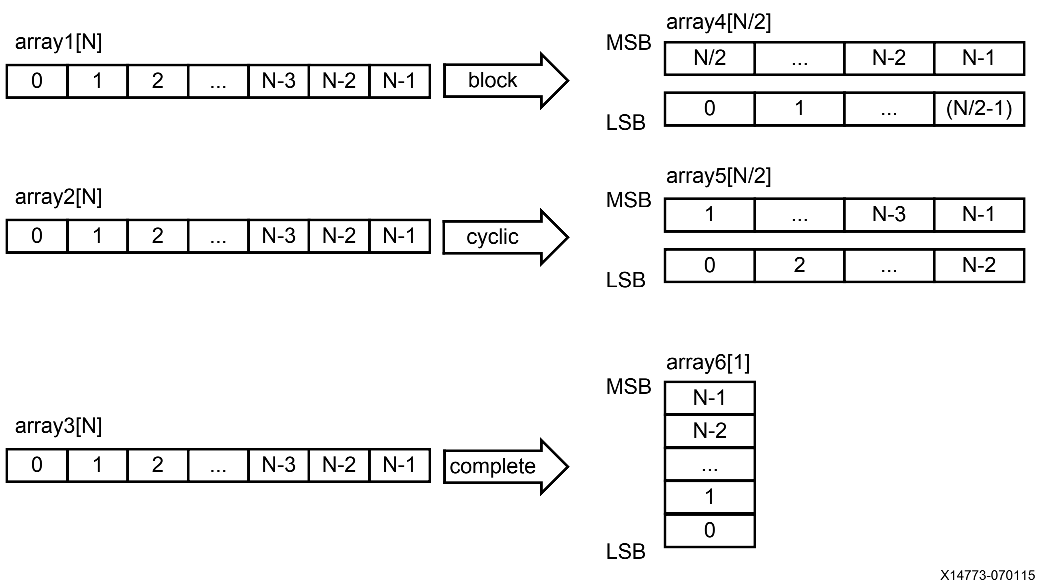
***Figure 16: Multi-dimension Array Partitioning***



### Array Reshaping

Arrays can also be reshaped to increase the memory bandwidth. Reshaping takes different elements from a dimension in the original array, and combines them into a single wider element. Array reshaping is similar to array partitioning, but instead of partitioning into multiple arrays, it widens array elements. The following figure illustrates the concept of array reshaping.

***Figure 17: Array Reshaping***




To use array reshaping in Vivado HLS, insert this in the hardware function source code:

```
#pragma HLS array_reshape variable=<variable> <block, cyclic, complete>
factor=<int> dim=<int>
```

The options have the same meaning as the array partition pragma.

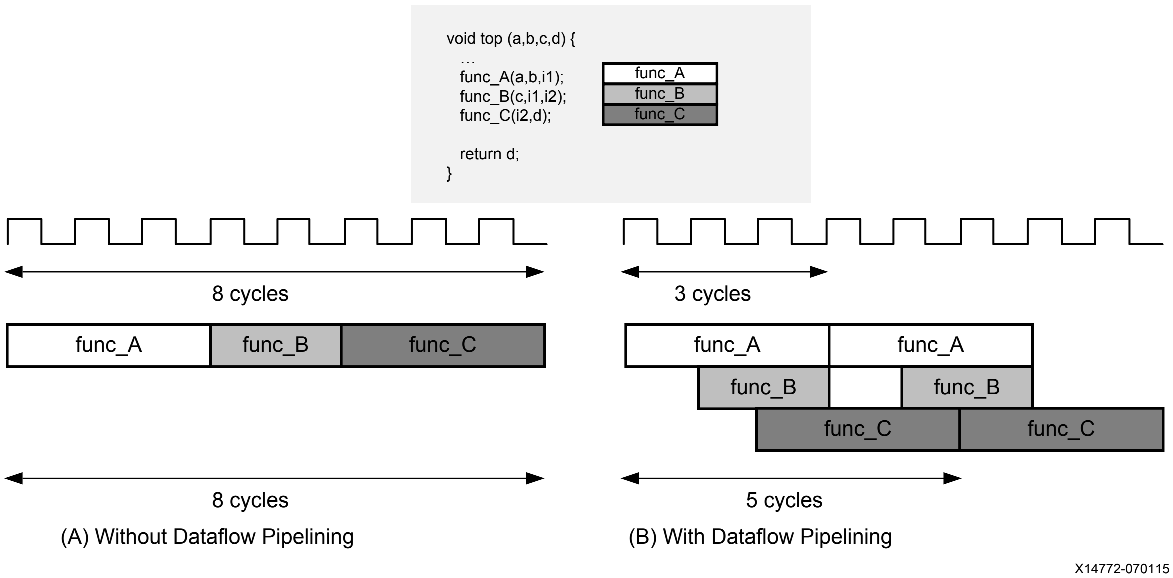
### Data Flow Pipelining

The previously discussed optimization techniques are all "fine grain" parallelizing optimizations at the level of operators, such as multiplier, adder, and memory load/store operations. These techniques optimize the parallelism between these operators. Data flow pipelining on the other hand, exploits the "coarse grain" parallelism at the level of functions and loops. Data flow pipelining can increase the concurrency between functions and loops.

### *Function Data Flow Pipelining*

The default behavior for a series of function calls in Vivado HLS is to complete a function before starting the next function. Part (A) in the following figure shows the latency without function data flow pipelining. Assuming it takes eight cycles for the three functions to complete, the code requires eight cycles before a new input can be processed by "func_A" and also eight cycles before an output is written by "func_C" (assume the output is written at the end of "func_C").

***Figure 18: Function Data Flow Pipelining***




An example execution with data flow pipelining is shown in the part (B) of the figure above. Assuming the execution of `func_A` takes three cycles, `func_A` can begin processing a new input every three clock cycles rather than waiting for all the three functions to complete, resulting in increased throughput, The complete execution to produce an output then requires only five clock cycles, resulting in shorter overall latency.

Vivado HLS implements function data flow pipelining by inserting "channels" between the functions. These channels are implemented as either ping-pong buffers or FIFOs, depending on the access patterns of the producer and the consumer of the data.

- If a function parameter (producer or consumer) is an array, the corresponding channel is implemented as a multi-buffer using standard memory accesses (with associated address and control signals).
- For scalar, pointer and reference parameters as well as the function return, the channel is implemented as a FIFO, which uses less hardware resources (no address generation) but requires that the data is accessed sequentially.

To use function data flow pipelining, put `#pragma HLS dataflow` where the data flow optimization is desired. The following code snippet shows an example:

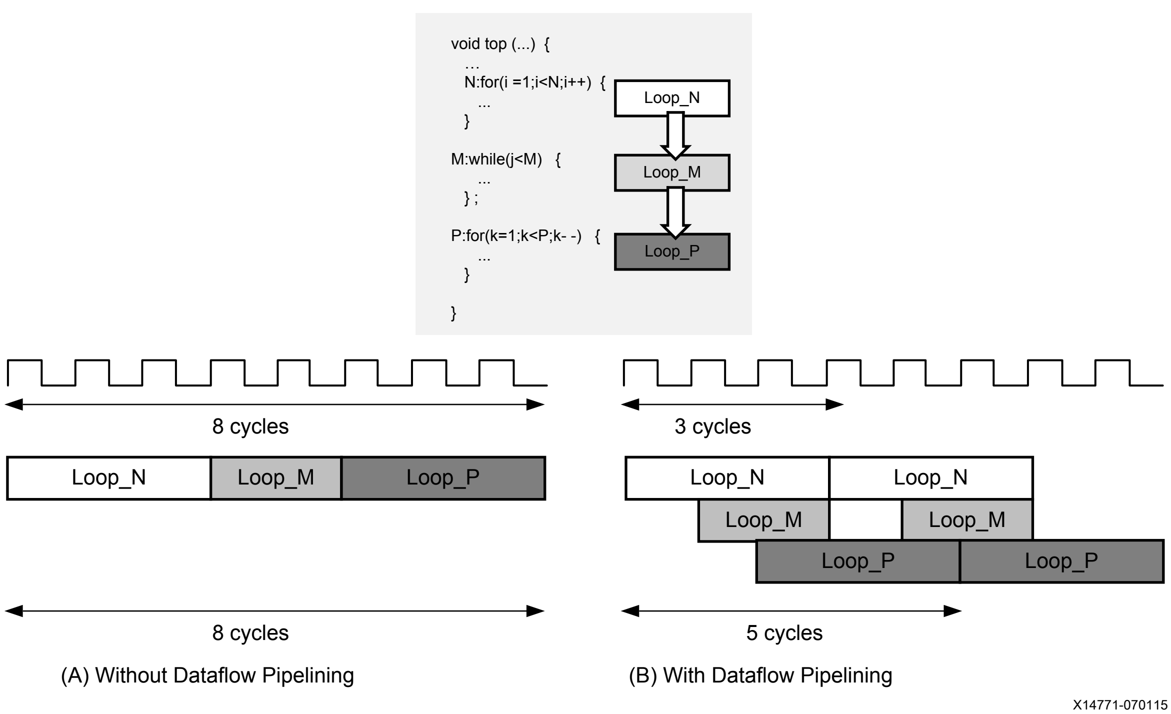
```
void top(a, b, c, d) {
#pragma HLS dataflow
    func_A(a, b, i1);
    func_B(c, i1, i2);
    func_C(i2, d);
}
```

### *Loop Data Flow Pipelining*

Data flow pipelining can also be applied to loops in similar manner as it can be applied to functions. It enables a sequence of loops, normally executed sequentially, to execute concurrently. Data flow pipelining should be applied to a function, loop or region which contains either all function or all loops: do not apply on a scope which contains a mixture of loops and functions.

The following figure shows the advantages data flow pipelining can produce when applied to loops. Without data flow pipelining, loop N must execute and complete all iterations before loop M can begin. The same applies to the relationship between loops M and P. In this example, it is eight cycles before loop N can start processing the next value and eight cycles before an output is written (assuming the output is written when loop P finishes).

***Figure 19: Loop Data Flow Pipelining***



With data flow pipelining, these loops can operate concurrently. An example execution with data flow pipelining is shown in part (B) of the figure above. Assuming the loop M takes three cycles to execute, the code can accept new inputs every three cycles. Similarly, it can produce an output value every five cycles, using the same hardware resources. Vivado HLS automatically inserts channels between the loops to ensure data can flow asynchronously from one loop to the next. As with data flow pipelining, the channels between the loops are implemented either as multi-buffers or FIFOs.

To use loop data flow pipelining, put `#pragma HLS dataflow` where the data flow optimization is desired.

## Using Vivado Design Suite HLS Libraries

This section describes how to use Vivado HLS libraries with the SDSoC environment.

Vivado® High-Level Synthesis (HLS) libraries are provided as source code with the Vivado HLS installation in the SDSoC environment. Consequently, you can use these libraries as you would any other source code that you plan to cross-compile for programmable logic using Vivado HLS. In particular, you must ensure that the source code conforms to the rules described in Hardware Function Argument Types, which might require you to provide a C/C++ wrapper function to ensure the functions export a software interface to your application.

The synthesizeable FIR example template for all basic platforms in the SDSoC IDE provides an example that uses an HLS library. You can find several additional code examples that employ HLS libraries in the `samples/hls_lib` directory. For example, `samples/hls_lib/hls_math` contains an example to implement and use a square root function.

The file `my_sqrt.h` contains:

```
#ifndef _MY_SQRT_H_
#define _MY_SQRT_H_

#ifdef __SDSVHLS__
#include "hls_math.h"
#else
// The hls_math.h file includes hdl_fpo.h which contains actual code and
// will cause linker error in the ARM compiler, hence we add the function
// prototypes here
static float sqrtf(float x);
#endif

void my_sqrt(float x, float *ret);

#endif // _SQRT_H_
```

The file `my_sqrt.cpp` contains:

```
#include "my_sqrt.h"

void my_sqrt(float x, float *ret)
{
    *ret = sqrtf(x);
}
```

The makefile has the commands to compile these files:

```
sds++ -c -hw my_sqrt –sds-pf zc702 my_sqrt.cpp
sds++ -c my_sqrt_test.cpp
sds++ my_sqrt.o my_sqrt_test.o -o my_sqrt_test.elf
```

# Debugging an Application

The SDSoC™ environment allows projects to be created and debugged using the SDSoC IDE. Projects can also be created outside the SDSoC IDE (user-defined makefiles) and debugged either on the command line or using the SDSoC IDE.

See *SDSoC Environment Tutorial: Introduction* (UG1028) for information on using the interactive debuggers in the SDSoC IDE.

## Debugging Linux Applications in the SDSoC IDE

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

1. Select the **Debug** as the active build configuration and build the project.
2. Copy the generated `Debug/sd_card` image to an SD card, and boot the board with it.
3. Make sure the board is connected to the network, and note its IP address, for example, by executing `ifconfig eth0` at the command prompt.
4. Select the **Debug As** option to create a new debug-configuration, and enter the IP address for the board
5. 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 Applications in the SDSoC IDE

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

1. Select **Debug** as the active build configuration and build the project.
2. Make sure the board is connected to your host computer using the JTAG Debug connector.
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 on the **Debug** icon, which provides a shortcut to the procedure described above.

# Debugging FreeRTOS Applications

If you create a FreeRTOS application project using the SDSoC™ environment, you can debug your application using the same steps as a standalone (bare-metal) application project.

# Peeking and Poking IP Registers

Two small executables called `mrd` and `mwr` are available to peek and poke registers in memory-mapped programmable logic. These executables are invoked with the physical address to be accessed.

For example: `mrd 0x80000000 10` reads ten 4-byte values starting at physical address 0x80000000 and prints them to standard output, while `mwr 0x80000000 20`writes the value 20 to the address 0x8000000.

These executables can be used to monitor and change the state of memory-mapped registers in hardware functions and in other IP generated by the SDSoC™ environment.

**CAUTION!** *Trying to access non-existent addresses can cause the system to hang.*

# Debugging Performance Tips

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

Estimate the actual hardware acceleration time by looking at the latency numbers in the Vivado® Design Suite HLS report files (`_sds/vhls/…/*.rpt`). Latency of X accelerator clock cycles = 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.

# Hardware/Software Event Tracing

The systems produced by the SDSoC environment are high-performance, complex, hardware/software systems. 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 there is a lot happening all at once. The SDSoC environment tracing feature provides the user, through the use of event tracing, a detailed view of what is happening in the system during execution of an application.

This detailed view helps the user understand the performance of their application given the workload, hardware/software partitioning, and system design choices. Such information helps the user optimize and improve system implementation. This view enables event tracing of software running on the processor, as well as hardware accelerators and data transfer links in the system. Trace events are produced and gathered into a timeline view, showing the user a detailed perspective unavailable anywhere else about how their application executes.

Tracing an application produces a log that records information about system execution. Compared to event logging, event tracing provides correlation between events for a duration of time (i.e., events have a duration, 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.

Tracing requires a design to have at least one function marked for hardware. There is no way for the user to customize what is traced and what is not. All possible trace points are included automatically, including standard HLS-produced hardware accelerators, AXI4-Stream interfaces that serve data to or from an accelerator core, and the accelerator control code in software (stub code). Future releases will support tracing most hardware entities in a design and other designated events in software.

