

# **SDSoC Environment User Guide**

## ***An Introduction to the SDSoC Environment***

UG1028 (v2016.2) July 13, 2016

---

## Revision History

The following table shows the revision history for this document.

Date	Version	Revision
07/13/2016	2016.2	<ul style="list-style-type: none"><li>Added chapter for Communication to Platform I/O Streams.</li><li>Updates to reflect changes to software.</li></ul>
05/11/2016	2016.1	Updates to reflect changes to software.

# Table of Contents

Revision History .....	2
Table of Contents .....	3
<b>Chapter 1: Introduction .....</b>	<b>5</b>
User Design Flow .....	5
System Requirements .....	7
Obtaining and Managing a License .....	7
Downloading .....	8
Installing.....	8
Validating Installation .....	10
<b>Chapter 2: Tutorial: Creating, Building and Running a Project .....</b>	<b>18</b>
Learning Objectives.....	18
Creating a New Project.....	18
Marking Functions for Hardware Implementation.....	22
Building a Design with Hardware Accelerators .....	24
Running the Project .....	25
Questions and Additional Exercises .....	27
<b>Chapter 3: Tutorial: Working with System Optimizations .....</b>	<b>29</b>
Introduction to System Ports and DMA.....	29
Learning Objectives.....	30
Creating a New Project.....	31
Marking Functions for Hardware Implementation.....	35
Specifying System Ports .....	37
Error Reporting .....	39
Additional Exercises .....	39
<b>Chapter 4: Tutorial: Debugging Your System .....</b>	<b>44</b>
Learning Objectives.....	44
Setting Up the Board.....	44
Creating a Standalone Project .....	45
Setting up the Debug Configuration .....	46
Running the Application.....	47
Additional Exercises .....	48
<b>Chapter 5: Tutorial: Estimating System Performance .....</b>	<b>52</b>
Learning Objectives.....	52

Setting Up the Board .....	52
Setting up the Project for Performance Estimation .....	53
Comparing Software and Hardware Performance .....	55
Changing Scope of Overall Speedup Comparison.....	57
Additional Exercises .....	58
 <b>Chapter 6: Tutorial: Task Pipelining Optimizations .....</b>	<b>61</b>
Task Pipelining .....	61
Learning Objectives.....	62
Task Pipelining in the Matrix Multiply Example .....	62
 <b>Chapter 7: Tutorial: Hardware/Software Event Tracing .....</b>	<b>64</b>
Tracing a Standalone or Bare-Metal Project .....	64
Tracing a Linux Project .....	73
Viewing Traces .....	80
 <b>Chapter 8: Tutorial: Communication to Platform I/O Streams.....</b>	<b>81</b>
Advanced Concepts: AXI4-Stream to Memory .....	81
Advanced Concepts: AXI4-Stream to Accelerator .....	83
Advanced Concepts: Lossless Data Capture.....	86
 <b>Appendix A: Troubleshooting.....</b>	<b>90</b>
Path Names Too Long.....	90
Use Correct Tool Scripts.....	90
 <b>Appendix B: Additional Resources and Legal Notices .....</b>	<b>91</b>
Xilinx Resources .....	91
Solution Centers .....	91
References.....	91
Please Read: Important Legal Notices.....	92

## Introduction

The SDSoC™ (Software-Defined System On Chip) environment is an Eclipse-based Integrated Development Environment (IDE) for implementing heterogeneous embedded systems using the Zynq®-7000 All Programmable SoC and Zynq UltraScale+™ MPSoC platforms. The SDSoC environment provides an embedded C/C++ application development experience with an easy to use Eclipse IDE, and comprehensive design tools for heterogeneous Zynq SoC development to software engineers and system architects. The SDSoC environment includes a full-system optimizing C/C++ compiler that provides automated software acceleration in programmable logic combined with automated system connectivity generation. The application programming model within the SDSoC environment should be intuitive to software engineers. An application is written as C/C++ code, with the programmer identifying a target platform and a subset of the functions within the application to be compiled into hardware. The SDSoC system compiler then compiles the application into hardware and software to realize the complete embedded system implemented on a Zynq device, including a complete boot image with firmware, operating system, and application executable.

The SDSoC environment abstracts hardware through increasing layers of software abstraction that includes cross-compilation and linking of C/C++ functions into programmable logic fabric as well as the ARM CPUs within a Zynq device. Based on a user specification of program functions to run in programmable hardware, the SDSoC environment performs program analysis, task scheduling and binding onto programmable logic and embedded CPUs, as well as hardware and software code generation that automatically orchestrates communication and cooperation among hardware and software components.

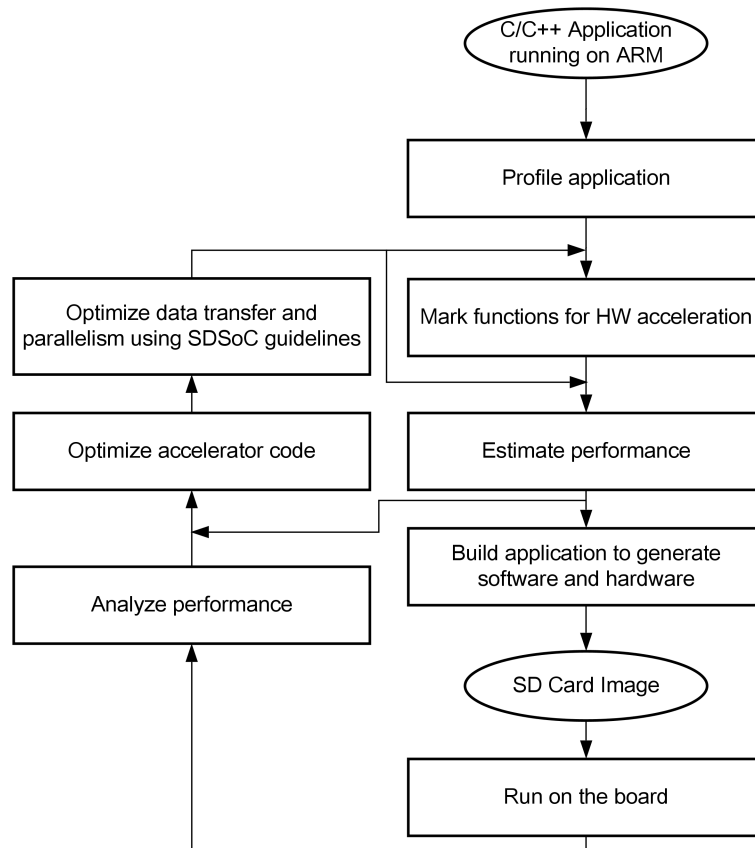
The SDSoC environment 2016.2 release includes support for the ZC702, ZC706, MicroZed, ZedBoard and Zybo development boards featuring the Zynq-7000 AP SoC, and for the ZCU102 development board featuring the Zynq UltraScale+ MPSoC. Additional platforms are available from partners and for more information, visit the [SDSoC environment web page](#).

---

## User Design Flow

The first step is to identify compute-intensive hot spots in the application that can be migrated to programmable logic to achieve higher performance, and to isolate them into functions that you can compile for hardware. C/C++ code compiled for programmable logic with the SDSoC environment must conform to coding guidelines described in [SDSoC Environment User Guide \(UG1027\), Coding Guidelines](#), and must also conform to Vivado® High-Level Synthesis (HLS) guidelines. For example, the code cannot invoke recursive functions, dynamically allocate memory, or make unrestricted use of pointers. See the [SDSoC Environment User Guide \(UG1027\), A Programmer's Guide to Vivado High-Level Synthesis](#) for more information. RTL IP needs to be wrapped into a C-callable library, as described in [SDSoC Environment User Guide: Platforms and Libraries \(UG1146\), Creating a Library](#).

**Figure 1–1: SDSoC Environment Flow**



X14740-070215

This document introduces the SDSoC environment development platform using its Eclipse based GUI. This integrated development environment provides interactive features to simplify the development process, project management, and build automation. However, most of these operations can also be scripted using makefiles.