As with application debugging, for event tracing, you must connect a board to the host PC via JTAG for standalone and Ethernet for Linux. The application must be executed by the SDSoC GUI from the host using a debug or run configuration. It cannot be run manually by the user.

# 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 *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 20: 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 the `sdscc/++` linker 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
```

To give you an idea of the approximate resource utilization of these hardware monitor cores, the following table shows the resource utilization of these cores for a Zynq-7000 (xc7z020-1clg400) device:

| Core Name | LUTs | FFs | BRAMs | DSPs |
|---|---|---|---|---|
| Accelerator | 79 | 18 | 0 | 0 |
| AXI4-Stream (basic) | 79 | 14 | 0 | 0 |
| AXI4-Stream (statistics) | 132 | 183 | 0 | 0 |

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. Future releases will enable the user to choose which mode to use. The core does support it today so adventurous users could potentially configure the core manually in the Vivado tools. However, this is not supported in the current release.

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. The following table shows the resource utilization of this core for a Zynq-7000 (xc7z020-1clg400) device:

| Number of Ports | LUTs | FFs | BRAMs | DSPs |
|---|---|---|---|---|
| 1 | 241 | 404 | 0 | 0 |
| 2 | 307 | 459 | 0 | 0 |
| 3 | 366 | 526 | 0 | 0 |
| 4 | 407 | 633 | 0 | 0 |
| 6 | 516 | 686 | 0 | 0 |

| Number of Ports | LUTs | FFs | BRAMs | DSPs |
|---|---|---|---|---|
| 8 | 644 | 912 | 0 | 0 |
| 16 | 1243 | 1409 | 0 | 0 |
| 32 | 2190 | 2338 | 0 | 0 |
| 63 | 3830 | 3812 | 0 | 0 |

Depending on the number of ports (i.e., monitor cores), the integration core will use on average 110 flip-flops (FFs) and 160 look-up tables (LUTs). At the system level for example, the resource utilization for the matrix multiplication template application on the ZC702 platform (using the same xc7z020-1clg400 part) is shown in the table below:

| System | LUTs | FFs | BRAMs | DSPs |
|---|---|---|---|---|
| Base (no trace) | 16,433 | 21,426 | 46 | 160 |
| Event trace enabled | 17,612 | 22,829 | 48 | 160 |

Based on the results above, the difference in designs is approximately 1,000 LUTs, 1,200 FFs, and two BRAMs. This design has a single accelerator with three AXI4-Stream ports (two inputs and one output). When event trace is enabled, four monitors are inserted into the system (one accelerator and three AXI4-Stream monitors), in addition to a single integration core and other associated read-out logic. Given the resource estimations above, 720 LUTs and 700 FFs are from the actual trace monitoring hardware (monitors and integration core). The remaining 280 LUTs, 500 FFs and two BRAMs are from the read-out logic which converts the AXI4-Stream output trace data stream to JTAG. The resource utilization for this read-out logic is static and does not vary based on the design.

## 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.

***Figure 21: Matrix Multiplication Example Vivado IP Integrator Design Without Tracing Hardware***

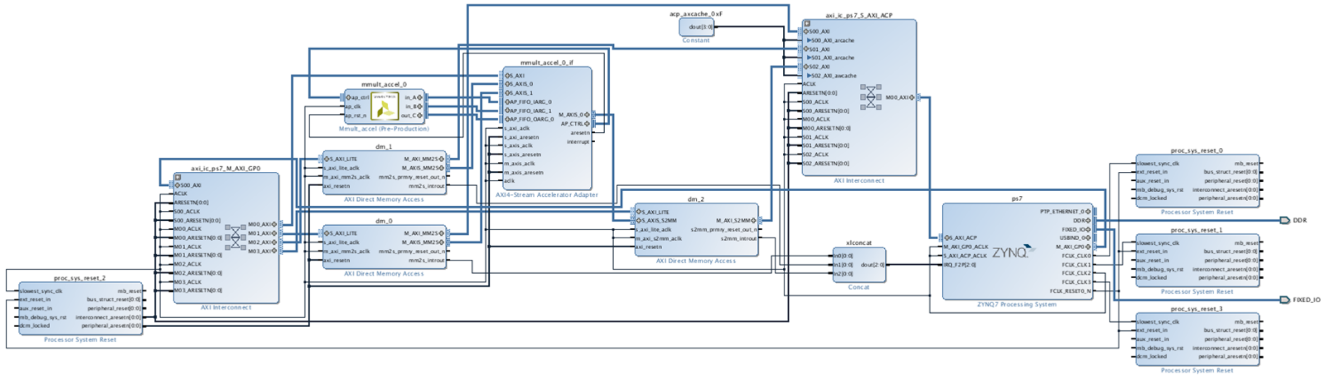
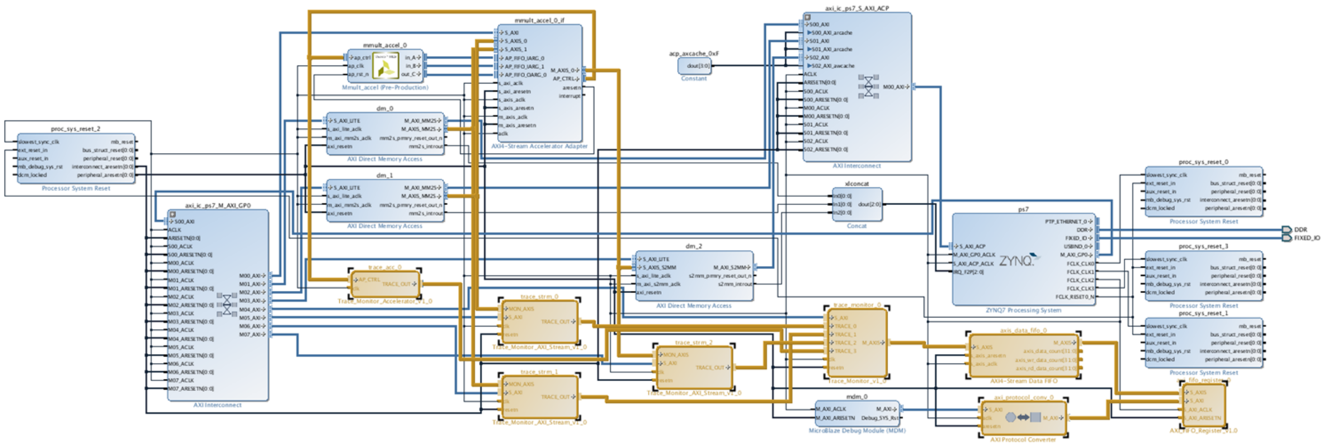



***Figure 22: Matrix Multiplication Example Vivado IP Integrator Design With Tracing Hardware (Shown in Orange)***




# Runtime Trace Collection

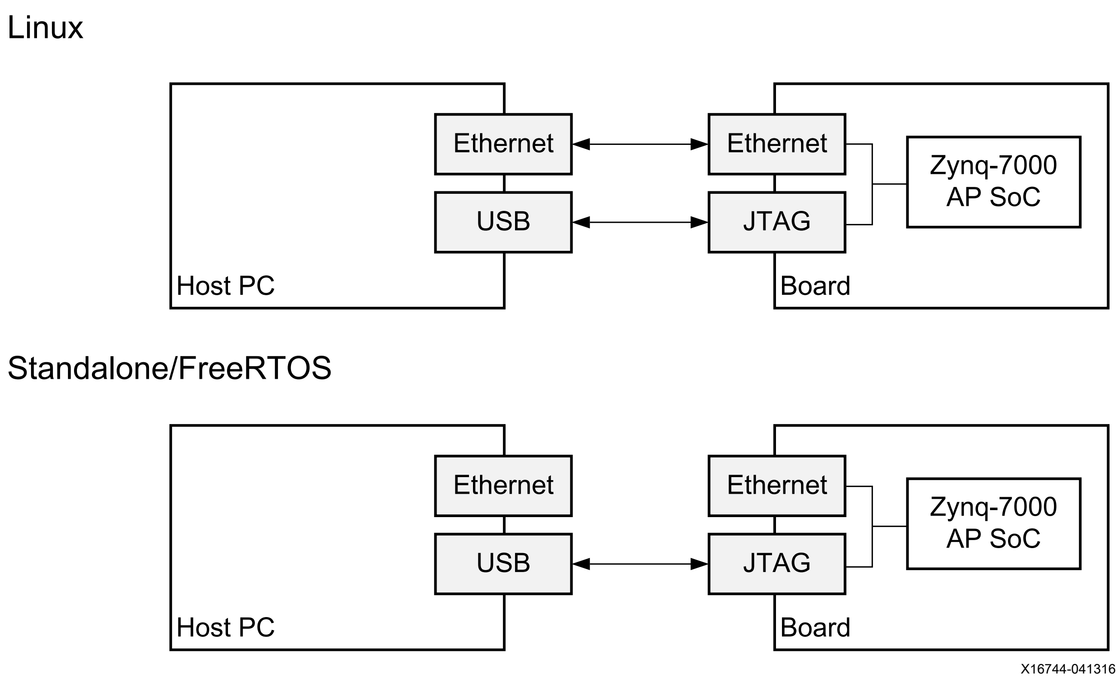
Software traces are inserted into the same storage path as the hardware traces and receive a timestamp using the same timer/counter as hardware traces. This single trace data stream is buffered in the hardware system and accessed over JTAG by the host PC.

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 is only displayed when the application is finished. In a future release, the real-time data will be displayed as it is captured.

The board connection requirements are slightly different depending on the operating system (standalone, FreeRTOS, or Linux). For standalone and FreeRTOS, the user program ELF is downloaded 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. The ELF is then copied and run 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.

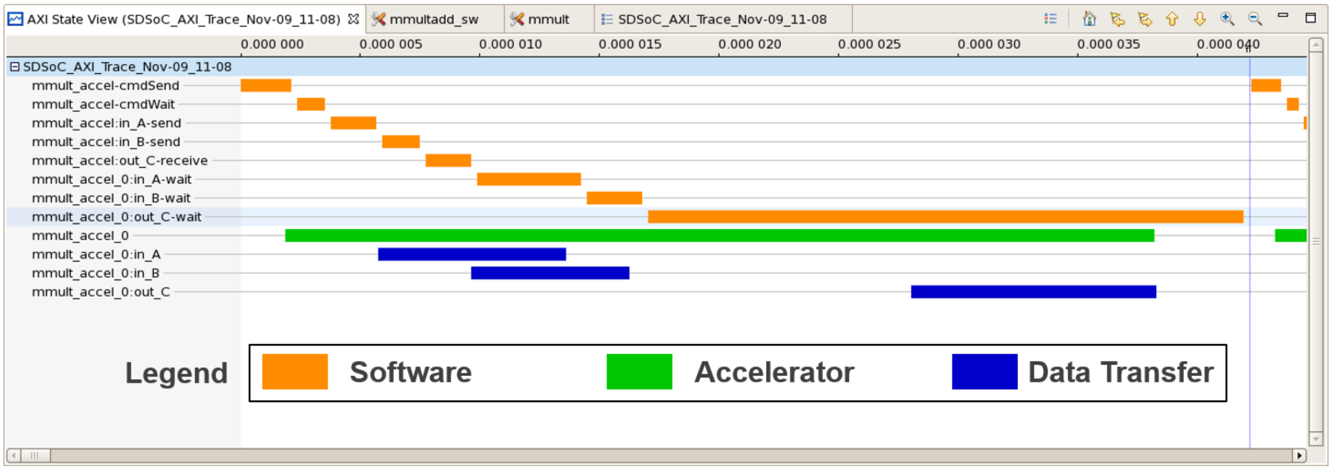
***Figure 23: Connections Required When Using Trace with Different Operating Systems***




# 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.

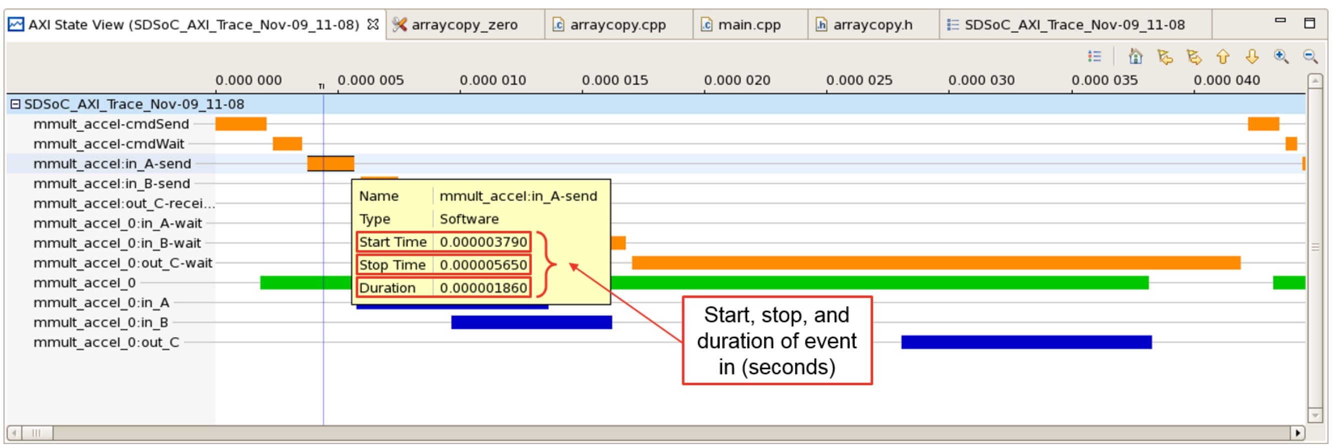
*Figure 24:* ***Example Trace Visualization Highlighting the Different Types of Events***




X16913-050216

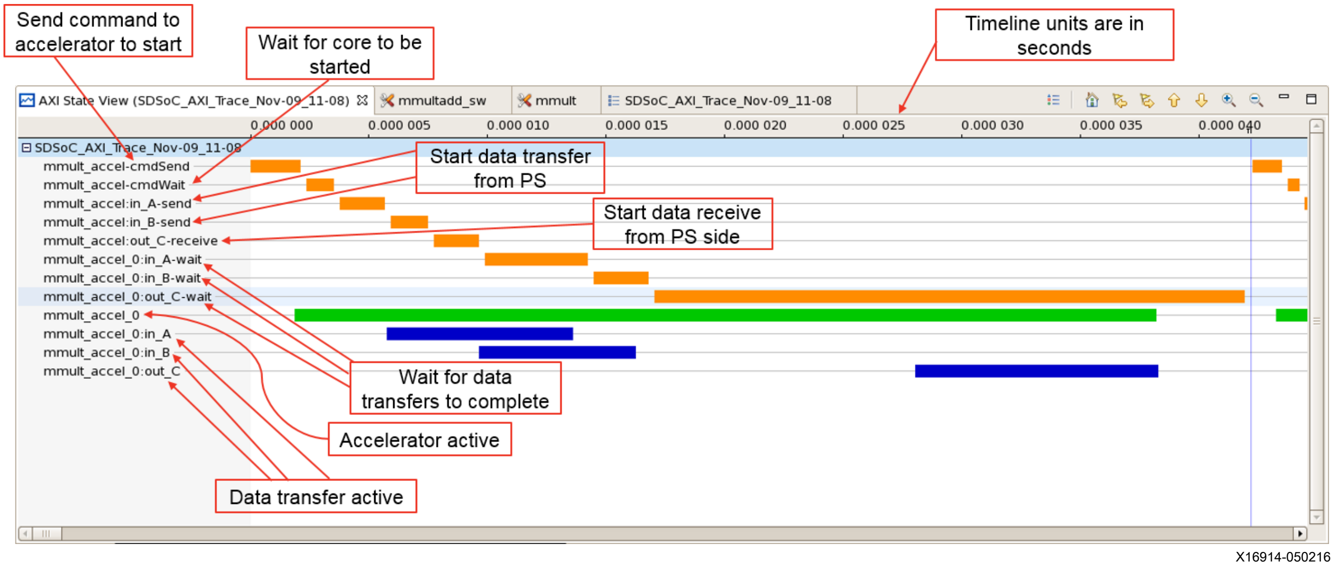
Each trace event has a few different attributes: name, type, start time, stop time, and duration. This data is shown as a tool-tip when the curser hovers above one of the event rectangles in the view.

*Figure 25:* ***Example Trace Visualization Highlighting the Detailed Information Available for Each Event***




X16912-050216

***Figure 26: Example Trace Visualization Highlighting the Event Names and Correlation to the User Program***




# Performance Measurement Using the AXI Performance Monitor

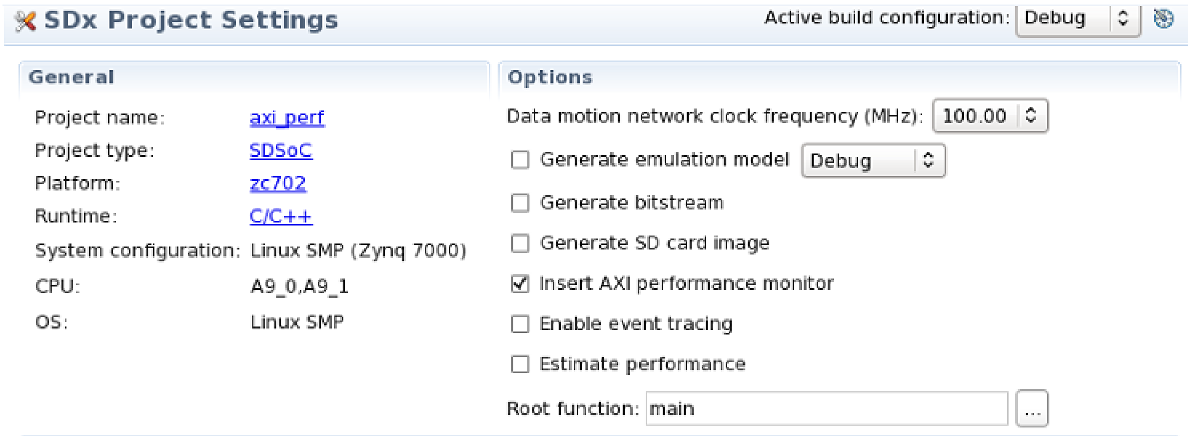
The AXI Performance Monitor (APM) module is used to monitor basic information about data transfers between the processing system (PS) ARM cores and the hardware in the programmable logic (PL). It captures statistics such as number of read/write transactions, throughput, and latency for the AXI transactions on the busses in the system.

In this section we will show how to insert an APM core into the system, monitor the instrumented system, and view the performance data produced.

## Creating a Standalone Project and Implementing APM

Open the SDSoC environment and create a new SDSoC Project using any platform or operating system selection. Choose the **Matrix Multiplication and Addition Template**.

In the **SDx Project Settings**, check the option **Insert AXI Performance Monitor**. Enabling this option and building the project adds the APM IP core to your hardware system. The APM IP uses a small amount of resources in the programmable logic. SDSoC connects the APM to the hardware/software interface ports, which are the Accelerator Coherency Port (ACP), General Purpose Ports (GP) and High Performance Ports (HP).

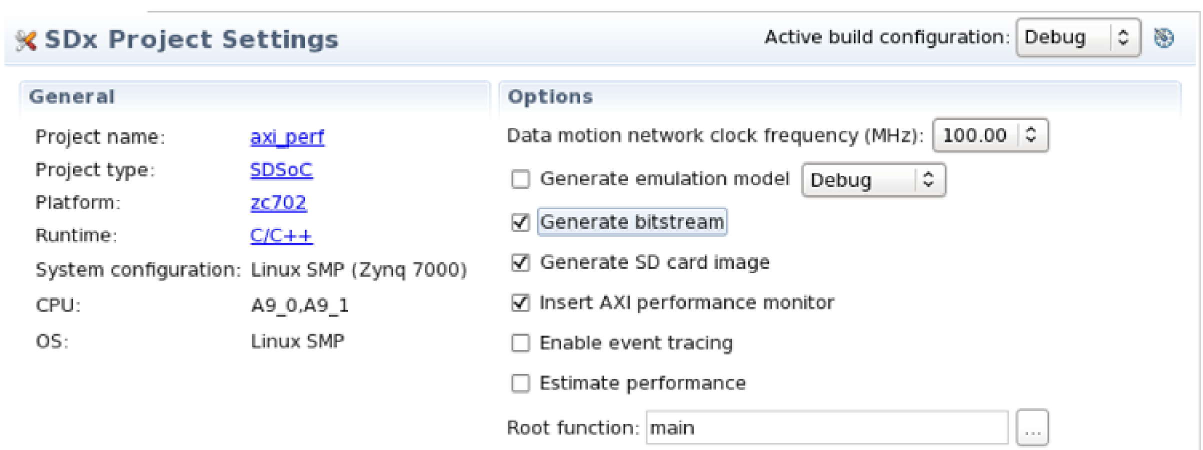



Select the `mmult` and `madd` functions to be implemented in hardware. Clean and build the project using the **Debug** configuration, which is selected by default.

## Creating a Linux Project and Implementing APM

Open the SDSoC environment and create a new SDSoC Project using any platform or operating system selection. Choose the **Matrix Multiplication and Addition Template**.

In the **SDx Project Settings**, check the option **Insert AXI Performance Monitor**. Enabling this option and building the project adds the APM IP core to your hardware system. The APM IP uses a small amount of resources in the programmable logic. SDSoC connects the APM to the hardware/software interface ports, which are the Accelerator Coherency Port (ACP), General Purpose Ports (GP) and High Performance Ports (HP).