## System Requirements

- Host running one of the following operating systems:
  - Linux – Red Hat Enterprise Workstation 6.6-6.7 and 7.0-7.1 (64-bit) and Ubuntu Linux 14.04.3 LTS (64-bit)
  - Windows – Windows 7.1 Professional (64-bit), English and Windows 10 Professional (64-bit), English
- Installation of the Xilinx SDSoC™ environment, which includes:
  - SDSoC environment 2016.2, including an Eclipse/CDT-based GUI, high-level system compiler, and ARM GNU toolchain
  - Vivado® Design Suite System Edition 2016.2, with Vivado High-Level Synthesis (HLS) and the Xilinx Software Development Kit (SDK)
- The SDSoC environment includes the same GNU ARM toolchain included with the Xilinx® Software Development Kit (SDK) 2016.2, which also provides additional tools used by the SDSoC environment. The SDSoC environment setup script sets PATH variables to use this toolchain.
  - The provided toolchains contain 32-bit executables, requiring your Linux host OS installation to include 32-bit compatibility libraries.
  - RHEL 6 and 7 64-bit x86 Linux installations might not include the 32-bit compatibility libraries, and might need to be added separately; see <https://access.redhat.com/site/solutions/36238>.
  - On RHEL, 32-bit compatibility libraries can be installed by becoming a superuser (or root) with root access privileges and running the `yum install glibc.i686` command.
  - On Ubuntu, 32-bit compatibility libraries can be installed by becoming a superuser (or root) with root access privileges and running the commands (refer to the SDSoC release notes for additional information):
 

```
sudo dpkg --add-architecture i386
sudo apt-get update
sudo apt-get install libc6:i386 libncurses5:i386 libstdc++6:i386
sudo apt-get install libgtk2.0-0:i386 dpkg-dev:i386
sudo ln -s /usr/bin/make /usr/bin/gmake
```
  - The version of the toolchain can be displayed by running the `arm-linux-gnueabi-g++ -v` command.
  - The last line of the output printed in the shell window should be `gcc version 4.9.2 20140904 (prerelease) (crosstool-NG linaro-1.13.1-4.9-2014.09 - Linaro GCC 4.9-2014.09)`.
- A mini-USB cable to observe the UART output from the board

## Obtaining and Managing a License

The SDSoC environment 2016.2 release uses the Xilinx FLEXnet license configuration manager. Ask your Xilinx technical contact for information on how to obtain a license key.

Install your license key using the appropriate method for node-locked or floating license servers. Node-locked licenses are typically copied to <home>/Xilinx (Linux) or C:\Xilinx (Windows). For existing floating license installations, you typically add the new license file contents to the existing license file and restart the server. For new floating license installations, run the FLEXnet utility, lmgrd, for example:

```
lmgrd -c <path_to_license>/Xilinx.lic -l <path_to_license>/log1.log
```

Client machines for node-locked licenses look for the license in one or more fixed locations. For the floating license, add the path to the license file or license server in the port@server format in the XILINXD\_LICENSE\_FILE environment variable.

**NOTE:** If you use Windows Explorer to create the folder C:\Xilinx, navigate to C:\ and when you click on **New Folder**, enter the folder name with a trailing dot .Xilinx. (dot Xilinx dot). Press **Enter** and Windows creates the folder name .Xilinx (dot Xilinx); the trailing dot tells Windows to allow dot as the first character of the folder name.



**TIP:** The SDSoC environment 2016.2 release licenses are administered in the same manner as other Xilinx products. Your local Xilinx license administrator can help install the SDSoC environment license key file.

## Downloading

To download the SDSoC™ environment, go to the [SDSoC environment web page](#).

## Installing

Download the installer files, execute them and follow the on-screen instructions. The instructions below illustrate a typical installation session.

1. Execute the xsetup.exe (on Windows) or xsetup (on Linux) installer files.  
The SDSoC Installer - Welcome page appears.
2. Click **Next**.  
The SDSoC Installer - Accept License Agreements page appears.
3. Check all the **I Agree** check boxes on the page to accept Xilinx and other third-party license terms and conditions.
4. Click **Next**.  
The SDSoC Environment Installer page appears.
5. Customize your installation by selecting options on the SDSoC Installer page.

**NOTE:** If you have installed Vivado Design Suite 2016.2 previously, you still need to install the SDSoC environment version of the Vivado tools, but you do not need to reinstall the Cable Drivers.

6. Click **Next**.  
The SDSoC Installer - Select Destination Directory page appears.



7. Select installation options such as location and shortcuts.
8. Click **Next**.  
The SDSoC Installer - Installation Summary page appears.
9. Click **Install** to begin the installation.

After the installation completes, you have a directory with the following structure:

```
<path_to_install>/SDSoC/2016.2
aarch32-linux
aarch32-none
aarch64-linux
aarch64-none
bin
data
docs
lib
llvm-clang
platforms
samples
scripts
SDK
tps
Vivado
Vivado_HLS
settings64.[csh|sh|bat]
```

The installed software includes a copy of the SDSoC environment 2016.2, Vivado Design Suite 2016.2, Vivado HLS 2016.2, Xilinx SDK 2016.2 and scripts to set the environment.

## Linux Environment Setup Script

To run the SDSoC™ environment, use the environment setup script (`settings64.csh` or `settings64.sh`) created by the installer. This script in turn runs setup scripts in the installation directory of each of the underlying tools to update the PATH environment.

To confirm that the environment is set up properly, type the commands below and confirm that the commands find the installation locations consistent with the tool setup script:

```
% source settings64.csh
% which sdsc # SDSoC C/C++ build tool version
% which vivado # Vivado design tool version
% which vivado_hls # Vivado High-Level Synthesis (HLS) tools
% which bootgen # Boot image creation tool (2016.2 version)
```

If the paths returned by the Linux `which` command are not consistent with the path to the installation directory, or the command was not found, confirm that the correct setup script was run.



**CAUTION!** *In each shell used to run the SDSoC environment, use only the environment setup scripts corresponding to the Xilinx tool releases or PATH environment setting listed above. Running Xilinx design tool environment setup scripts from other or additional releases in the same shell might result in incorrect behavior or results with the SDSoC environment.*

## Windows Environment Setup Script

To run the SDSoC environment, open the tool from the Windows Start Menu by clicking **Xilinx Design Tools > SDSoC 2016.2 > SDSoC 2016.2** or by double-clicking a desktop shortcut created by the installer.

To confirm that the environment is set up properly, invoke the SDSoC Terminal by double-clicking the **SDSoC 2016.2 Terminal** shortcut created by the installer.

Type the following commands, and confirm that the installation location is consistent with the SDSoC environment 2016.2 installation. (Do not enter the comments beginning with REM.)

```
> REM SDSoC C/C++ build tool
> where sdsc
> REM Vivado design tool
> where vivado
> REM Vivado High-Level Synthesis (HLS) tools
> where vivado_hls
> REM Boot image creation tool (2016.2 version)
> where bootgen
```

If the paths returned by the `where` command are not consistent with the path to the installation directory, or the command was not found, confirm that you ran the commands in an **SDSoC 2016.2 Terminal**.



**CAUTION!** *If Cygwin is included in a global PATH environment variable and issues are encountered, it may need to be temporarily removed when running SDSoC™ development environment flows.*

For example, in a command shell, type:

```
set PATH=%PATH:c:\cygwin\bin;=%
```

## Validating Installation

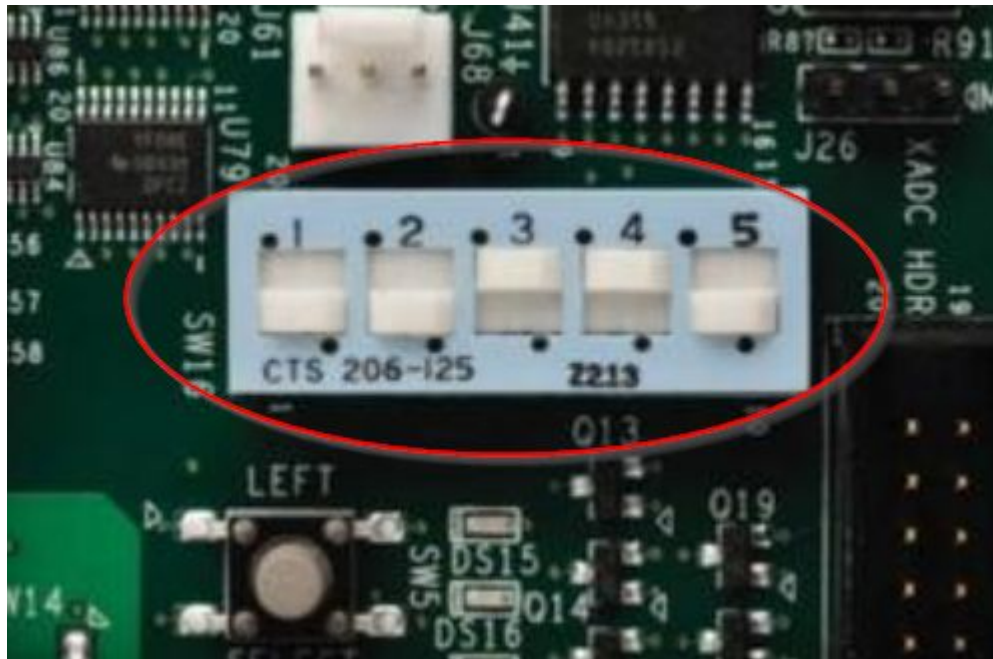
A basic use flow for validating the SDSoC™ environment installation includes the steps outlined below. Specific instructions are given for the ZC702 board, but if you have a different board, use this information as a reference.

1. Testing the board setup, connections and terminal setup using a matrix multiplication pre-built design that can be copied to an SD card. Insert the SD card into your board, power on the board, run the ELF and observe the output of the matrix multiplication on a terminal connected to the board via a USB UART connection.
2. Using the SDSoC environment to create a simple matrix multiplication application example, targeting the matrix multiplication function for conversion to a hardware accelerator block that resides on the programmable logic. This step validates the tool installation and environment setup.
3. Running the example on the board. The SDSoC environment produces an SD card image containing a Linux bootloader, Linux kernel with the required drivers to communicate with the hardware accelerator, file system, and the application ELF. Use the SD card to run the ELF and observe the output.