Select the `mmult` and `madd` functions to be implemented in hardware. Clean and build the project using the **Debug** configuration, which is selected by default.

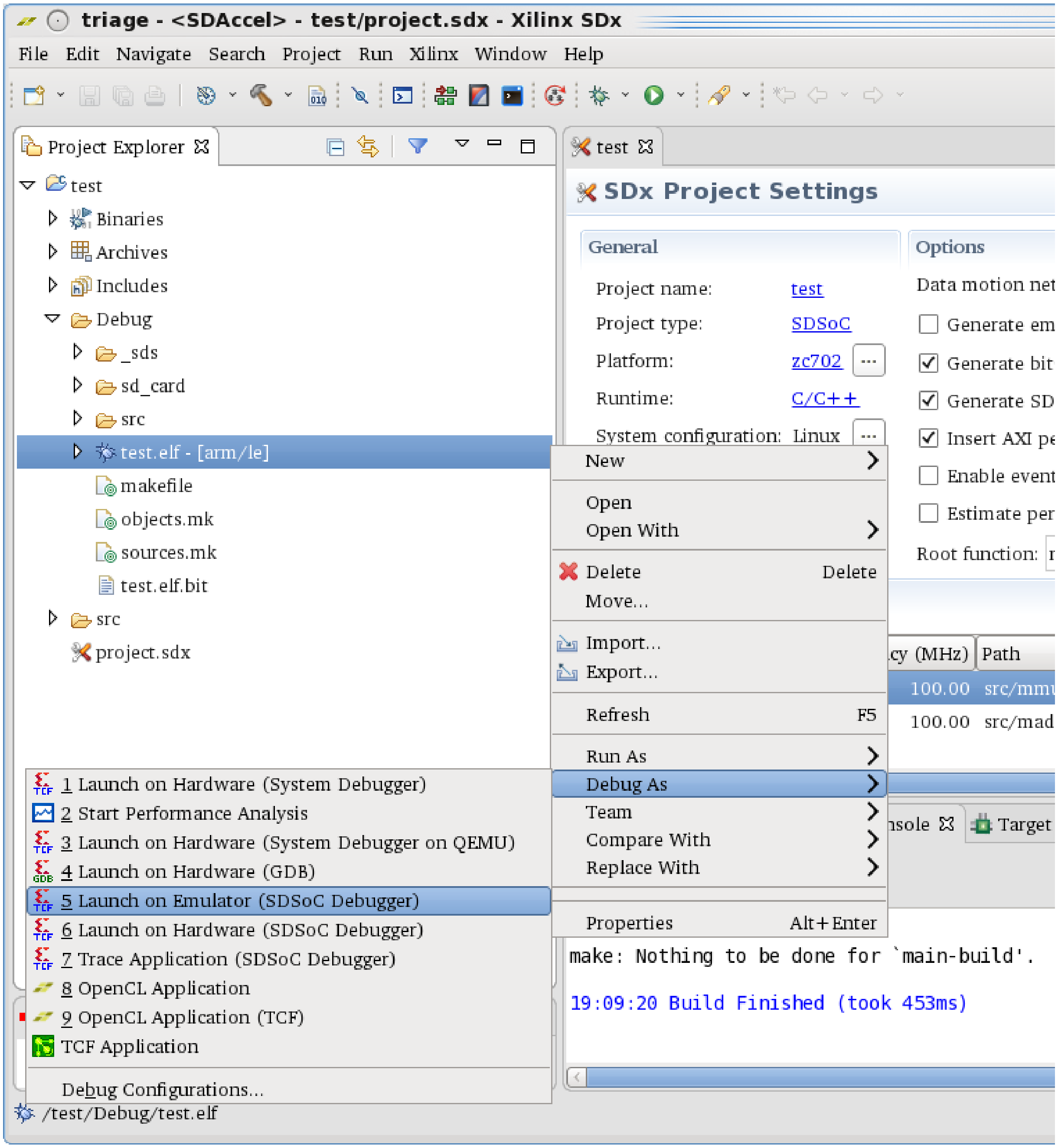
## Monitoring the Standalone Instrumented System

After the build completes, connect the board to your computer and power up the board. Click the **Debug** button to launch the application on the target. Switch to the **Debug** perspective. After programming the PL and launching the ELF, the program halts in main. Click on **Window→Perspective**.

Select **Performance Analysis** in the **Open Perspective** dialog and click **OK**.

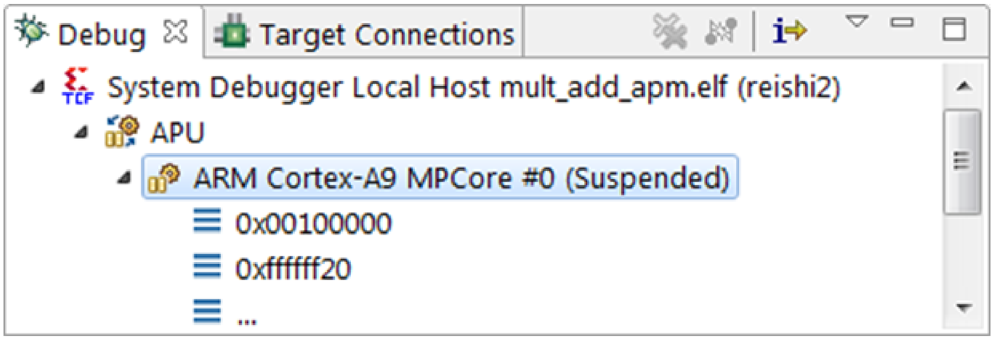
Switch back to the **SDx** perspective.

Expand the **Debug** folder in the **Project Explorer** view. Right click the ELF executable and select **Debug As→Launch on Hardware (SDSoC Debugger)**. If you are prompted to relaunch the application, click **OK**.

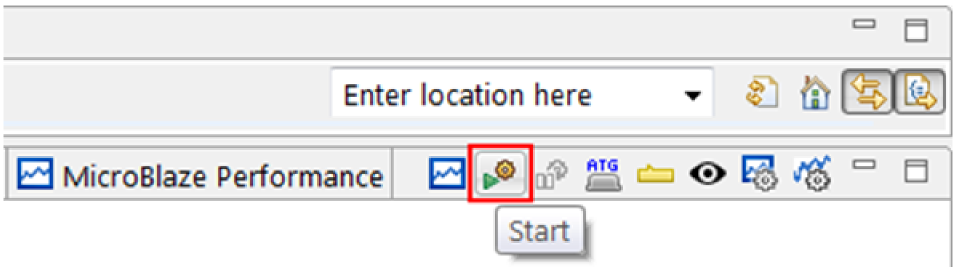



Click **Yes** to switch to the **Debug** perspective. After the application launches and halts at a breakpoint in the main function, switch back to the **Performance Analysis** perspective.

In the **Debug** view in the top left of the perspective, click on **ARM Cortex-A9 MPCore #0**.




Next, click on the **Start Analysis** button, which opens the **Performance Analysis Input** dialog.

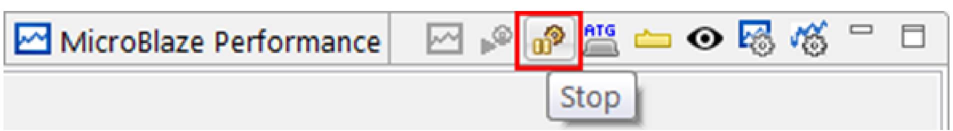



Check the box to **Enable APM Counters**. Click the **Edit** button to set up **APM Hardware Information**.

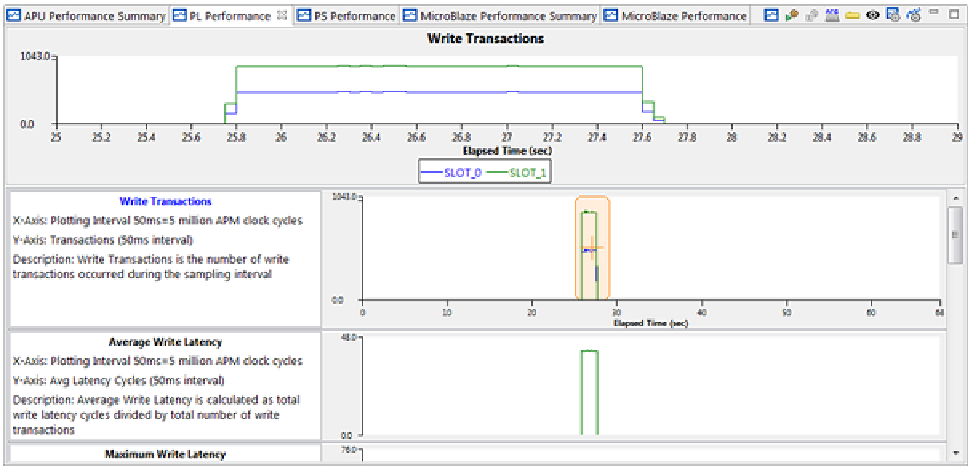
Click the **Load** button in the **APM Hardware Information** dialog. Navigate to `workspace_path/project/Debug/_sds/p0/vpl` and select the `zc702.hdf` file (zc702 is the platform name used in this example - use your platform instead). Click **Open**, then click **OK** in the **APM Hardware Information** dialog. Finally, click **OK** in the **Performance Analysis Input** dialog.

The **Analysis** views open in the **PL Performance** tab. Click the **Resume** button to run the application.

After your program completes execution, click the **Stop Analysis** button. If prompted by the **Confirm Perspective Switch** dialog to stay in the **Performance Analysis** perspective, click **No**.




Scroll through the analysis plots in the lower portion of the perspective to view different performance statistics. Click in any plot area to show a bigger version in the middle of the perspective. The orange box below allows you to focus on a particular time slice of data.




## Monitoring the Linux Instrumented System

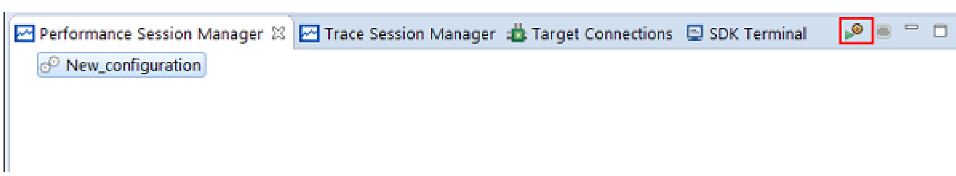
After the build completes, copy the contents of the `sd_card` directory onto an SD card, and boot Linux on the board. Connect the board to your computer (both UART and JTAG cables). Set up the Linux TCF agent target connection with the IP address of the board. Click the **Debug** button to launch the application on the target. Switch to the **Debug** perspective. After launching the ELF, the program halts in main.

Create a new Run Configuration by selecting **Run→Run Configuration** and double-clicking on **Xilinx C/C++ application (System Debugger)**. Ensure that the **Debug Type** is set to **Attach to running target**, then click **Run** to close the Run Configurations window. Click **Yes** in the Conflict dialog box that says "Existing launch configuration 'System Debugger on Local <your project>.elf' conflicts with the newly launched configuration...".

Switch to the Performance Analysis perspective by clicking on **Window→Open Perspective→Other ...**

Select **Performance Analysis** in the **Open Perspective** dialog and click **OK**.

Next, click on the **Start Analysis** button, which opens the **Performance Analysis Input** dialog.

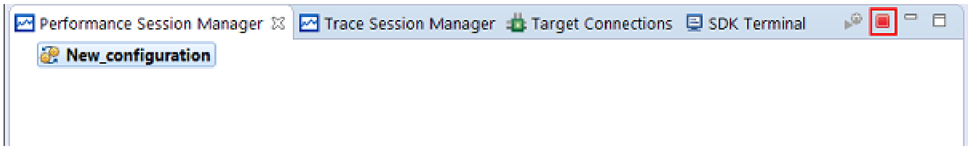



Check the box to **Enable APM Counters**. Click the **Edit** button to set up **APM Hardware Information**.

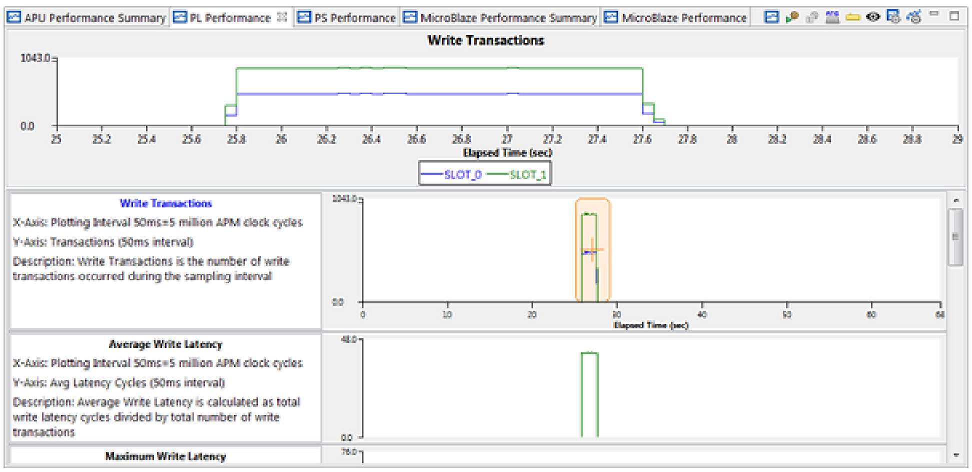
Click the **Load** button in the **APM Hardware Information** dialog. Navigate to `workspace_path/project/Debug/_sds/p0/vpl` and select the `zc702.hdf` file (zc702 is the platform name used in this example - use your platform instead). Click **Open**, then click **OK** in the **APM Hardware Information** dialog. Finally, click **OK** in the **Performance Analysis Input** dialog.

The **Analysis** views open in the **PL Performance** tab. Click the **Resume** button to run the application.

After your program completes execution, click the **Stop Analysis** button. If prompted by the **Confirm Perspective Switch** dialog to stay in the **Performance Analysis** perspective, click **No**.




Scroll through the analysis plots in the lower portion of the perspective to view different performance statistics. Click in any plot area to show a bigger version in the middle of the perspective. The orange box below allows you to focus on a particular time slice of data.




## Analyzing the Performance

In this system, the APM is connected to the two ports in use between the PS and the PL: the Accelerator Coherency Port (ACP) and the general purpose AXI port (GP). The multiplier and adder accelerator cores are both connected to the ACP for data input and output. The GP port is used to issue control commands and get the status of the accelerator cores only, not for data transfer. The blue Slot 0 is connected to the GP port, and the green Slot 1 is connect to the ACP.

The APM is configured in Profile mode with two monitoring slots, one for each: ACP and GP ports. Profile mode provides event counting functionality for each slot. The type of statistics computed by the APM for both reading and writing include:

- Transaction Count - Total number of requests that occur on the bus
- Byte Counter - Total number of bytes sent (used for write throughput calculation)
- Latency - Time from the start of the address issuance to the last element sent

The latency and byte counter statistics are used by the APM to automatically compute the throughput (in mega-bytes per second: MB/sec). The latency and throughput values shown are for a 50 millisecond (ms) time interval. Also, minimum, maximum, and averages are also displayed for latency and throughput statistics.

# 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 will 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 will 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.

# SDSoC Pragma Specification

This section describes pragmas (directives) for the SDSoC system compilers `sdscc/sds++` to assist system optimization.

All pragmas specific to the SDSoC environment are prefixed with `#pragma SDS` and should be inserted into C/C++ source code, either immediately prior to a function declaration or a function call site.

There is no single dominant industry standard in wide use for compilers that target heterogeneous embedded systems that employ hardware accelerators, but the pragmas and pragma syntax has been defined to be consistent with standards like OpenACC. In a future release, the SDSoC environment might adopt an industry standard pragmas should a suitable standard become widely adopted. For more information about pragmas, refer to *SDSoC Environment Tutorial: Introduction* (UG1253).

## Data Transfer Size

The syntax for this pragma is:

```
#pragma SDS data copy|zero_copy(ArrayName[offset:length])
```

This pragma must be specified immediately preceding a function declaration, or immediately preceding other `#pragma SDS` bound to the function declaration. This pragma applies to all the callers of the bound function.

Some notes about the syntax:

- The copy implies that data is explicitly copied from the processor memory to the hardware function. A suitable data mover as described in Improving System Performance performs the data transfer. The zero_copy means that the hardware function accesses the data directly from shared memory through an AXI4 bus interface. If no copy or zero_copy pragma is specified to an array argument, the SDSoC compiler assumes the copy semantics.
- The [offset:length] part is optional. When this part is not specified, this pragma is only used to select between copying the memory to/from the accelerator versus directly accessing the memory by the accelerator. For the array size, the SDSoC compiler first analyzes the callers to the accelerator function to determine the transfer size based on the memory allocation APIs for the array (for example, malloc or sds_alloc etc.). If the analysis fails, it checks the argument type to see if the argument type has a compile-time array size and use that size as the data transfer size. If no data transfer size can be determined, the compiler generates an error message so that the user can specify this pragma. If the data size is different between the caller and callee, or different between multiple callers, the compiler also generates an error message so that the user can correct the source code or use this pragma to override the compiler analysis.
- For a multi-dimensional array, each dimension should be specified. For example, for a 2-dimensional array, use ArrayName[offset_dim1:length_dim1][offset_dim2:length2_dim2]
- Multiple arrays can be specified in the same pragma, separated by a comma(,). For example, use copy(ArrayName1[offset1:length1], ArrayName2[offset2:length2])
- ArrayName must be one of the formal parameters of the function definition, that is, not from the prototype (where parameter names are optional) but from the function definition.
- offset is the number of elements from the first element in the corresponding dimension. It must be a compile-time constant. This is currently ignored.
- length is the number of elements transferred for that dimension. It can be an arbitrary expression as long as the expression can be resolved at runtime inside the function.

## Example 1

The following code snippet shows an example of applying the "copy" pragma to the "A" and "B" arguments of an accelerator function "foo" right before the function declaration:

```
#pragma SDS data copy(A[0:size*size], B[0:size*size])
void foo(int *A, int *B, int size)
```

The SDSoC system compiler will replace the body of the function "foo" with accelertor control, data transfer, and data synchronization code. The following code snippet shows the data transfer part:

```
void _p0_foo_0(int *A, int *B, int size)
{
    ...
    cf_send_i(&(_p0_swinst_foo_0.A), A, (size*size) * 4, &_p0_request_0);
    cf_receive_i(&(_p0_swinst_foo_0.B), B, (size*size) * 4, &_p0_request_1);
    ...
}
```

As shown above, the pragma value "size*size" is used to tell the SDSoC runtime the number of elements of array "A" and "B". The cf_send_i and cf_receive_i require the number of bytes, so the compiler will multiply the "size*size" with the number of bytes for each element (4 in this case). As shown in the example above, length need not be a compile-time constant; it can be a C arithmetic expression involving other scalar arguments of the same function.

## Example 2

The following code snippet shows an example of applying the "zero_copy" pragma instead of the "copy" pragma above:

```
#pragma SDS data zero_copy(A[0:size*size], B[0:size*size])
void foo(int *A, int *B, int size)
```

The data transfer part of the replaced function body becomes:

```
    cf_send_ref_i(&(_p0_swinst_foo_0.A), A, (size*size) * 4,
&_p0_request_0);
    cf_receive_ref_i(&(_p0_swinst_foo_0.B), B, (size*size) * 4,
&_p0_request_1);
```

The cf_send_ref_i and cf_receive_ref_i mean only transfer the reference or pointer of the array to the accelerator, and the accelerator will access the memory directly.

## Example 3

The following code snippet illustrates a common mistake—using an argument name in the function declaration that is different from the function definition:

```
"foo.h"
#pragma SDS data copy(in_A[0:1024])
void foo(int *in_A, int *out_B)

"foo.cpp"
#include "foo.h"
void foo(int *A, int *B)
{
...
}
```

This code will go through gcc without any problems. Actually, any C/C++ compiler will ignore the argument name in the function declaration, because the C/C++ standard makes the argument name in the function declaration optional. Only the argument name in the function definition is used by the compiler. In case of SDSoC, it will issue a warning later:

```
WARNING: [DMAnalysis 83-4484] Cannot find argument in_A in accelerator
function foo(int *A, int *B)
```

# Memory Attributes

For an operating system like Linux that supports virtual memory, user-space allocated memory is paged, which can affect system performance. SDSoC runtime also provides API to allocate physically contiguous memory. The pragmas in this section can be used to tell the compiler whether the arguments have been allocated in physically contiguous memory.

## Physically Contiguous Memory

**IMPORTANT:** *The syntax and implementation of this pragma might be revised in a future release.*

The syntax for this pragma is:

```
#pragma SDS data mem_attribute(ArrayName:contiguity)
```

This pragma must be specified immediately preceding a function declaration, or immediately preceding another `#pragma SDS` bound to the function declaration. This pragma applies to all the callers of the function.

Some notes about the syntax:

- `ArrayName` must be one of the formal arguments of the function definition.
- `Contiguity` must be either PHYSICAL_CONTIGUOUS or NON_PHYSICAL_CONTIGUOUS. The default value is set to be NON_PHYSICAL_CONTIGUOUS.

  PHYSICAL_CONTIGUOUS means that all memory corresponding to the associated `ArrayName` is allocated using `sds_alloc`, while NON_PHYSICAL_CONTIGUOUS means that all memory corresponding to the associated `ArrayName` is allocated using `malloc` or as a free variable on the stack. This helps the SDSoC compiler select the optimal data mover.
- Multiple arrays can be specified in one pragma, separated by commas.

## Example 1

The following code snippet shows an example of specifying the `contiguity` attribute:

```
#pragma SDS data mem_attribute(A:PHYSICAL_CONTIGUOUS)
void foo(int A[1024], int B[1024])
```

In the above example, the user tells the SDSoC compiler that array `A` is allocated in the memory block that is physically contiguous. The SDSoC compiler then chooses AXI_DMA_Simple instead of AXI_DMA_SG, because the former is smaller and faster at transferring physically contiguous memory.

# Data Access Pattern

The syntax for this pragma is:

```
#pragma SDS data access_pattern(ArrayName:pattern)
```

This pragma must be specified immediately preceding a function declaration, or immediately preceding another `#pragma SDS` bound to the function declaration.

Some notes about the syntax:

- `pattern` can be either `SEQUENTIAL` or `RANDOM`, by default it is `RANDOM`

This pragma specifies the data access pattern in the hardware function. If a `copy` pragma has been specified for an array argument, SDSoC checks the value of this pragma to determine the hardware interface to synthesize. If the access pattern is `SEQUENTIAL`, a streaming interface (such as ap_fifo) will be generated. Otherwise, with `RANDOM` access pattern, a RAM interface will be generated. Refer to Data Motion Network Generation in SDSoC for the usage of this pragma in data motion network generation in SDSoC.

**Example 1**:

The following code snippet shows an example of using this pragma for an array argument:

```
#pragma SDS data access_pattern(A:SEQUENTIAL)
void foo(int A[1024], int B[1024])
```

In the example shown above, a streaming interface will be generated for argument `A`, while a RAM interface will be generated for argument `B`. The access pattern for argument `A` must be A[0], A[1], A[2], ... , A[1023], and all elements must be accessed only once. On the other hand, argument `B` can be accessed in a random fasion, and each element can be accessed zero or more times.

**Example 2**:

The following code snippet shows an example of using this pragma for a pointer argument:

```
#pragma SDS data access_pattern(A:SEQUENTIAL)
void foo(int *A, int B[1024])
```

In the above example, if argument `A` is intended to be a streaming port, the two pragmas shown must be applied. Without these, SDSoC synthesizes argument `A` as a register (IN, OUT, or INOUT based on the usage of `A` in function `foo`).

**Example 3**:

The following code snippet shows the effect of `zero_copy` pragma (refer to Data Transfer Size) on the `access_pattern` pragma:

```
#pragma SDS data zero_copy(A)
#pragma SDS data access_pattern(A:SEQUENTIAL)
void foo(int A[1024], int B[1024])
```

In the above example, the `access_pattern` pragama is ignored. Once a `zero_copy` pragma has been applied to an argument, the AXI4 interface will be synthesized for that argument. Please refer to Zero Copy Data Mover for more details.

# Data Mover Type

**IMPORTANT:** *This pragma is not recommended for normal use. Only use this pragma if the compiler-generated data mover type does not meet the design requirement.*

The syntax for this pragma is:

```
#pragma SDS data data_mover(ArrayName:DataMover[:id])
```

This pragma must be specified immediately preceding a function declaration, or immediately preceding another `#pragma SDS` bound to the function declaration. This pragma applies to all the callers of the bound function.

Some notes about the syntax:

- Multiple arrays can be specified in one pragma, separated by a comma (,). For example:

  ```
  #pragma SDS data data_mover(ArrayName:DataMover[:id],
  ArrayName:DataMover[:id])
  ```

- `ArrayName` must be one of the formal parameters of the function.
- `DataMover` must be either AXIFIFO, AXIDMA_SG, or AXIDMA_SIMPLE.
- `:id` is optional, and `id` must be a positive integer.

This pragma specifies the data mover HW IP type used to transfer an array argument. By default, the compiler chooses the type of the data automatically by analyzing the code. This pragma can be used to override the compiler inference rules. Without the optional `:id`, the compiler automatically assigns a data mover HW IP instance for transferring the corresponding array. The `:id` can be used to override the compiler's choice and assign a specific data mover HW IP instance for the associated formal parameter. If more than two formal parameters have the same HW IP type and same `id`, they will share the same data mover HW IP instance.

There are some additional requirements for using AXIDMA_SIMPLE.

- The corresponding array must be allocated uisng `sds_alloc()`.

## Example 1

The following code snippet shows an example of specifying the data mover ID in the pragma:

```
#pragma SDS data data_mover(A:AXIDMA_SG:1, B:AXIDMA_SG:1)
void foo(int A[1024], int B[1024])
```

In the above example, the same AXIDMA_SG IP instance is shared to transfer data for arguments `A` and `B`, because the same data mover `ID` has been specified.

# SDSoC Platform Interfaces to External Memory

**IMPORTANT:** *The syntax and implementation of this pragma might be revised in a future release.*

The syntax for this pragma is:

```
#pragma SDS data sys_port(ArrayName:port)
```

This pragma must be specified immediately preceding a function declaration, or immediately preceding another `#pragma SDS` bound to the function declaration, and applies to all the callers of the function.

Some notes about the syntax:

- `ArrayName` must be one of the formal arguments of the function definition.
- `port` must be ACP or AFI or MIG. The Zynq-7000 All Programmable SoC provides a cache coherent interface between programmable logic and external memory (S_AXI_ACP) and high-performance ports (S_AXI_HP) for non-cache coherent access (AFI). If no `sys_port` pragma is specified for an array argument, the interface to external memory is determined automatically by the SDSoC system compilers, based on array memory attributes (cacheable or non-cacheable), array size, data mover used, etc. This pragma overrides the SDSoC compiler choice of memory port. MIG is valid only for the `zc706_mem` platform.
- Multiple arrays can be specified in one pragma, separated by commas.

## Example 1

The following code snippet shows an example of using this pragma:

```
#pragma SDS data sys_port(A:AFI)
void foo(int A[1024], int B[1024])
```

In the above example, if the caller passes an array allocated with `malloc` to `A`, the SDSoC compiler uses the AFI platform interface, even though this might not be the optimal choice.

# Hardware Buffer Depth

The syntax of this pragma is:

```
#pragma SDS data buffer_depth(ArrayName:BufferDepth)
```

**IMPORTANT:** *The hardware interpretation of this pragma might be revised in a future release.*

This pragma must be specified immediately preceding a function declaration, or immediately preceding another `#pragma SDS` bound to the function declaration, and applies to all the callers of the function.

Some notes about the syntax:

- Multiple arrays can be specified in one pragma, separated by a comma(,). For example:

```
#pragma SDS data buffer_depth(ArrayName1:BufferDepth1,
ArrayName2:BufferDepth2)
```

- `ArrayName` must be one of the formal parameters of the function.
- `BufferDepth` must a compile-time constant value.
- This pragma applies only to arrays that map to BRAM or FIFO interfaces. For a BRAM-mapped array, the value specifies hardware multi-buffer depth. For a FIFO-mapped array, the value specifies the depth of the hardware FIFO allocated for the array. For this pragma, the following must hold:
  - BRAM: $1 \leq$ BufferDepth $\leq 4$, and $2 \leq$ ArraySize $\leq 16384$.
  - FIFO: BufferDepth = $2^n$, where $4 \leq n \leq 20$.

# Asynchronous Function Execution

These two pragmas are paired to support manual control of the hardware function synchronization.

The syntax of these pragmas is:

```
#pragma SDS async(ID)
#pragma SDS wait(ID)
```

The `async` pragma is specified immediately preceding a call to a hardware function, directing the compiler not to automatically generate the wait based on data flow analysis.

The `wait` pragma must be inserted at an appropriate point in the program to direct the CPU to wait until the associated `async` function call (same ID) has completed.

- The `ID` must be a compile time unsigned integer constant.
- In the presence of an `async` pragma, the SDSoC system compiler does not generate an `sds_wait()` in the stub function for the associated call. The program must contain the matching `sds_wait(ID)` or `#pragma SDS wait(ID)` at an appropriate point to synchronize the controlling thread running on the CPU with the hardware function thread. An advantage of using the `#pragma SDS wait(ID)` over the `sds_wait(ID)` function call is that the source code can then be compiled by compilers other than `sdscc` (such as gcc that does not interpret either `async` or `wait` pragmas).

## Example 1

The following code snippet shows an example of using these pragmas with the same ID to pipeline the data transfer and accelerator execution:

```
for (int i = 0; i < pipeline_depth; i++) {
     #pragma SDS async(1)
     mmult_accel(A[i%NUM_MAT], B[i%NUM_MAT], C[i%NUM_MAT]);
}

for (int i = pipeline_depth; i < NUM_TESTS-pipeline_depth; i++) {
     #pragma SDS wait(1)
     #pragma SDS async(1)
     mmult_accel(A[i%NUM_MAT], B[i%NUM_MAT], C[i%NUM_MAT]);
}
for (int i = 0; i < pipeline_depth; i++) {
     #pragma SDS wait(1)
}
```

In the above example, the first loop ramps up the pipeline with a depth of pipeline_depth, the second loop executes the pipeline, and the third loop ramps down the pipeline. The hardware buffer depth (discussed in Hardware Buffer Depth) should be set to the same value as pipeline_depth. The goal of this pipeline is to transfer data to the accelerator for the next execution while the current execution is not finished. Refer to Increasing System Parallelism and Concurrency for more information.

## Example 2

The following code snippet shows an example of using these pragmas with different ID:

```
{
    #pragma SDS async(1)
    mmult(A, B, C);
    #pragma SDS async(2)
    mmult(D, E, F);
    ...
    #pragma SDS wait(1)
    #pragma SDS wait(2)
}
```

The program running on the hardware first transfers A and B to the mmult hardware and returns immediately. Then the program transfers D and E to the mmult hardware and returns immediately. When the program later executes to the point of #pragma SDS wait(1), it waits for the output C to be ready. When the program later excutes to the point of #pragma SDS wait(2), it waits for the output F to be ready.

# Specifying Resource Binding

This pragma can be used for function callsites to manually specify resource binding.

The syntax of the pragma is:

```
#pragma SDS resource(ID)
```

The `resource` pragma is specified immediately preceding a call to a hardware function, directing the compiler to bind the caller to a specified accelerator instance.

The `ID` must be a compile time unsigned integer constant. For the same function, each unique ID represents a unique instance of the hardware accelerator.

## Example 1

The following code snippet shows an example of using this pragma with a different `ID`:

```
{
    #pragma SDS resource(1)
    mmult(A, B, C);
    #pragma SDS resource(2)
    mmult(D, E, F);
    ...
}
```

In the above example, the first call to `mmult` will be bound to an accelerator with an ID of 1, and the second call to `mmult` will be bound to another accelerator with an ID of 2.

# Specifying Partitions

The SDSoC system compilers `sdscc/sds++` can automatically generate multiple bitstreams for a single application that is loaded dynamically at run-time. Each bitstream has a corresponding partition identifier. A platform might not support bitstream reloading, for example, due to platform peripherals that cannot be shut down and then brought back up after reloading.

The syntax of this pragma is:

```
#pragma SDS partition(ID)
```

The `partition` pragma is specified immediately preceding a call to a hardware function, directing the compiler to assign the implementation of the hardware function to the partition `ID`.

- In the absence of a `partition` pragma, a hardware function is implemented in partition 0.
- `ID` must be a positive integer. Partition `ID 0` is reserved.

## Example 1

The following example shows an example of using this pragma:

```
foo(a, b, c);
#pragma SDS partition (1)
bar(c, d);
#pragma SDS partition (2)
bar(d, e);
```

In this example, hardware function `foo` has no partition pragma, so it is implemented in partition 0. The first call to `bar` is implemented in partition 1, and the second `bar` is implemented in partition 2.

A complete example showing the usage of this pragma can be found in `<install_path>/samples/file_io_manr_sobel_partitions`.

# Trace Monitoring

The syntax for this pragma is:

```
#pragma SDS monitor trace(var1[:SW|HW][,var2[:SW|HW]])
```

This pragma must be specified immediately preceding a function declaration, or immediately preceding another `#pragma SDS` bound to the function declaration.

This pragma specifies the trace insertion for the accelerator with different granularity. The user can set the `var` to be the accelerator function name or individual argument name. The kind of trace can be either `SW` or `HW` or both. `HW` trace means the "start" and "stop" of the corresponding hardware component, such as the "start" and "stop" of the hardware accelerator, or the "start of data transfer" and "stop of data transfer" of the argument. `SW` trace means the `stub command` for the accelerator and arguments.

## Example 1

The following code snippet shows an example of using this pragma to trace the accelerator `foo`:

```
#pragma SDS monitor trace(foo)
void foo(int a, int b);
```

In the example shown above, both `HW` and `SW` traces are inserted for the accelerator `foo`.

## Example 2

The following code snippet shows an example of using this pragma to trace an argument.

```
#pragma SDS monitor trace(a, b:SW, c:HW)
void foo(int a, int b, int *c);
```

In the above example, both `HW` and `SW` traces are inserted for argument `a`. For argument `b`, only the `SW` trace is inserted. For argument `c`, only the `HW` trace is inserted.

Chapter 11

# SDSCC/SDS++ Compiler Commands and Options

This section describes the SDSoC `sdscc/sds++` compiler commands and options.

## Name

```
sdscc – SDSoC C compiler
```

```
sds++ - SDSoC C++ compiler
```

## Command Synopsis

```
sdscc | sds++ [hardware_function_options] [system_options]
[performance_estimation_options] [options_passed_through_to_cross_compiler]
[-mno-ir]
[-sds-pf platform_name] [-sds-pf-info platform_name] [-sds-pf-list]
[-sds-sys-config configuration name [-sds-proc processor_name]] [-target-os
os_name]
[-verbose] [ -version] [--help] [files]
```

### Hardware Function Options

```
[-sds-hw function_name file [-clkid clock_id_number] [-files file_list]
[-hls-tcl hls_tcl_directives_file] [-mno-lint] -shared-aximm –sds-end]*
```

### Performance Estimation Options

```
[[-perf-funcs function_name_list -perf-root function_name] |
[-perf-est data_file][-perf-est-hw-only]]
```

## System Options

```
[[-apm] [-disable-ip-cache] [-dm-sharing <0-3>] [-dmclkid clock_id_number]
[-emulation mode] [-impl-strategy <strategy>]
[-instrument-stub] [-maxthreads number] [-mno-bitstream][-mno-boot-files]
[-rebuild-hardware]
[-synth-strategy <strategy>] [-trace] [-trace-buffer depth] [-trace-no-sw]
[-maxjobs <number>] [-sdcard <data_directory>]]
```

The `sdscc`/`sds++` compilers compile and link C/C++ source files into an application-specific hardware/software system on chip implemented on a Zynq-7000 All Programmable SoC or Zynq UltraScale+ MPSoC.

The command usage and options are identical for `sdscc` and `sds++`.

Options not recognized by `sdscc` are passed to the ARM cross-compiler. Compiler options within an `-sds-hw ... -sds-end` clause are ignored for the `-c foo.c` option when `foo.c` is not the file containing the specified hardware function.

When linking the application ELF, `sdscc` creates and implements the hardware system, and generates an SD card image containing the ELF and boot files required to initialize the hardware system, configure the programmable logic and run the target operating system.

When linking application ELF files for non-Linux targets, for example Standalone or FreeRTOS, default linker scripts found in the folder `<install_path>/platforms/<platform_name>` are used. If a user-defined linker script is required, it can be specified using the `-Wl,-T -Wl,<path_to_linker_script>` linker option.

When building a system containing no functions marked for hardware implementation, `sdscc` uses pre-built hardware when available for the target platform. To force bitstream generation, use the `-rebuild-hardware` option.

Report files are found in the folder `_sds/reports`.

When running Linux applications that use shared libraries, the libraries must be contained in the root file system or SD card, and the path to the libraries added to the LD_LIBRARY_PATH environment variable.

**Optional PL Configuration After Linux Boot**

When `sdscc/sds++` creates a bitstream `.bin` file in the `sd_card` folder, it can be used to configure the PL after booting Linux and before running the application ELF. The embedded Linux command used is `cat bin_file > /dev/xdevcfg`.

# General Options

The following command line options are applicable to any `sdscc` invocation or display information for the user.

## -sds-pf platform_name

Specify the target platform that defines the base system hardware and software, including operation system and boot files. The `platform_name` can be the name of a platform in the SDSoC™ environment installation, or a file path to a folder containing platform files, with the last component of the path matching the platform name. The platform defines the base hardware and software, including operation system and boot files. Use this option when compiling accelerator source files and when linking the ELF file. Use the `-sds-pf-list` option to list available platforms.

## -sds-pf-info platform_name

Display general information about a platform. Use the `-sds-pf-list` option to list available platforms. The information displayed includes available system configurations that can be specified with the `-sds-sys-config` system_configuration option.

## -sds-pf-list

Display a list of available platforms and exit (no other options are specified). The information displayed includes available system configurations that can be specified with the `-sds-sys-config` system_configuration option.

## -sds-sys-config configuration_name

Specify the system configuration that defines the software platform used, which includes the target operating system and other settings. The `-sds-pf-list` and `-sds-pfinfo` options can be used to list the available system configurations for a platform. When the `-sds-sys-config` option is used, do not specify the `-target-os` option. If the `-sds-sys-config` option is not specified, the default system configuration is used.

## -sds-proc processor_name

Specify the processor name to use with the system configuration defined by the `-sds-sys-config` option. A system configuration normally specifies a target CPU, and this option is not required.

## -target-os os_name

Specify the target operating system. The selected OS determines the compiler toolchain used, and include file and library paths added by `sdscc`. `os_name` can be one of the following:

- `linux` : for the Linux OS. This is the default if the command line contains no `-target-os` option
- `standalone` : for standalone or bare-metal applications
- `freertos` : for FreeRTOS

If the `-sds-sys-config` system_configuration option is specified, do not specify the `-target-os` option, because a system configuration itself defines a target operating system. If you do not specify the `-sds-sys-config` but do specify the `-target-os` option, SDSoC searches for a system configuration with an OS that matches the one specified by `-target-os`.

## -verbose

Print verbose output to STDOUT.

## -version

Print the `sdscc` version information to STDOUT.

## --help

Print command line help information. Note that two consecutive hyphen or dash characters – are used.

The following command line options are applicable only to `sdscc` invocations used to compile a source file.

## -mno-ir

Suppress the generation of an intermediate representation (IR) for a source file that does not contain hardware accelerators or their callers. This option is not used unless needed to override an error condition during compilation of a specific source file (do not apply this option to every source file), for example IR generation does not handle source files containing Zynq NEON intrinsics. By default, an IR is created for each source file when it is compiled and used in the analysis of the application program.

# Hardware Function Options

Hardware function options provide a means to consolidate sdscc options within a Makefile to simplify command line calls and make minimal modifications to a pre-existing Makefile. The Makefile fragment below illustrates the use of -sds-hw blocks to collect all options in the SDSFLAGS Makefile variable and to replace an original definition of CC with sdscc ${SDSFLAGS} or sds++ ${SDSFLAGS}. Thus the original Makefile for an application can be converted to an sdscc/sds++ compiler Makefile with minimal changes.