## Configuring the Board for SD Card Boot

To boot the board from an SD card, you need to physically change either a switch or jumper settings on the board. This section describes settings for a ZC702 board. If you have a different board, consult your board reference guide for the appropriate switch or jumper settings.

1. Identify whether you have to change a jumper or a switch.  
This depends on the version of the ZC702 board.



2. To set jumpers or switches to boot from the SD card:
  - For Revision C and earlier boards:
    - J22 1-2 (connects pins 1 and 2)
    - J25 1-2 (connects pins 1 and 2)
  - For Revision D and newer boards:
    - DIP switch SW16 (light blue/grey color) positions 3 and 4 should be set to 1.

The SD boot settings for the ZC702 Evaluation Board are also available in the [Zynq-7000 XC7Z020 All Programmable SoC User Guide \(UG850\)](#) and the wiki page, [Boot Pre-Built Xilinx ZC702 Image](#).

## Connecting the Board to a Serial Terminal

To connect a ZC702 board to a serial terminal you need a mini USB cable to connect the UART port on the board to the computer where you run a serial terminal. There is a serial terminal available as part of the SDSoc IDE (tab labeled Terminal 1 at bottom of screen).

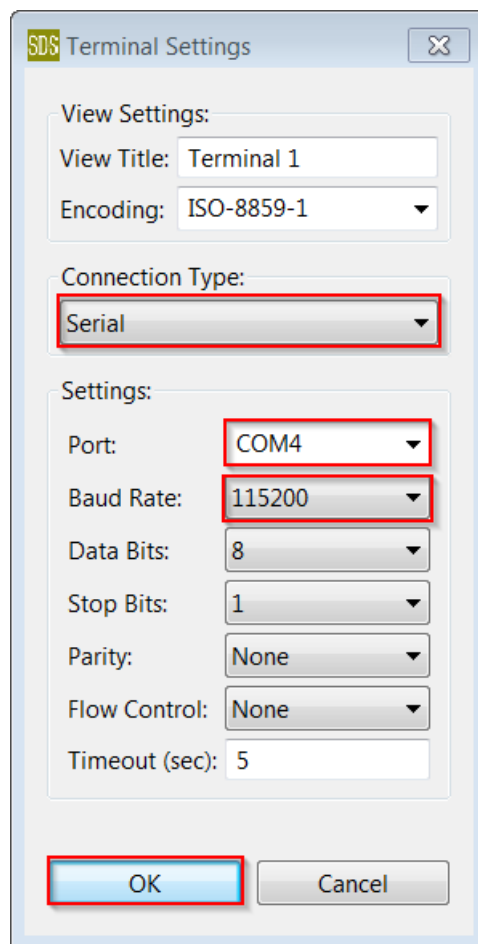
If you have a different board, consult your board reference guide for the proper cable type and connector to use, as well as USB UART driver installation. Serial terminal setup steps are similar.

1. Connect the mini USB cable to the UART port.



2. Set up the serial terminal (for example, PuTTY, minicom, or the SDSoC environment SDK Terminal or Terminal views - when running the SDSoC environment on Windows hosts, PuTTY or the SDSoC environment SDK Terminal or Terminal views are recommended, and on Linux hosts, minicom or the SDSoC environment SDK Terminal view are recommended):
  - Set the baud rate to 115200 baud.
  - In Windows, set the serial port to COMn, where n is a number and can be found as follows:
    - Click **Start**, right-click on **Computer**, then click **Properties**.
    - Select **Device Manager** and open Ports (COM & LPT).
    - Use the COM port labeled Silicon Labs CP210x USB to UART Bridge.

**NOTE:** If the right COM port does not appear on the Terminal Settings window, make sure the board is connected to the USB port and turned on. Restart the SDSoC environment by selecting **File > Restart** and the COM port should appear on the list. The dialog below is from the SDSoC Terminal view invoked by **Window > Show > View > Other** and selecting **Terminal > Terminal**.



### 3. Power on the board.

The board needs to be powered on at least once with the mini USB cable connected for Windows to recognize the UART and install the driver. You might need to power cycle the board. If the driver does not load you can download it from <http://www.silabs.com/products/mcu/pages/usbtouartbridgevcpcdrivers.aspx> and install it manually.

## Executing a Pre-built Application

The installation directory for the SDSoC™ environment contains several examples in the `<path_to_install>/SDSoC/2016.2/samples` folder.

The `mmult_pipeline` example contains C++ source files containing the main application that calls a matrix multiplication function and displays the output on stdout using `printf()` statements, a makefile to build the application, and the `sd_card_prebuilt` folder containing pre-built files. The files in the `sd_card_prebuilt` folder are used to validate your board, board connections, and terminal setup before using the SDSoC environment.

The `mmult_pipeline` example is found in the folder `<path_to_install>/SDSoC/2016.2/samples/mmult_pipeline`, which has the directory structure below:

```
<path_to_install>/SDSoC/2016.2/samples/mmult_pipeline
Makefile
mmult.cpp
mmult_accel.cpp
mmult_accel.h
sd_card_prebuilt
  BOOT.BIN
  README.txt      boot.bif
  devicetree.dtb  mmult.elf
  uImage
  uramdisk.image.gz
```

The `mmult_pipeline` example includes a pre-built application for the ZC702 board. If you have a different board, look in `<path_to_install>/SDSoC/2016.2/samples` to locate an application that runs on your board and build the application. Each example includes a README file describing how to run the application. If using a partner board or platform, a pre-built SD card application may be available. If a pre-built application cannot be located, skip this step.

To run the pre-built application on the ZC702 board, follow these steps:

1. Copy the contents of the `sd_card_prebuilt` folder to the root folder of an SD card. The SD card must be formatted using FAT32 (not NTFS).
2. Insert the card into the SD card slot of the ZC702 board.
3. Confirm jumpers or switches are set to boot from the SD card. See [Configuring the Board for SD Card Boot](#).
4. Connect an Ethernet cable from the board to your network. This is optional. It allows you to connect to the network.
5. Set up a serial terminal. See [Connecting the Board to a Serial Terminal](#).
6. With the SD card inserted and cables connected, power up the board and start the serial terminal session.

You should see the Done LED turn green and Linux booting.



7. At the prompt, type `cd /mnt`

This takes you to the SD card folder containing the application ELF file.

8. To run the application ELF, type: `./mmult.elf`
9. The application displays information about the run and the results of the matrix multiplication.

You see output similar to that shown below:

```
Testing mmult ...
Average number of processor cycles for golden version: 182299
Average number of processor cycles for hardware version: 18685
TEST PASSED
```

If you are able to run the pre-built application, continue on to [Building and Executing an Example Application](#).



### IMPORTANT:

*Do not proceed to [Building and Executing an Example Application](#) if a pre-built application exists and you are not able to run it. If a pre-built application does not exist for your board, follow the instructions in [Building and Executing an Example Application](#) to build an application and validate your board setup.*

*If you power up the board and the Done LED does not turn green, this indicates the bootloader is not configuring the programmable logic. Confirm that the pre-built SD card files were copied to the root (top) location of the SD card and not into a folder, and that the file sizes match. Confirm jumper settings. Use the SD card to boot another board to determine if the first one is not working properly. Confirm the SD card was formatted using FAT32 (not NTFS).*

*If you do not see Linux booting on the terminal, check the baud rate and COM port settings. Confirm the USB UART drivers are installed (uninstall and reinstall if unsure).*

*The command `sdscc -sds-pf-list` can be used to list platforms included with SDSoC.*

## Building and Executing an Example Application

You can now build the sample design

`<path_to_installation>/SDSoC/2016.2/samples/mmult_pipelined`, and in doing so validate your tool installation and environment setup.

The `mmult_pipelined` example was created to run on the ZC702 board. The instructions below are for that board.

If you have a different board, you might still be able to use this design by changing the makefile to specify platform option for your board, for example a ZC706 (`-sds-pf zc706`), instead of the ZC702 (`-sds-pf zc702`). Alternatively, check the file `<path_to_install>/SDSoC/2016.2/samples/README.txt` to locate an application that runs on your board and use that application, or for partner boards and platforms, use a partner-provided example.

The `mmult_pipelined` folder contains C++ source files containing the main application that calls a matrix multiplication function and displays the output on stdout using `printf()` statements.

A user makefile invokes the SDSoC environment to produce the hardware system including hardware accelerators, along with software libraries and API to utilize the hardware. The top-level makefile contains targets to build object files (.o) from application source files. Link them to create the application ELF file and produce an SD card image.

The following sections provide details on how to build the application on supported hosts.

- [Building an Application on a Linux Host](#)
- [Building an Application on a Windows Host](#)

If the `make` command completes successfully, run the application on the ZC702 board. For details, see [Executing an Example Application](#).