```
APPSOURCES = add.cpp main.cpp
EXECUTABLE = add.elf

CROSS_COMPILE = arm-xilinx-linux-gnueabi-
AR = ${CROSS_COMPILE}ar
LD = ${CROSS_COMPILE}ld
#CC = ${CROSS_COMPILE}g++
PLATFORM = zc702
SDSFLAGS = -sds-pf ${PLATFORM} \
           -sds-hw add add.cpp -clkid 1 -sds-end \
           -dmclkid 2
CC = sds++ ${SDSFLAGS}

INCDIRS = -I..
LDDIRS =
LDLIBS =
CFLAGS = -Wall -g -c ${INCDIRS}
LDFLAGS = -g ${LDDIRS} ${LDLIBS}

SOURCES := $(patsubst %,../%,$(APPSOURCES))
OBJECTS := $(APPSOURCES:.cpp=.o)

.PHONY: all

all: ${EXECUTABLE}

${EXECUTABLE}: ${OBJECTS}
 ${CC} ${OBJECTS} -o $@ ${LDFLAGS}

%.o: ../%.cpp
 ${CC} ${CFLAGS} $<
```

## -sds-hw function_name file [[-files file_list] [-hls-tcl hls_tcl_directives_file] [-clkid <n>] [-mno-lint]] –sds-end

An sdscc command line may include zero or more -sds-hw blocks, and each block is associated with a top-level hardware function specified as the first argument and its containing source file specified as the second argument. If the file name associated with an -sds-hw block matches the source file to be compiled, the options are applied. Options outside of -sds-hw blocks are applied where applicable.

When using the AuvizCV library, the `function_name` is the template function instantiation enclosed in double quotes, for example "auCanny<1080,1920,0,0,3,2,1,1,1>", and the file is the source file containing the template function instantiation, for example `au_canny_tb.cpp`.

## -clkid <n>

Set the accelerator clock ID to `<n>`, where `<n>` has one of the values listed in the table below. (You can use the command `sdscc -sds-pf-info platform_name` to display the information about a platform.) If the `clkid` option is not specified, the default value for the platform is used. Use the command `sdscc -sds-pf-list` to list available platforms and settings.

| Platform | Value of <n> |
|---|---|
| zc702 | 0 – 166 MHz |
| | 1 – 142 MHz |
| | 2 – 100 MHz |
| | 3 – 200 MHz |
| zc706 | 0 – 166 MHz |
| | 1 – 142 MHz |
| | 2 – 100 MHz |
| | 3 – 200 MHz |
| zed and microzed | 0 – 166 MHz |
| | 1 – 142 MHz |
| | 2 – 100 MHz |
| | 3 – 200 MHz |
| zybo | 0 – 25 MHz |
| | 1 – 100 MHz |
| | 2 – 125 MHz |
| | 3 – 50 MHz |
| zcu102 | 0 – 100 MHz |
| | 1 – 150 MHz |
| | 2 – 200 MHz |
| | 3 – 300 MHz |

## -files file_list

Specify a comma-separated list (without white space) of one or more files required to compile the current top-level function into hardware using Vivado® HLS. If any of these files contain source code that is not used by HLS but is required to produce the application executable, they must be compiled separately to create object files (.o), and linked with other object files during the link phase.

When using the AuvizCV library, the `-files` option specifies the path to the source file containing the function template definition, for example `au_canny.hpp`.

### -hls-tcl hls_tcl_directives_file

When using the Vivado® HLS tool to synthesize the hardware accelerator, source the specified Tcl file containing HLS directives. During HLS synthesis, `sdscc` creates a `run.tcl` file used to drive the Vivado HLS tool and in this Tcl file, the following commands are inserted:

```
# synthesis directives
create_clock -period <clock_period>
set_clock_uncertainty 27.0%
config_rtl -reset_level low
source <sdsoc_generated_tcl_directives_file>
# end synthesis directives
```

If the `-hls-tcl` option is used, the user-defined Tcl file is sourced after the synthesis directives generated by the SDSoC environment.

### -mno-lint

Suppress the static analysis of hardware accelerator source files. This linting process checks for potential errors or issues in the source file. This option should only be used if the analysis prevents generation of the hardware accelerator and you are certain that you can continue.

### -shared-aximm

Share AXIMM ports instead of enabling multiple ports.

## Compiler Macros

Predefined macros allow you to guard code with `#ifdef` and `#ifndef` preprocessor statements; the macro names begin and end with two underscore characters '_'. The `__SDSCC__` macro is defined whenever `sdscc` or `sds++` is used to compile source files; it can be used to guard code depending on whether it is compiled by `sdscc/sds++` or another compiler, for example `GCC`.

When `sdscc` or `sds++` compiles source files targeted for hardware acceleration using Vivado HLS, the `__SDSVHLS__` macro is defined to be used to guard code depending on whether high-level synthesis is run or not.

The code fragment below illustrates the use of the `__SDSCC__` macro to use the `sds_alloc()` and `sds_free()` functions when compiling source code with sdscc/sds++, and `malloc()` and `free()` when using other compilers.

```
#ifdef __SDSCC__
#include <stdlib.h>
#include "sds_lib.h"
#define malloc(x) (sds_alloc(x))
#define free(x) (sds_free(x))
#endif
```

In the example below, the `__SDSVHLS__` macro is used to guard code in a function definition that differs depending on whether it is used by Vivado HLS to generate hardware or used in a software implementation.

```
#ifdef __SDSVHLS__
void mmult(ap_axiu<32,1,1,1> A[A_NROWS*A_NCOLS],
           ap_axiu<32,1,1,1> B[A_NCOLS*B_NCOLS],
           ap_axiu<32,1,1,1> C[A_NROWS*B_NCOLS])
#else
void mmult(float A[A_NROWS*A_NCOLS],
           float B[A_NCOLS*B_NCOLS],
           float C[A_NROWS*B_NCOLS])
#endif
```

# System Options

## -apm

Insert an AXI Performance Monitor (APM) IP block to monitor all generated hardware/software interfaces. Within the SDSoC IDE, in the Debug Perspective, you can activate the APM prior to running your application by clicking the **Start** button within the Performance Counters View. For more information on the SDSoC IDE, see *SDSoC Environment Tutorial: Introduction* (UG1028).

## -disable-ip-cache

Do not use a cache of pre-synthesized IP cores. The use of IP caching for synthesis reduces the overall build time by eliminating the synthesis step for static IP cores. If the resources required to implement the hardware system exceeds available resources by a small amount, the –`disable-ip-cache` option forces SDSoC to synthesize all IP cores in the context of the design and may reduce resource usage enough to enable implementation.

## -dm-sharing <n>

The `-dm-sharing <n>` option enables exploration of data mover sharing capabilities if the initial schedule can be relaxed. The level of sharing defaults to 0 (low) if not specified. Other values are 1 (medium), 2 (high) and 3 (maximum – schedule can be relaxed infinitely). For example, to enable maximum data mover sharing, add the `sdscc -dm-sharing 3` option.

## -dmclkid <n>

Set the data motion network clock ID to `<n>`, where `<n>` has one of the values listed in the table below. (You can use the command `sdscc -sds-pf-info platform_name` to display the information about the platform.) If the `dmclkid` option is not specified, the default value for the platform is used. Use the command `sdscc -sds-pf-list` to list available platforms and settings.

| Platform | Value of <n> |
|---|---|
| zc702 | 0 – 166 MHz |
| | 1 – 142 MHz |
| | 2 – 100 MHz |
| | 3 – 200 MHz |
| zc706 | 0 – 166 MHz |
| | 1 – 142 MHz |
| | 2 – 100 MHz |
| | 3 – 200 MHz |
| zed and microzed | 0 – 166 MHz |
| | 1 – 142 MHz |
| | 2 – 100 MHz |
| | 3 – 200 MHz |
| zybo | 0 – 25 MHz |
| | 1 – 100 MHz |
| | 2 – 125 MHz |
| | 3 – 50 MHz |
| zcu102 | 0 – 100 MHz |
| | 1 – 150 MHz |
| | 2 – 200 MHz |
| | 3 – 300 MHz |

## -emulation <mode>

Generate files required to run emulation of the system using QEMU for the processing subsystem and the Vivado Logic Simulator for the programmable logic. The <mode> specifies the type of simulation models created for the PL, `debug` or `optimized`. In the same directory that you ran `sds++`, type the command `sdsoc_emulator` to run the emulation in the current shell.

## -impl-strategy <strategy_name>

Specify the Vivado implementation strategy name to use instead of the default strategy, for example Performance_Explore. The strategy name can be found in the **Vivado Implementation Settings** dialog in the **Strategy** menu, and the strategies are described in *Vivado Design Suite User Guide: Implementation* (UG904). When creating the Tcl file for synthesis and implementation, this command is added: `set_property strategy <strategy_name> [get_runs impl_1]`.

## -instrument-stub

The `-instrument-stub` option instruments the generated hardware function stubs with calls to the counter function `sds_clock_counter()`. When a hardware function stub is instrumented, the time required to call send and receive functions, as well as the time spent for waits, is displayed for each call to the function.

## -maxjobs <n>

The `-maxjobs <n>` option specifies the maximum number of jobs used for Vivado synthesis. The default is the number of cores divided by 2.

## -maxthreads <n>

The `-maxthreads <n>` option specifies the number of threads used in multithreading to speed up certain tasks, including Vivado placement and routing. The number of threads can be an integer from 1 to 8. The default value is 4, but the tools will not use more threads than the number of cores on the machine. Also, a general limit based on the OS applies to all tasks.

## -mno-bitstream

Do not generate the bitstream for the design used to configure the programmable logic (PL). Normally a bitstream is generated by running the Vivado implementation feature, which can be time-consuming with runtimes ranging from minutes to hours depending on the size and complexity of the design. This option can be used to disable this step when iterating over flows that do not impact the hardware generation. The application ELF is compiled before bitstream generation.

## -mno-boot-files

Do not generate the SD card image in the folder `sd_card`. This folder includes your application ELF and files required to boot the device and bring up the specified OS. This option disables the creation of the `sd_card` folder in case you would like to preserve an earlier version of this folder.

## -rebuild-hardware

When building a software-only design with no functions mapped to hardware, `sdscc` uses a pre-built bitstream if available within the platform, but use this option to force a full system build.

## -sdcard <data_directory>

Specify an optional directory containing additional files to include in the SD card image.

## -synth-strategy <strategy_name>

Specify the Vivado synthesis strategy name to use instead of the default strategy, for example Flow_RuntimeOptimized. The strategy name can be found in the **Vivado Synthesis Settings** dialog in the **Strategy** menu, and the strategies are described in *Vivado Design Suite User Guide: Synthesis* (UG901). When creating the Tcl file for synthesis and implementation, this command is added: `set_property strategy <strategy_name> [get_runs synth_1]`.

## -trace

The `-trace` option inserts hardware and software infrastructure into the design to enable tracing functionality.

### -trace-buffer depth

The `-trace-buffer` option specifies the trace buffer depth, which must be at least 16 and a power of 2. If this option is not specified, the default value of 1024 is used.

### -trace-no-sw

The `-trace-no-sw` option inserts hardware trace monitors into the design without instrumenting the software when enabling tracing functionality.

## Compiler Toolchain Support

The SDSoC environment uses the same GNU ARM cross-compiler toolchains included with the Xilinx Software Development Kit (SDK). The Linaro-based GCC compiler toolchains support the Zynq®-7000 and Zynq UltraScale+™ family devices, and this section includes additional information on toolchain usage that might be useful.

When compiling and linking applications, use only object files and libraries built using the same compiler toolchain and options as those used by the SDSoC environment. All SDSoC provided software libraries and software components (Linux kernel, root filesystem, BSP libraries, and other pre-built libraries) are built with the included toolchains. If you use sdscc or sds++ to compile object files, the tools automatically insert a small number of options, and if you invoke the underlying toolchains, you must use the same options. For example, if you use a different Zynq-7000 floating-point application binary interface (ABI), your binary objects are incompatible and cannot be linked with SDSoC Zynq-7000 binary objects and libraries.

The table below summarizes sdscc and sds++ usage of Zynq-7000 toolchains and options. Where options are listed, you only need to specify them if you use the toolchain gcc and g++ commands directly instead of invoking sdscc and sds++.

| Usage | Description |
|---|---|
| Zynq-7000 ARM bare-metal compiler and linker options | -mcpu=cortex-a9 -mfpu=vfpv3 -mfloat-abi=hard |
| Zynq-7000 ARM bare-metal linker options | -Wl,--build-id=none -specs=<specfile><br><br>where the <specfile> contains<br><br>*startfile:<br><br>crti%O%s crtbegin%O%s |
| Zynq-7000 ARM bare-metal compiler | ${SDSOC_install}/SDK/gnu/aarch32/<host>/gcc-arm-none-eabi/bin<br><br>Toolchain prefix: arm-none-eabi<br><br>gcc executable: arm-none-eabi-gcc |

| Usage | Description |
|---|---|
| | g++ executable: arm-none-eabi-g++ |
| Zynq-7000 SDSoC bare-metal software (lib, include) | ${SDSOC_install}/aarch32-none |
| Zynq-7000 ARM Linux compiler | ${SDSOC_install}/SDK/gnu/aarch32/<host>/gcc-arm-linux-gnueabi/bin<br>Toolchain prefix: arm-linux-gnueabihf-<br>gcc executable: arm-linux-gnueabihf-gcc<br>g++ executable: arm-linux-gnueabihf-g++ |
| Zynq-7000 SDSoC Linux software (lib, include) | ${SDSOC_install}/aarch32-linux |

The table below summarizes sdscc and sds++ usage of Zynq UltraScale+ Cortex-A53 toolchains and options. Where options are listed, you only need to specify them if you use the toolchain gcc and g++ commands directly instead of invoking sdscc and sds++.

| Usage | Description |
|---|---|
| Zynq UltraScale+ ARM bare-metal compiler and linker options | Use default options |
| Zynq UltraScale+ ARM bare-metal linker options | -Wl,--build-id=none |
| Zynq UltraScale+ ARM bare-metal compiler | ${SDSOC_install}/SDK/gnu/aarch64/<host>/aarch64-none/bin<br>Toolchain prefix: aarch64-none-elf<br>gcc executable: aarch64-none-elf-gcc<br>g++ executable: aarch64-none-elf-g++ |
| Zynq UltraScale+ SDSoC bare-metal software (lib, include) | ${SDSOC_install}/aarch64-none |
| Zynq UltraScale+ ARM Linux compiler | ${SDSOC_install}/SDK/gnu/aarch64/<host>/aarch64-linux/bin<br>Toolchain prefix: aarch64-linux-gnu-<br>gcc executable: aarch64-linux-gnu-gcc<br>g++ executable: aarch64-linux-gnu-g++ |
| Zynq UltraScale+ SDSoC Linux software (lib, include) | ${SDSOC_install}/aarch64-linux |

The table below summarizes sdscc and sds++ usage of Zynq UltraScale+ Cortex-R5 toolchains and options. Where options are listed, you only need to specify them if you use the toolchain gcc and g++ commands directly instead of invoking sdscc and sds++.

| Usage | Description |
|---|---|
| Zynq UltraScale+ ARM bare-metal compiler and linker options | Use default options |
| Zynq UltraScale+ ARM bare-metal linker options | -Wl,--build-id=none |
| Zynq UltraScale+ ARM bare-metal compiler | ${SDSOC_install)/SDK/gnu/armr5/<host>/gcc-arm-none-eabi/bin<br><br>Toolchain prefix: armr5-none-eabi<br><br>gcc executable: armr5-none-eabi-gcc<br><br>g++ executable: armr5-none-eabi-g++ |
| Zynq UltraScale+ SDSoC bare-metal software (lib, include) | ${SDSOC_install}/armr5-none |

When using sdscc and sds++ to compile Zynq-7000 source files, be aware that SDSoC tools that process and analyze source files issue errors if they contain NEON instrinsics. If hardware accelerator (or caller) source files contain NEON intrinsics, guard them using the `__SDSCC__` and `__SDSVHLS__` macros. For source files that don't contain hardware accelerators or callers but do use NEON intrinsics, you can either compile them directly using the GNU toolchain and link the objects with sds++, or you can add the sdscc/sds++ command line option **-mno-ir** for these source files. The option prevents clang-based tools from being invoked to create an intermediate representation (IR) used in analysis, because we know they are not required (i.e., no accelerators or callers). For the latter solution, if you are using the SDSoC environment, you can apply the option on a per-file basis by right-clicking the source file, select **Properties** and go to the **Settings** dialog under **C/C++ Build Settings→Settings**.

# Exporting a Library for GCC

This chapter demonstrates how to use the `sdscc/sds++` compiler to build a library with entry points into hardware functions implemented in programmable logic. This library can later be linked into applications using the standard GCC linker for Zynq®-7000 All Programmable SoCs. In addition to the library, `sdscc` generates a complete boot image that includes an FPGA bitstream containing the hardware functions and data motion network. You can then develop software applications that call into the hardware functions (and fixed hardware) using the standard GCC toolchains. Such code will compile quickly and will not change the hardware. You are still targeting the same hardware system and using the `sdscc`-generated boot environment, but you are then free to develop your software using the GNU toolchain in the software development environment of your choice.

**NOTE:** *In the current SDSoC release, libraries are not thread-safe, so they must be called into from a single thread within an application, which could consist of many threads and processes.*

**NOTE:** *In the current SDSoC release, shared libraries can be created only for Linux target applications.*

## Building a Shared Library

To build a shared library, `sdscc` requires at least one accelerator. This example provides three entry points into two hardware accelerators: a matrix multiplier and a matrix adder. You can find these files in the `samples/libmatrix/build` directory.

- `mmult_accel.cpp` – Accelerator code for the matrix multiplier
- `mmult_accel.h` – Header file for the matrix multiplier
- `madd_accel.cpp` – Accelerator code for the matrix adder
- `madd_accel.h` – Header file for the matrix adder
- `matrix.cpp` – Code that calls the accelerators and determines the data motion network
- `matrix.h` – Header file for the library

The `matrix.cpp` file contains functions that define the accelerator interfaces as well as how the hardware functions communicate with the platform (i.e., the data motion networks between platform and accelerators). The function `madd` calls a single matrix adder accelerator, and the function `mmult` calls a single matrix multiplier accelerator. Another function `mmultadd` is implemented using two hardware functions, with the output of the matrix multiplier connected directly to the input of the matrix adder.