### ***Building an Application on a Linux Host***

To build the example application on a Linux host:

1. Set up your environment to run the SDSoC environment by running the setup script created by the installer:

- for a C-Shell

```
% source <path_to_install>/SDSoC/2016.2/settings64.csh
```

- for a Bourne Shell or Bash

```
% . <path_to_install>/SDSoC/2016.2/settings64.sh
```

2. Copy the folder named `mmult_pipeline` to a working directory where you have write permission:

```
% cp -r <path_to_install>/SDSoC/2016.2/samples/mmult_pipeline .
% cd mmult_pipeline
```

3. Build the application and the SD card image:

```
% which sdsc # displays path to the sdsc tool
% make
```

The build takes some time. After completion, a folder named `sd_card` contains the ELF file and boot image required to start Linux on the ZC702 board and run the ELF application.

### ***Building an Application on a Windows Host***

To build the example application on a Windows host:

1. Run the **SDSoC 2016.2 Terminal** shortcut.

This sets up your environment using commands described in [Building an Application on a Linux Host](#).



2. Type the commands below at the Windows command shell prompt '>' (you need not enter the comments beginning with REM at the prompt):

```
> cp -r <path_to_sdsoc_install>\SDSoC\2016.2\samples\mmult_pipelined
<path_to_user_folder>\mmult_pipelined
> cd <path_to_user_folder>\mmult_pipelined
> REM displays path to the sdscc tool
> where sdscc
> make
```

**NOTE:** CP is a Linux command, which is a part of your environment inherited from the SDSoC environment.

A subset of Linux shell commands are available if you want to use them; otherwise you can use Windows commands, such as:

```
> xcopy <path_to_install>\SDSoC\2016.2\samples\mmult_pipelined
<path_to_user_folder>\mmult_pipelined /s /e /h
> cd <path_to_user_folder>\mmult_pipelined
> make
```

## Executing an Example Application

After generating the application, run it on the ZC702 board. For details, see [Executing a Pre-built Application](#). In the summary below, steps that were performed earlier (jumper and switch settings, Ethernet cable connection, and serial terminal setup) are not described again and have been omitted.

1. Copy the contents of the sd\_card folder to an SD card.
2. Insert the card into the SD card slot of the ZC702 board, and confirm jumpers or switches are set to boot from the SD card. See [Configuring the Board for SD Card Boot](#).
3. Connect the mini USB cable from the board to the computer.
4. With the SD card inserted and cables connected, power up the board and start the serial terminal session. See [Connecting the Board to a Serial Terminal](#).

You should see Linux booting.

5. At the prompt, type `cd /mnt`.

This takes you to the SD card folder containing the application ELF file.

6. To run the application ELF, type: `./mmult.elf`
7. The application displays information about the run and the results of the matrix multiplication.

The output is similar to that shown below:

```
Testing mmult ...
Average number of processor cycles for golden version: 182299
Average number of processor cycles for hardware version: 18685
TEST PASSED
```

If the application runs properly on the board, try running and modify additional designs in the samples folder. [SDSoC Environment User Guide \(UG1027\)](#) demonstrates techniques for increasing performance across a range of implementation examples.



**IMPORTANT:** If you are unable to run the application and have confirmed the board has been set up properly, contact [Xilinx Support](#).

# Tutorial: Creating, Building and Running a Project

This tutorial demonstrates how you can use the SDSoC environment to create a new project using available templates, mark a function for hardware implementation, build a hardware implemented design, and run the project on a ZC702 board.

**NOTE:** This tutorial is separated into steps, followed by general instructions and supplementary detailed steps allowing you to make choices based on your skill level as you progress through it. If you need help completing a general instruction, go to the detailed steps, or if you are ready, simply skip the step-by-step directions and move on to the next general instruction.

**NOTE:** You can complete this tutorial if you do not have a ZC702 board. When creating the SDSoC environment project, select your board and one of the available applications if the suggested template **Matrix Multiplication and Addition** is not found. For example, boards such as the MicroZed with smaller Zynq-7000 devices offer the **Matrix Multiplication and Addition (area reduced)** application as an available template. Any application can be used to learn the objectives of this tutorial. A pre-built project is only available for the ZC702 board.

---

## Learning Objectives

After you complete the tutorial (lab1), you should be able to:

- Create a new SDSoC environment project for your application from a number of available platforms and project templates.
- Mark a function for hardware implementation.
- Build your project to generate a bitstream containing the hardware implemented function and an executable file that invokes this hardware implemented function.

[Questions and Additional Exercises](#)

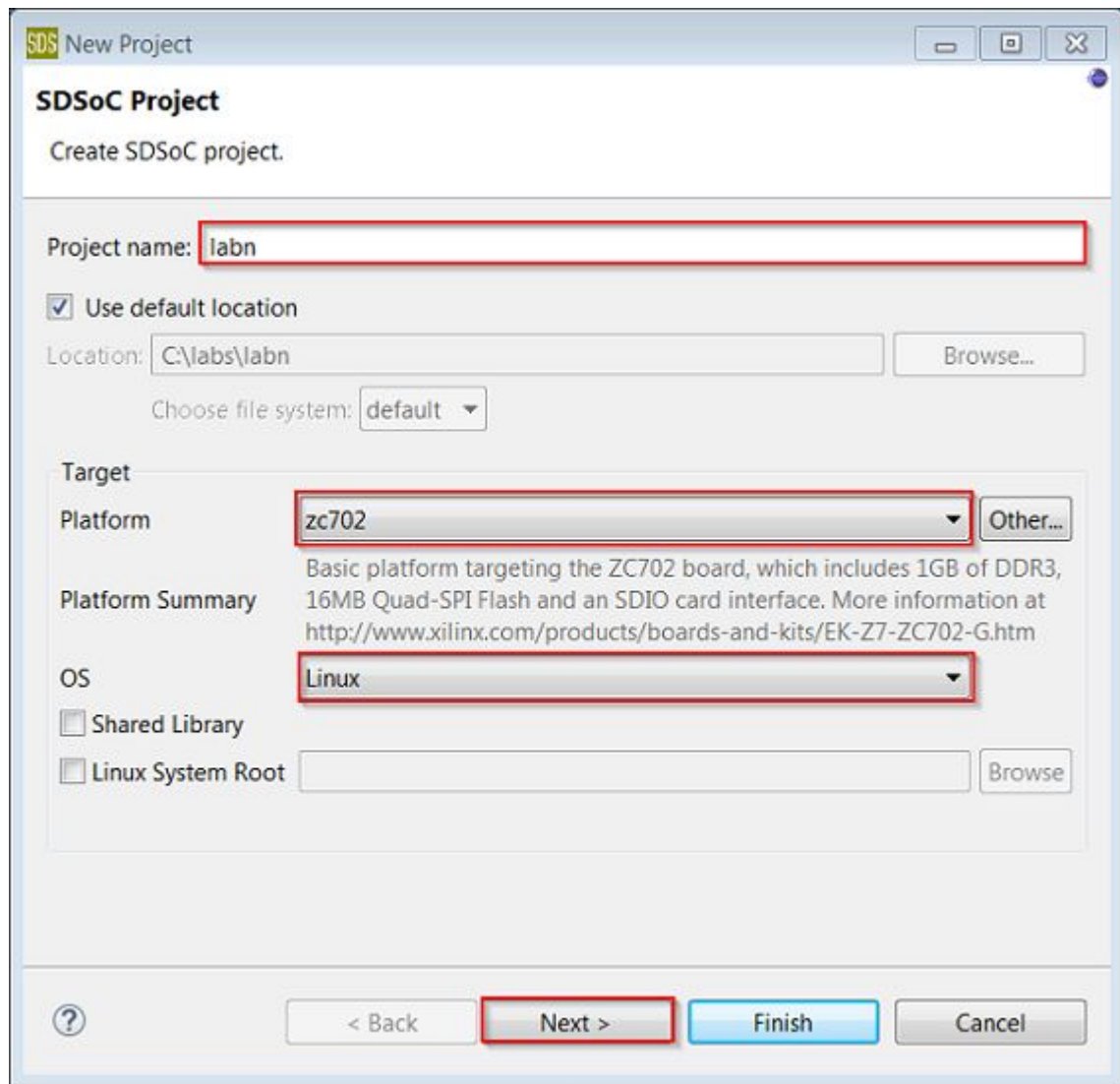
---

## Creating a New Project

To create a project in the SDSoC™ environment that performs matrix multiplication and addition:

1. Launch the SDSoC environment using the desktop icon or the **Start** menu.
2. When you launch the SDSoC environment, the **Workspace Launcher** dialog appears. Click **Browse** to enter a workspace folder used to store your projects (you can use workspace folders to organize your work), then click **OK** to dismiss the **Workspace Launcher** dialog.

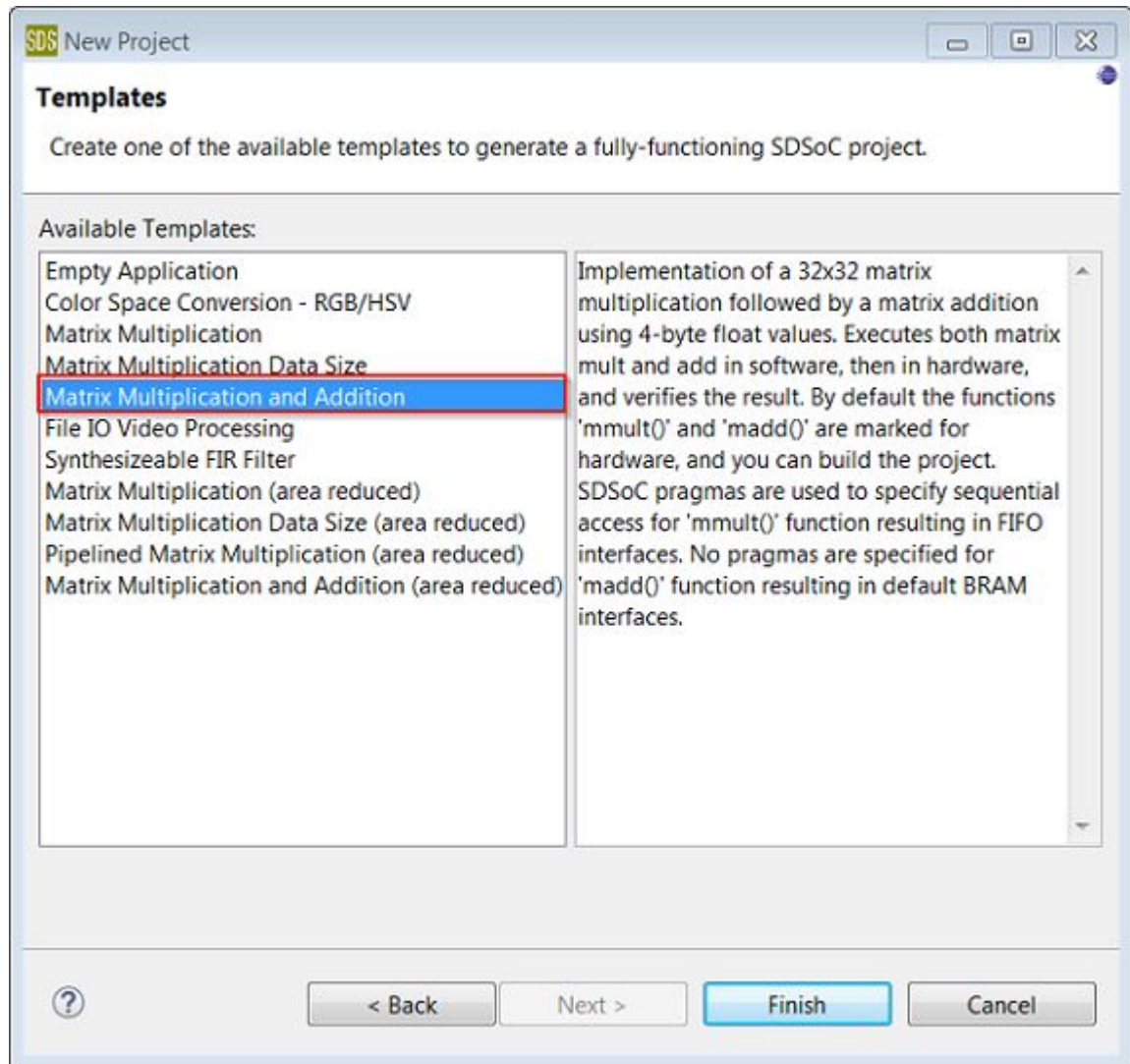
3. The SDSoC environment window opens with the **Welcome** tab visible when you create a new workspace. The tab includes links for quickly getting started, for example **Create SDSoC Project**, and for accessing documentation and tutorials, for example **SDSoC User Guide**. The **Welcome** tab can be dismissed by clicking the X icon or minimized if you do not wish to use it.
4. In the **Welcome** tab click **Create SDSoC Project** or in the SDSoC menu bar select **File > New > SDSoC Project**.



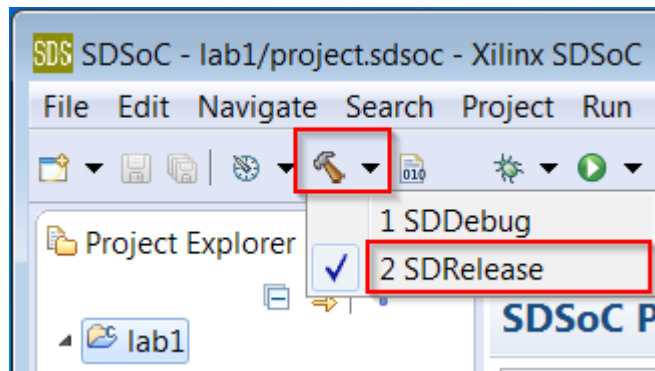
5. Specify the name of the project. The figure shows `labn` as the **Project name**, but the tutorial steps use `lab1` for the first tutorial, `lab2` for the second tutorial, etc.
6. From the **Platform** drop-down list of available platforms, select **zc702**.
7. From the **OS** drop-down list for the selected platform, select **Linux**.
8. Click **Next**.

The Templates page appears, containing source code examples for the selected platform.

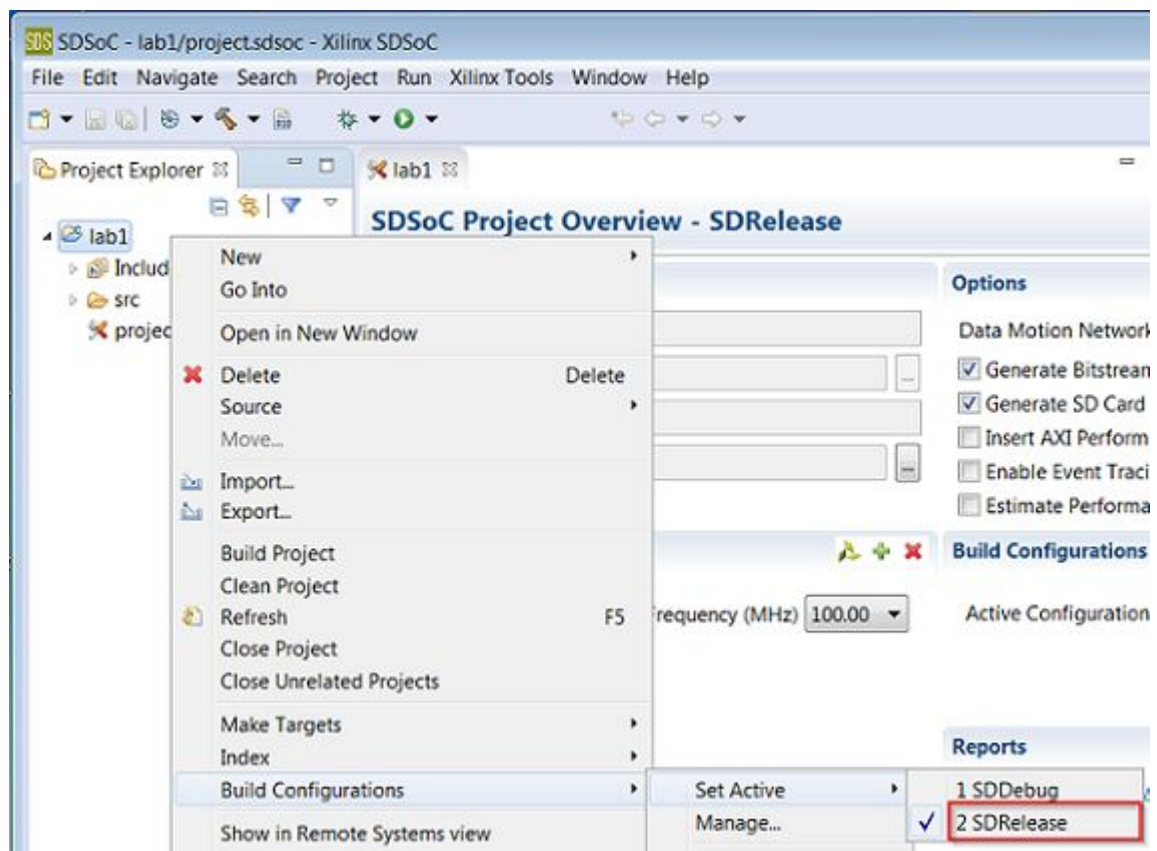
9. From the list of application templates, select **Matrix Multiplication and Addition** and click **Finish**.



10. The standard build configurations are SDDebug and SDRelease, and you can create additional build configurations. To get the best runtime performance, switch to use the SDRelease configuration by clicking on the project and selecting **SDRelease** from the **Build** icon, or by right-clicking the project and selecting **Build Configurations > Set Active > SDRelease**. The SDRelease build configuration uses a higher compiler optimization setting than the SDDebug build configuration. The SDSoc Project Overview window also allows you to select the active configuration or create a build configuration. The **Build** icon provides a drop-down menu for selecting the build configuration and building the project. Clicking on the **Build** icon builds the project.