```
/* matrix.cpp */
#include "madd_accel.h"
```

```
#include "mmult_accel.h"

void madd(float in_A[MSIZE*MSIZE], float in_B[MSIZE*MSIZE], float
out_C[MSIZE*MSIZE])
{
  madd_accel(in_A, in_B, out_C);
}

void mmult(float in_A[MSIZE*MSIZE], float in_B[MSIZE*MSIZE], float
out_C[MSIZE*MSIZE])
{
  mmult_accel(in_A, in_B, out_C);
}

void mmultadd(float in_A[MSIZE*MSIZE], float in_B[MSIZE*MSIZE], float
in_C[MSIZE*MSIZE],
float out_D[MSIZE*MSIZE])
{
  float tmp[MSIZE * MSIZE];

  mmult_accel(in_A, in_B, tmp);
  madd_accel(tmp, in_C, out_D);
}
```

The `matrix.h` file defines the function interfaces to the shared library, and will be included in the application source code.

```
/* matrix.h */
#ifndef MATRIX_H_
#define MATRIX_H_

#define MSIZE 16

void madd(float in_A[MSIZE*MSIZE], float in_B[MSIZE*MSIZE], float
out_C[MSIZE*MSIZE]);

void mmult(float in_A[MSIZE*MSIZE], float in_B[MSIZE*MSIZE], float
out_C[MSIZE*MSIZE]);

void mmultadd(float in_A[MSIZE*MSIZE], float in_B[MSIZE*MSIZE], float
in_C[MSIZE*MSIZE],
float out_D[MSIZE*MSIZE]);

#endif /* MATRIX_H_ */
```

The `Makefile` shows how the project is built by specifying that the functions `mmult_accel`, `madd`, and `mmult_add` must be implemented in programmable logic.

```
SDSFLAGS = \
    -sds-pf ${PLATFORM} \
    -sds-hw mmult_accel mmult_accel.cpp -sds-end \
    -sds-hw madd_accel madd_accel.cpp -sds-end
```

As is the case for normal shared libraries, object files are generated with position independent code (`-fpic` option).

```
sds++ ${SDSFLAGS} -c -fpic -o mmult_accel.o mmult_accel.cpp
    sds++ ${SDSFLAGS} -c -fpic -o madd_accel.o madd_accel.cpp
    sds++ ${SDSFLAGS} -c -fpic -o matrix.o matrix.cpp
```

To link the objects files we also follow the standard method and use the `-shared` switch.

```
sds++ ${SDSFLAGS} -shared -o libmatrix.so mmult_accel.o madd_accel.o
matrix.o
```

After building the project, these files will be generated

- `libmatrix.so` – Shared library suitable for linking using GCC and for runtime use
- `sd_card` – Directory containing an SD card image for booting the board

### Delivering a Library

The following structure allows compiling and linking into applications using GCC in standard ways.

```
<path_to_library>/include/matrix.h
<path_to_library>/lib/libmatrix.so
<path_to_library>/sd_card
```

**NOTE:** *The `sd_card` folder is to be copied into an SD card and used to boot the board. This image includes a copy of the `libmatrix.so` file that is used at runtime.*

## Compiling and Linking Against a Library

The following is an example of using the library with a GCC compiler. The library is used by including the header file `matrix.h` and then calling the necessary library functions.

```
/* main.cpp (pseudocode) */
#include "matrix.h"

int main(int argc, char* argv[])
{
  float *A, *B, *C, *D;
  float *J, *K, *L;
  float *X, *Y, *Z;
  ...
  mmultadd(A, B, C, D);
  ...
  mmult(J, K, L);
  ...
  madd(X, Y, Z);
  ...
}
```

To compile against a library, the compiler needs the header file. The path to the header file is specified using the `-I` switch. You can find example files in the `samples/libmatrix/use` directory.

**NOTE:** *For explanation purposes, the code above is only pseudocode and not the same as the* `main.cpp` *file in the directory. The file has more code that allows full compilation and execution.*

```
gcc -I <path_to_library>/include -o main.o main.c
```

To link against the library, the linker needs the library. The path to the library is specified using the `-L` switch. Also, ask the linker to link against the library using the `-l` switch.

```
gcc -I <path_to_library>/lib -o main.elf main.o -lmatrix
```

For detailed information on using the GCC compiler and linker switches refer to the GCC documentation.

### Use a library at runtime

At runtime, the loader will look for the shared library when loading the executable. After booting the board into a Linux prompt and before executing the ELF file, add the path to the library to the `LD_LIBRARY_PATH` environment variable. The `sd_card` created when building the library already has the library, so the path to the mount point for the sd_card must be specified.

For example, if the sd_card is mounted at `/mnt`, use this command:

```
export LD_LIBRARY_PATH=/mnt
```

## Exporting a Shared Library

The following steps demonstrate how to export an SDSoC environment shared library with the corresponding SD card boot image using the SDSoC environment GUI.

1. Select **File→New→SDSoC Project** to bring up the **New Project** dialog box.
2. Create a new SDSoC project.
    a. Type `libmatrix` in the **Project name** field.
    b. Select **Platform** to be `zc702`.
    c. Put a checkmark on the **Shared Library** checkbox.
    d. Click **Next**.
3. Choose the application template.
    a. Select `Matrix Shared Library` from the **Available Templates**.
    b. Click **Finish**.

A new SDSoc shared library application project called `libmatrix` is created in the **Project Explorer** view. The project includes two hardware functions `mmult_accel` and `madd_accel` that are visible in the **SDSoC Project Overview**.

4. Build the library.
   a. In the **Project Explorer** view, select the `libmatrix` project.
   b. Select **Project→Build Project**.

   After the build completes, there will be a boot SD card image under the Debug (or current configuration) folder.

Chapter 13

# Compiling Your OpenCL Kernel Using the Xilinx OpenCL Compiler (xocc)

The Xilinx® OpenCL™ Compiler (xocc) is a standalone command line utility for compiling an OpenCL kernel supporting all flows in the SDSoC™ environment. It provides a mechanism for command line users to compile their kernels, which is ideal for compiling host applications and kernels using a makefile.

Following are details of xocc command line format and options.

Syntax:

```
xocc [options] <input_file>
```

**Table 3: XOCC Options**

| Option | Valid Values | Description |
|---|---|---|
| **--platform** <*arg*> | Supported acceleration platforms by Xilinx and third-party board partners | Required<br>Set target Xilinx device. See *SDx Environments Release Notes, Installation, and Licensing Guide* (UG1238) for all supported devices. |
| **--list_xdevices** | N/A | Lists the supported devices. |
| **--target** <*arg*> | [**sw_emu** \| **hw_emu** \| **hw**] | Specify a compile target.<br>• **sw_emu**: CPU emulation<br>• **hw_emu**: Hardware emulation<br>• **hw**: Hardware<br>Default: **hw**<br>**NOTE:** *Without the **-c** or **-l** option, xocc is run in build mode, an `.xclbin` file is generated.* |
| **--compile** | N/A | Optional |

| Option | Valid Values | Description |
|---|---|---|
| | | Run xocc in compile mode, generate `.xo` file. |
| **--link** | N/A | Optional<br>Run xocc in link mode, link `.xo` input files, generate `.xclbin` file. |
| **--kernel** <*arg*> | Kernel to be compiled from the input `.cl` or `.c/.cpp` kernel source code | Required for C/C++ kernels<br>Optional for OpenCL kernels<br>Compile/build only the specified kernel from the input file. Only one **-k** option is allowed per command.<br>**NOTE:** *When an OpenCL kernel is compiled without the* ***-k*** *option, all the kernels in the input file are compiled.* |
| **--output** <*arg*> | File name with `.xo` or `.xclbin` extension depending on mode | Optional<br>Set output file name.<br>Default:<br>`a.xo` for compile mode<br>`a.xclbin` for link and build mode |
| **--version** | N/A | Prints the version and build information. |
| **--help** | N/A | Print help. |
| **--define** <*arg*> | Valid macro name and definition pair<br><*name*>=<*definition*> | Predefine name as a macro with definition. This option is passed to the openCL preprocessor. |
| **--include** <*arg*> | Directory name that includes required header files | Add the directory to the list of directories to be searched for header files. This option is passed to the SDSoC compiler preprocessor. |
| **--kernel_frequency** | Frequency (MHz) of the kernel. | Sets a user defined clock frequency in MHz for a the |

| Option | Valid Values | Description |
|---|---|---|
| | | kernel overriding a default value from the DSA. |
| **--nk** <*arg*> | `<kernel_name>: <compute_units>`<br><br>(for example, **foo:2**) | N/A in compile mode<br><br>Optional in link mode<br><br>Instantiate the specified number of compute units for the given kernel in the `.xclbin` file.<br><br>Default: One compute unit per kernel. |
| **--pk** <*arg*> | [**kernel_name**\|**all**] : [**none**\|**stream**\|**pipe**\|**memory**] | Optional<br><br>Set a stall profile type for the given kernel(s)<br><br>Default: none |
| **--max_memory_ports** <*arg*> | [**all** \| <*kernel_name*>] | Optional<br><br>Set the maximum memory port property for all kernels or a given kernel. |
| **--memory_port_data_width** <*arg*> | [**all** \| <*kernel_name*>]:<*width*> | Set the specified memory port data width for all kernels or a given kernel. Valid width values are 32, 64, 128, 256, and 512. |
| **--optimize**<*arg*> | Valid optimization levels: 0, 1, 2, 3, s, quick<br><br>example: --optimize2 | These options control the default optimizations performed by the Vivado® hardware synthesis engine.<br><br>**NOTE:** *Familiarity with the Vivado tool suite is recommended in order to make the most use of these settings.*<br><br>• 0: Default optimization. Reduce compilation time and make debugging produce the expected results. |

| Option | Valid Values | Description |
|---|---|---|
| | | • 1: Optimize to reduce power consumption. This takes more time to compile the design.<br>• 2: Optimize to increase kernel speed. This option increases both compilation time and the performance of the generated code.<br>• 3: This is the highest level of optimization. This option provides the highest level performance in the generated code, but compilation time may increase considerably.<br>• s: Optimize for size. This reduces the logic resources for the kernel<br>• quick: Quick compilation for fast run time. This may result in reduced performance and a greater use of resources in the hardware implementation. |
| **--xp** | Refer to the following table, XP Parameters. | Specify detailed parameter and property settings in the Vivado tool suite used to implement the FPGA hardware.<br><br>**NOTE:** *Familiarity with the Vivado tool suite is recommended in order to make the most use of these parameters.* |
| **--debug** | N/A | Generate code for debugging. |
| **--log** | N/A | Creates a log for in the current working directory. |
| **--message-rules** <*arg*> | Message rule file name | Optional - |

| Option | Valid Values | Description |
|---|---|---|
| | | Specify a message rule file with message controlling rules. See *Using the Message Rule File* chapter for more details. |
| **--report** <*arg*> | Generate [estimate \| system] reports | Generate a report type specified by <*arg*>.<br>estimate: Generate estimate report in `report_estimate.xtxt`<br>system: Generate the estimate report and detailed hardware reports in report directory |
| **--save-temps** | N/A | Save intermediate files/ directories created during the compilation and build process. |
| **--report_dir** <*arg*> | Directory | Specify a report directory. If the --report option is specified, the default is to generate all reports in the current working directory (cwd). |
| **--log_dir** <*arg*> | Directory | Specify a log directory. If the --log option is specified, the default is to generate the log file in the current working directory (cwd). |
| **--temp_dir** <*arg*> | Directory | Specify a log directory. If the --save-temps option is specified, the default is to create the temporary compilation and build files in the current working directory (cwd). |
| **--export_script** | N/A | This option allows detailed control of the Vivado tool suite used to implement the FPGA hardware. |

| Option | Valid Values | Description |
|---|---|---|
| | | **NOTE:** *Familiarity with the Vivado tool suite is recommended in order to make the most use of the Tcl file generated by this option.*<br><br>Generates the Tcl script used to execute Vivado HLS `<kernel_name>.tcl` but halts before Vivado HLS starts. The expectation is for the script to be modified and used with the `--custom_script` option.<br><br>Not supported for `-t sw_emu` with OpenCL kernels. |
| **--custom_script** | `<kernel_name>:<path to kernel Tcl file>` | Intended for use with the `<kernel_name>.tcl` file generated with option `-export_script`.<br><br>This option allows you to customize the Tcl file used to create the kernel and execute using the customize version of the script. |
| **--jobs** `<arg>` | Number of parallel jobs | Optional<br><br>This option allows detailed control of the Vivado tool suite used to implement the FPGA hardware.<br><br>**NOTE:** *Familiarity with the Vivado tool suite is recommended in order to make the most use of the Tcl file generated by this option.* |

| Option | Valid Values | Description |
|---|---|---|
| | | Specify the number of parallel jobs to be passed to the Vivado tool suite for implementation. Increasing the number of jobs allows the hardware implementation step to spawn more parallel processes and complete faster. |
| **--lsf** <*arg*> | `bsub` command line to pass to LSF cluster<br>**NOTE:** *This argument is required.* | Optional<br>Use IBM Platform Load Sharing Facility (LSF) for Vivado implementation. |
| **input file** | OpenCL or C/C++ kernel source file | Compile kernels into a `.xo` or `.xclbin` file depending on the xocc mode. |
| **--sp** | <*kernel_compute_unit_name*>.<br><*kernel_port*>:<*system_port*><br>(for example, k1.M_AXI_GMEM:bank0) | Supported for unified platform. System port mapping. This will replace `map_connect` for unified platform. |
| **--clkid** | index number | Supported for unified platform. Passes the index number to `sdx_link`. Each index available from selected platform has a different default clock frequency. |
| **--remote_ip_cache** | directory | Supported for unified platform. Specify a location for a remote IP cache. Passed to `vpl`. |
| **--no_ip_cache** | X | Display verbose/debug information (including output from Vivado runs). |

**IMPORTANT:** *All examples in the SDSoC installation use Makefile to compile OpenCL applications with gcc and xocc commands, which can be used as references for compiling user applications using xocc.*

# XP Parameters

Use the `--xp` switch to specify parameter values in SDSoC™. These parameters allow fine grain control over the hardware generated by SDSoC and the hardware emulation process.

**IMPORTANT:** *Familiarity with the Vivado™ tool suite is recommended in order to make the most use of these parameters.*

Parameters are specified as `parm:<parameter>=<value>`. For example:

```
xocc -xp param:compiler.enableDSAIntegrityCheck=true
-xp param:prop:kernel.foo.kernel_flags="-std=c++0x"
```

The `-xp` command option may be specified multiple times in a single `xocc` invocation, or the value(s) may be specified in a `xocc.ini` file with each option specified on a separate line (without `--xp` switch).

```
param:prop:solution.device_repo_paths=../dsa
param:compiler.preserveHlsOutput=1
```

Upon invocation, xocc first looks for an `xocc.ini` file in the `$HOME/.Xilinx/sdx` directory. If the file does not exist there, xocc will then look for it in the current working directory. If the same `--xp` parameter value is specified in both the command line and `xocc.ini` file, the command line value will be used.

The following table lists the `-xp` parameters and their values.

**Table 4: XP Parameters**

| Parameter Name | Type | Default Value | Description |
|---|---|---|---|
| **param:compiler. enableDSAIntegrityCheck** | Boolean | False | Enables the DSA Integrity Check.<br><br>If this value is set to True, and SDSoC detects a DSA which has been modified outside the of the Vivado® tool suite SDSoC halts operation. |
| **param:compiler. errorOnHoldViolation** | Boolean | True | Error out if there is hold violation. |
| **param:compiler. maxComputeUnits** | Int | `-1` | The maximum compute units allowed in the system. Any positive value will overwrite the **numComputeUnits** setting in the DSA. |
| **param:hw_em.debugLevel** | String | OFF | The debug level of the simulator. Option OFF is used for optimized run times, BATCH is for batch runs and GUI for use in GUI-mode |
| **param:hw_em. enableProtocolChecker** | Boolean | False | Enables the AXI protocol checker during HW emulation. This is used to confirm |

| Parameter Name | Type | Default Value | Description |
|---|---|---|---|
| | | | the accuracy of any AXI interfaces in the design. |
| **param:compiler. interfaceLatency** | Int | `-1` | This option specifies the expected latency on the kernel AXI bus, the number of clock cycles from when bus access is requested until it is granted. |
| **param:compiler. xclDataflowFifoDepth** | Int | `-1` | Specifies the depth of FIFOs used in kernel dataflow region. |
| **param:compiler. interfaceWrOutstanding** | Int Range | 0 | Specifies how many outstanding writes to buffer are on the kernel AXI interface. Values are 1 through 256. |
| **param:compiler. interfaceRdOutstanding** | Int Range | 0 | Specifies how many outstanding reads to buffer are on the kernel AXI interface. Values are 1 through 256. |
| **param:compiler. interfaceWrBurstLen** | Int Range | 0 | Specifies the expected length of AXI write bursts on the kernel AXI interface. This is used with option **compiler.interfaceWrOutstanding** to determine the hardware buffer sizes. Values are 1 through 256. |
| **param:compiler. interfaceRdBurstLen** | Int Range | 0 | Specifies the expected length of AXI read bursts on the kernel AXI interface. This is used with option **compiler.interfaceRdOutstanding** to determine the hardware buffer sizes. Values are 1 through 256. |
| **misc:map_connect=<type>. kernel.<kernael_name>. <kernel_AXI_interface>.core. OCL_REGION_0.<dest_port>** | String | `<empty>` | Used to map AXI interfaces from a kernel to DDR memory banks.<br>• `<type>` is add or remove.<br>• `<kernel_name>` is the name of the kernel.<br>• `<dest_port>` is DDR memory bank M00_AXI, M01_AXI, M02_AXI or M03_AXI. |
| **prop:kernel.<kernel_name>. kernel_flags** | String | `<empty>` | Sets specific compile flags on kernel `<kernelk_name>`. e.g. |
| **prop:solution. device_repo_path** | String | `<empty>` | Specifies the path to the DSA repository. |

| Parameter Name | Type | Default Value | Description |
|---|---|---|---|
| **prop:solution.hls_pre_tcl** | String | <empty> | Specifies the path to a Vivado HLS Tcl file, which is executed before the C code is synthesized. This allows Vivado HLS configuration settings to be applied prior to synthesis. |
| **prop:solution.hls_post_tcl** | String | <empty> | Specifies the path to a Vivado HLS Tcl file, which is executed after the C code is synthesized. |
| **prop:solution. kernel_compiler_margin** | Float | 12.5% of the kernel clock period. | The clock margin in ns for the kernel. This value is substracted from the kernel clock period prior to synthesis to provide some margin for P&R delays. |
| **vivado_prop:<object_type>.<object_name>.<prop_name>** | Various | Various | This allows you to specify any property used in the Vivado hardware compilation flow.<br>Object_type is run\|fileset\|file\|project<br>The `object_name` and `prop_name` values are described in *Vivado Design Suite Properties Reference Guide*, (UG912)<br>Examples:<br>`vivado_prop:run.impl_1.{STEPS.PLACE_DESIGN.ARGS.MORE OPTIONS}={-fanout_opt}`<br>`vivado_prop:fileset.current.top=foo`<br>**NOTE:** *For `object_type` file, `current` is not supported*<br>**NOTE:** *For object type `run` the special value of `__KERNEL__` can be used to specify run optimization settings for ALL kernels, instead of having to specify them one by one* |

# Running Software and Hardware Emulation in XOCC Flow

In the XOCC/Makefile flow, users manage compilation and execution of host code and kernels outside the Xilinx® SDSoC™ development environment. Follow the steps below to run software and hardware emulation:

1. Create the emulation configuration file.

   For software or hardware emulation, the runtime library needs to know what devices and how many to emulate. This information is provided to the runtime library by an emulation configuration file. SDSoC provides a utility, `emconfigutil` to automate creation of the emulation configuration file. The following are details of the `emconfigutil` command line format and options:

   | Option | Valid Values | Description |
   |---|---|---|
   | `--xdevice` | Target device | Required: Set target device. Check Appendix B for all supported devices |
   | `--nd` | Any positive integer | Optional: Number of devices. Default is 1. |
   | `--od` | Valid directory | Optional: Output directory, `emconfig.json` file must be in the same directory as the host executable. |
   | `--xp` | Valid Xilinx parameters and properties | Optional: Specify additional parameters and properties. For example: `--xp prop:solution.device_repo_paths=my_dsa_path` Sets the search path for the device specified in --xdevice option. |
   | `-h` | NA | Print help messages |

   The `emconfigutil` command creates the configuration file `emconfig.json` in the output directory.

   The `emconfig.json` file must be in the same directory as the host executable.

   The following example creates a configuration file targeting two `xilinx:adm-pcie-7v3:1ddr:3.0` devices.

   ```
   $emconfigutil --xdevice xilinx:adm-pcie-7v3:1ddr:3.0 --nd 2
   ```

2. Set `XILINX_SDX` environment variable

The XILINX_SDX environment needs to be set and pointed to the SDSoC installation path for the emulation to work. Below are examples assuming SDSoC is installed in /opt/Xilinx/SDx/2017.2

C Shell:

```
setenv XILINX_SDX /opt/Xilinx/SDx/2017.2
```

Bash:

```
export XILINX_SDX=/opt/Xilinx/SDx/2017.2
```

3. Set emulation mode

   Setting XCL_EMULATION_MODE environment variable to sw_emu or hw_emu changes the application execution to emulation mode (sw_emu for software emulation and hw_emu for hardware emulation) so that the runtime looks for the file emconfig.json in the same directory as the host executable and reads in the target configuration for the emulation runs.

   C Shell:

   ```
   setenv XCL_EMULATION_MODE sw_emu
   ```

   Bash:

   ```
   export  XCL_EMULATION_MODE=sw_emu
   ```

   Unsetting the XCL_EMULATION_MODE environment variable will turn off the emulation mode.

4. Run CPU and hardware emulation

   With the configuration file emconfig.json and XCL_EMULATION_MODE set to true, execute the host application with proper arguments to run CPU and hardware emulation:

   ```
   $./host.exe kernel.xclbin
   ```

XILINX
ALL PROGRAMMABLE™

*Chapter 14*

# SDSoC Environment API

This chapter describes functions in `sds_lib` available for applications developed in the SDSoC environment.

**NOTE:** *To use the library,* `#include "sds_lib.h"` *in source files. You must include* `stdlib.h` *before including* `sds_lib.h` *to provide the* `size_t` *type declaration.*

The SDSoC™ environment API provides functions to map memory spaces, and to wait for asynchronous accelerator calls to complete.

**`void sds_wait(unsigned int id)`**

Wait for the first accelerator in the queue identified by `id`, to complete. The recommended alternative is the use `#pragma SDS wait(id)`, as described in Asynchronous Function Execution.

**`void *sds_alloc(size_t size)`**

Allocate a physically contiguous array of `size` bytes.

**`void *sds_alloc_non_cacheable(size_t size)`**

Allocate a physically contiguous array of `size` bytes that is marked as non-cacheable. Memory allocated by this function is not cached in the processing system. Pointers to this memory should be passed to a hardware function in conjunction with

```
#pragma SDS data mem_attribute (p:NON_CACHEABLE)
```

**`void sds_free(void *memptr)`**

Free an array allocated through `sds_alloc()`

**`void *sds_mmap(void *physical_addr, size_t size, void *virtual_addr)`**

Create a virtual address mapping to access a memory of `size` bytes located at physical address `physical_addr`.

- `physical_addr`: physical address to be mapped.
- `size`: size of physical address to be mapped.
- `virtual_addr`:
  - If not null, it is considered to be the virtual-address already mapped to the `physical_addr`, and `sds_mmap` keeps track of the mapping.
  - If null, `sds_mmap` invokes `mmap()` to generate the virtual address, and `virtual_addr` is assigned this value.

**`void *sds_munmap(void *virtual_addr)`**

Unmaps a virtual address associated with a physical address created using `sds_mmap()`.

**`unsigned long long sds_clock_counter(void)`**

Returns the value associated with a free-running counter used for fine grain time interval measurements.

**`unsigned long long sds_clock_frequency(void)`**

Returns the frequency (in ticks/second) associated with the free-running counter that is read by calls to `sds_clock_counter`. This is used to translate counter ticks to seconds.

*Appendix A*

# 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

## References

These documents provide supplemental material useful with this guide:


1. *SDx Environments Release Notes, Installation, and Licensing Guide* (UG1238)
2. *SDSoC Environment User Guide* (UG1027)
3. *SDSoC Environment Optimization Guide* (UG1235)
4. *SDSoC Environment Tutorial: Introduction* (UG1028)
5. *SDSoC Environment Platform Development Guide* (UG1146)
6. SDSoC Development Environment web page
7. *UltraFast Embedded Design Methodology Guide* (UG1046)
8. *ZC702 Evaluation Board for the Zynq-7000 XC7Z020 All Programmable SoC User Guide* (UG850)
9. *Vivado Design Suite User Guide: High-Level Synthesis* (UG902)
10. *PetaLinux Tools Documentation: Workflow Tutorial* (UG1156)
11. Vivado® Design Suite Documentation
12. *Vivado Design Suite User Guide: Creating and Packaging Custom IP* (UG1118)

# Please Read: Important Legal Notices



## AUTOMOTIVE APPLICATIONS DISCLAIMER