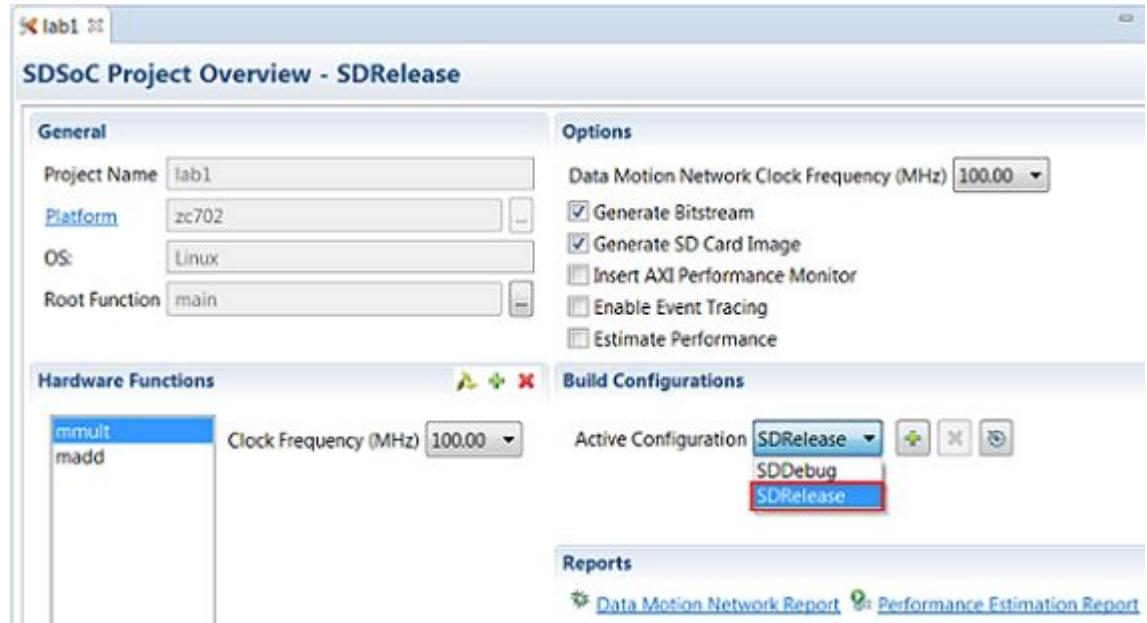


In the Project Explorer you can right-click on the project to select the build configuration.



The SDSoc Project Overview window includes a Build Configurations panel where you can select the active configuration or create a build configuration.






The SDSoC Project Overview window provides a summary of the project settings.

When you build an SDSoC application, you use a build configuration (a collection of tool settings, folders and files). Each build configuration has a different purpose. SDDebug builds the application with extra information in the ELF (compiled and linked program) that you need to run the debugger. The debug information in an ELF increases the size of the file and makes your application information visible. The SDRelease option provides the same ELF file as the SDDebug option with the exception that it has no debug information. The Estimate Performance option can be selected in any build configuration and is used to run the SDSoC environment in a mode used to estimate the performance of the application (how fast it runs), which requires different settings and steps (see [Tutorial: Estimating System Performance](#)).

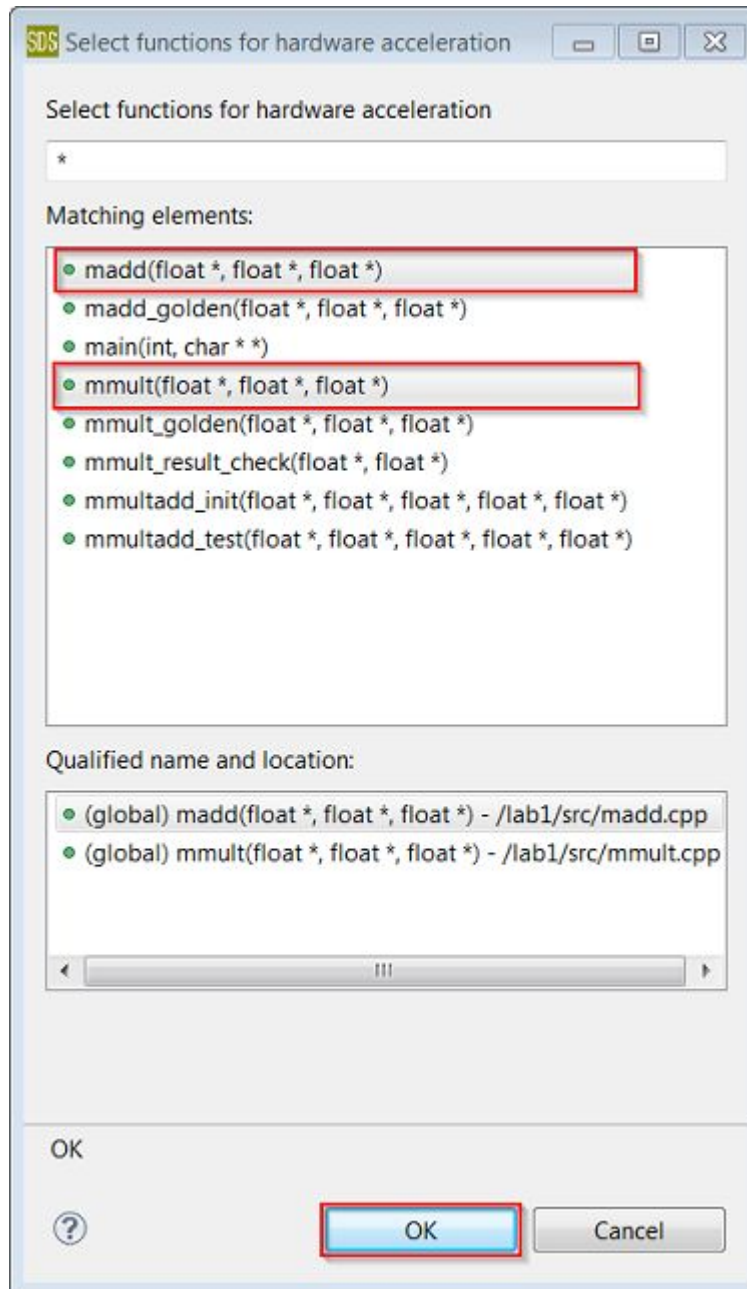
## Marking Functions for Hardware Implementation

This application has two hardware functions. One hardware function, `mmult`, multiplies two matrices to produce a matrix product, and the second hardware function, `madd`, adds two matrices to produce a matrix sum. These hardware functions are combined to compute a matrix multiply-add function. Both hardware functions `mmult` and `madd` are specified to be implemented in hardware.

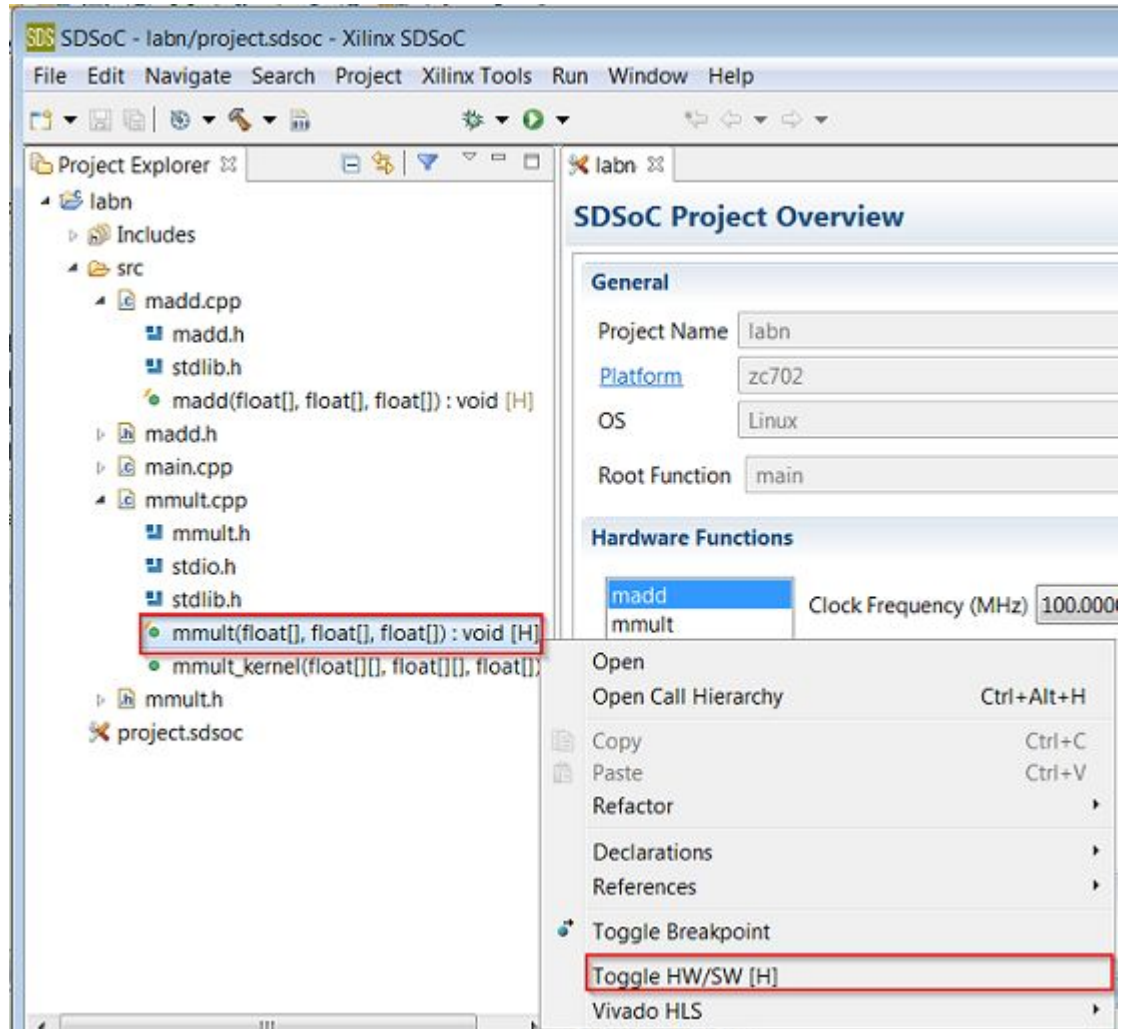
When the SDSoC environment creates the project from a template, it specifies the hardware functions for you. In cases where hardware functions have been removed or have not been specified, follow the steps below to add hardware functions (for this lab, you do not need to mark functions for hardware – the SDSoC environment has already marked them).

1. The SDSoC Project Overview window provides a central location for setting project values. Click on the tab labeled **<name of project>** (if the tab is not visible, double-click on the **project.sdsoc** file in the Project Explorer tab) and in the **Hardware Functions** panel, click on the Add Hardware Function icon  to invoke a dialog to specify hardware functions.

2. Ctrl-click (press the Ctrl key and left click) on the `mmult` and `madd` functions to select them in the "Matching elements" list. Click OK, and observe that both functions have been added to the Hardware Functions list.



Alternatively, you can expand `mmult.cpp` and `madd.cpp` in the Project Explorer, right click on `mmult` and `madd` functions, and select **Toggle HW/SW** (when the function is already marked for hardware, you will see **Toggle HW/SW [H]**). When you have a source file open in the editor, you can also select hardware functions in the Outline window.



**CAUTION!** Not all functions can be implemented in hardware. See the [SDSoC Environment User Guide \(UG1027\)](#), [Coding Guidelines](#) for more information.

## Building a Design with Hardware Accelerators

To build a project and generate an executable, bitstream, and SD Card boot image:

1. Right-click **lab1** in the **Project Explorer** and select **Build Project** from the context menu that appears.

The SDSoC™ system compiler stdout is directed to the Console tab. The functions selected for hardware are compiled using Vivado® HLS into IP blocks and integrated into a generated Vivado tools hardware system based on the selected base platform. The system compiler then invokes Vivado synthesis, place and route tools to build a bitstream, and invokes the ARM GNU compiler and linker to generate an application ELF executable file.

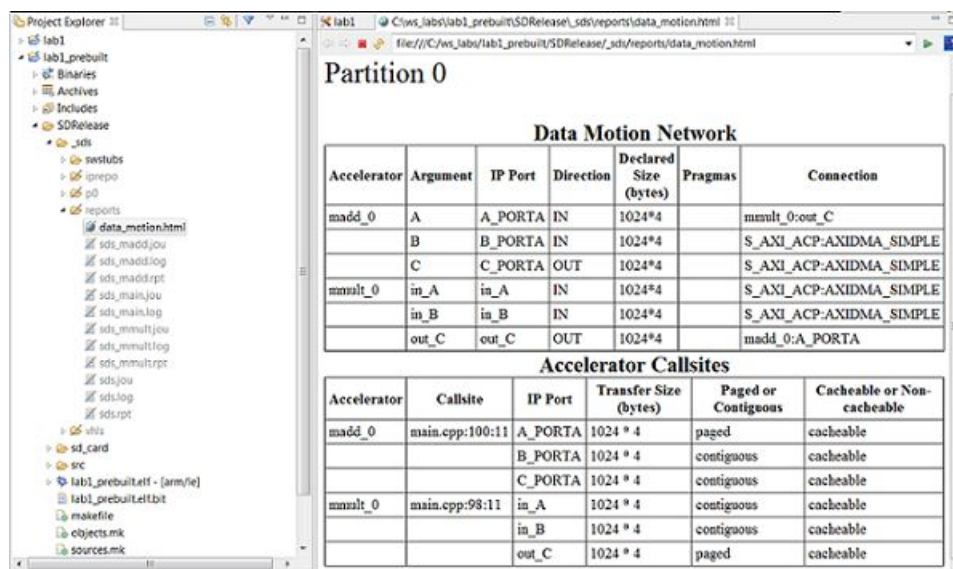


- Because the Vivado synthesis process takes some time, instead of building the project you have the option to import the pre-built files into your workspace with these steps:

**NOTE:** To minimize disk usage in the SDSoC installation, the imported project might contain fewer files than a project you build, but it includes the files required to complete the tutorial.

- Select **File > Import** and then select **General > Existing Projects into Workspace** and click **Next**.
  - Select **Select archive file** and click **Browse** to navigate to <path to install>/SDSoC/2016.2/docs/labs/lab1\_prebuilt.zip.
  - Select **lab1\_prebuilt.zip**, and click **Open**.
  - Click **Finish**.
- In the **SDSoC Project Overview** window, under the **Reports** pane, click on **Data motion** to view the Data Motion Network report.

This report shows the connections done by the SDSoC environment and the types of data transfers for each function implemented in hardware. For details, see the [Tutorial: Working with System Optimizations](#).



**Partition 0**

**Data Motion Network**

Accelerator	Argument	IP Port	Direction	Declared Size (bytes)	Pragmas	Connection
madd_0	A	A_PORTA	IN	1024*4		mmult_0:out_C
	B	B_PORTA	IN	1024*4		S_AXI_ACP-AXIDMA_SIMPLE
	C	C_PORTA	OUT	1024*4		S_AXI_ACP-AXIDMA_SIMPLE
mmult_0	in_A	in_A	IN	1024*4		S_AXI_ACP-AXIDMA_SIMPLE
	in_B	in_B	IN	1024*4		S_AXI_ACP-AXIDMA_SIMPLE
	out_C	out_C	OUT	1024*4		madd_0:A_PORTA

**Accelerator Callsites**

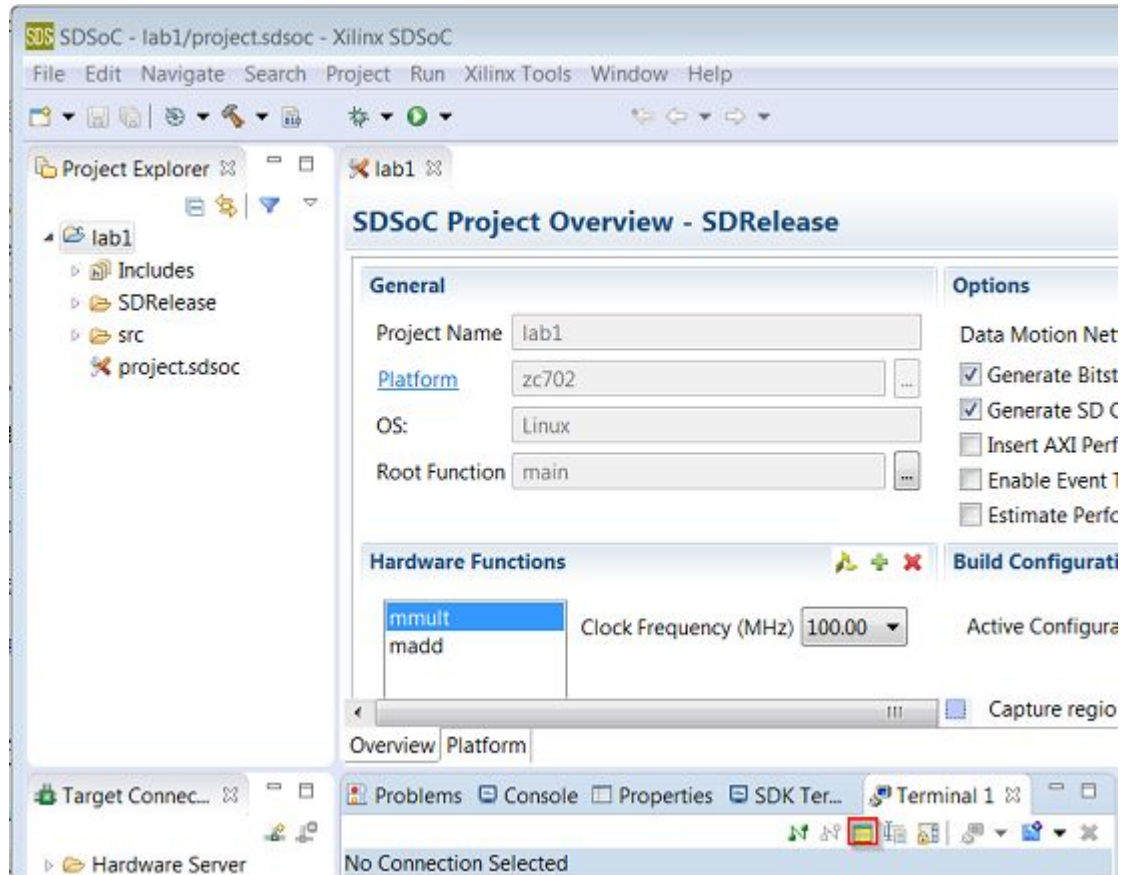
Accelerator	Callsite	IP Port	Transfer Size (bytes)	Paged or Contiguous	Cacheable or Non-cacheable
madd_0	main.cpp:100:11	A_PORTA	1024 * 4	paged	cacheable
		B_PORTA	1024 * 4	contiguous	cacheable
		C_PORTA	1024 * 4	contiguous	cacheable
mmult_0	main.cpp:98:11	in_A	1024 * 4	contiguous	cacheable
		in_B	1024 * 4	contiguous	cacheable
		out_C	1024 * 4	paged	cacheable

- Open the lab1\_prebuilt/SDRelease/\_sds/swstubs/mmult.cpp file, to see how the SDSoC system compiler replaced the original mmult function with one named \_p0\_mmult\_0 that performs transfers to and from the FPGA using cf\_send and cf\_receive functions. The SDSoC system compiler also replaces calls to mmult with \_p0\_mmult\_0 in lab1\_prebuilt/SDRelease/\_sds/swstubs/main.cpp. The SDSoC system compiler uses these rewritten source files to build the ELF that accesses the hardware functions.

## Running the Project

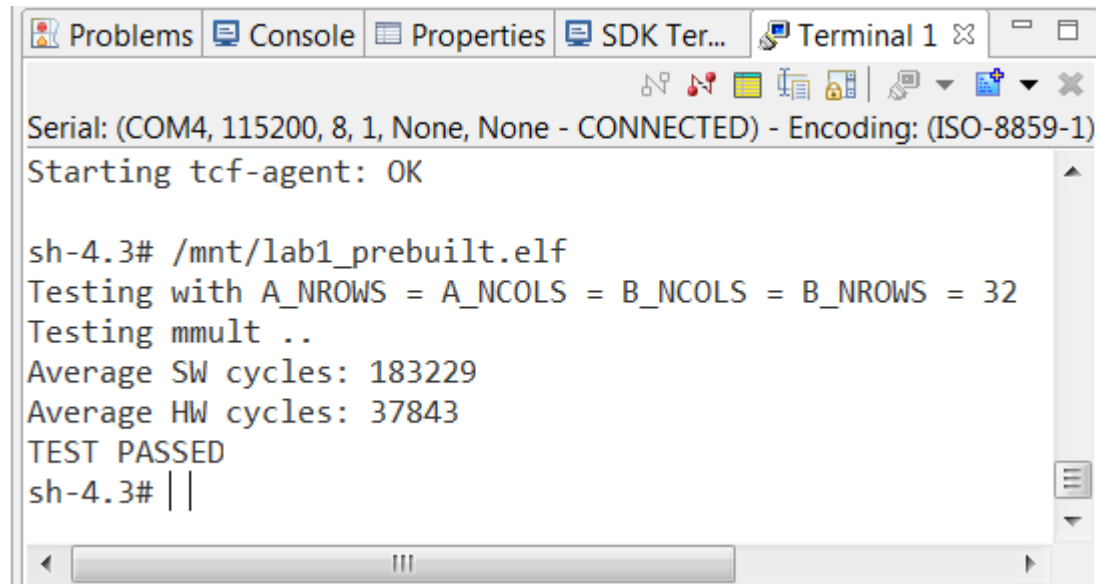
To run your project on a ZC702 board:

1. From **Project Explorer**, select the `lab1_prebuilt/SDRelease` directory and copy all files inside the `sd_card` directory to the root of an SD card.
2. Insert the SD card into the ZC702 and power on the board.
3. Connect to the board from a serial terminal in the Terminal tab of the SDSoc environment. Click the yellow-pad icon to open the settings.



4. Set up the terminal. See [Connecting the Board to a Serial Terminal](#).

5. After the board boots up, you can execute the application at the Linux prompt. Type `/mnt/lab1_prebuilt.elf`.



```

Serial: (COM4, 115200, 8, 1, None, None - CONNECTED) - Encoding: (ISO-8859-1)
Starting tcf-agent: OK

sh-4.3# /mnt/lab1_prebuilt.elf
Testing with A_NROWS = A_NCOLS = B_NCOLS = B_NROWS = 32
Testing mmult ..
Average SW cycles: 183229
Average HW cycles: 37843
TEST PASSED
sh-4.3# | |
  
```

## Questions and Additional Exercises

To test your understanding, answer the following questions.

- Why is the number of functions that can be implemented in hardware device-specific?
- What is the speedup obtained by implementing the `mmult` and `madd` kernels in hardware?
- What sub-tools are invoked by the SDSoc™ system compiler?
- Examine the contents of the `SDRelease/_sds` folder. Notice the `reports` folder. This folder contains multiple log files and report (`.rpt`) files with detailed logs and reports from all the tools invoked by the build.
- If you are familiar with Vivado® IP integrator, in the Project Explorer, double-click on `SDRelease/_sds/p0/ipi/zc702.xpr`. This is the hardware design generated from the application source code. Open the block diagram and inspect the generated IP blocks.

## Answers

- The amount of programmable logic varies from one device to another. Larger devices allow multiple functions to be implemented in hardware while smaller devices do not.
- The speedup is about 4.3 times faster. The application running on the processor takes 180k cycles while the application running on both the processor and FPGA takes 41k cycles.
- `sdscc`, `sds++`, `arm-linux-gnueabihf-gcc`, `arm-linux-gnueabihf-g++`, `vivado_hls`, `vivado`, `bootgen`
  - `sdscc` is used to compile C language sources
  - `sds++` is used to compile C++ language sources and also to link the object files created by `sdscc` and `sds++`
  - `arm-linux-gnueabihf-gcc` is called by `sdscc` to generate object code for C language sources that are targeted to the processor
  - `arm-linux-gnueabihf-g++` is called by `sds++` to generate object code for C++ language sources that are targeted to the processor, and also to link all the object files to create an executable that runs on the processor
  - `vivado_hls` is called by `sdscc/sds++` to generate RTL code for C/C++ functions that are marked for hardware implementation
  - `vivado` is called by `sds++` to generate the bitstream
  - `bootgen` is called by `sds++` to create a bootable image containing the executable that runs on the processor along with the bitstream for the PL or FPGA logic portion of the chip

# Tutorial: Working with System Optimizations

This tutorial demonstrates how you can modify your code to optimize the hardware-software system generated by the SDSoC environment. You also learn how to find more information about build errors so that you can correct your code.

**NOTE:** This tutorial is separated into steps, followed by general instructions and supplementary detailed steps allowing you to make choices based on your skill level as you progress through it. If you need help completing a general instruction, go to the detailed steps, or if you are ready, simply skip the step-by-step directions and move on to the next general instruction.

**NOTE:** You can complete this tutorial if you do not have a ZC702 board. When creating the SDSoC environment project, select your board and one of the available applications if the suggested template **Matrix Multiplication and Addition** is not found. For example, boards such as the MicroZed with smaller Zynq-7000 devices offer the **Matrix Multiplication and Addition (area reduced)** application as an available template. In this tutorial you are not asked to run the application on the board, and you can complete the tutorial following the steps for the ZC702 to satisfy the learning objectives.

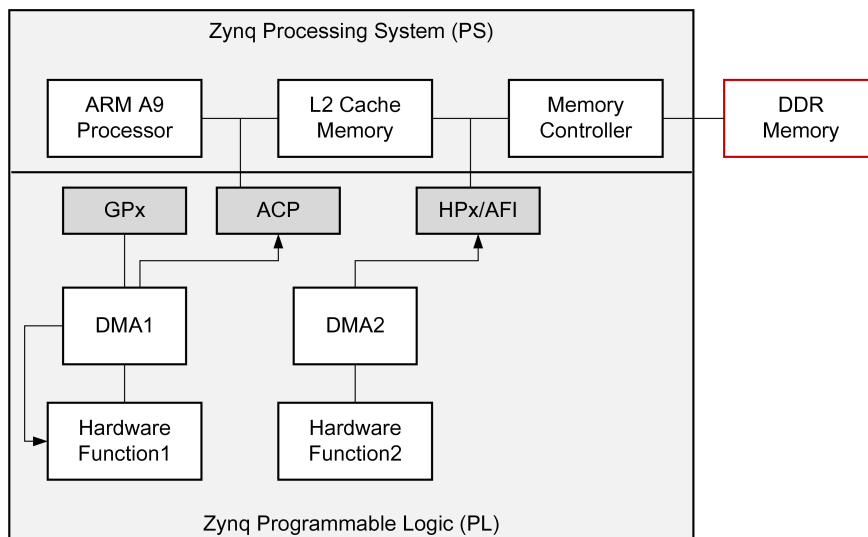
---

## Introduction to System Ports and DMA

In Zynq®-7000 All Programmable SoC device systems, the memory seen by the ARM A9 processors has two levels of on-chip cache followed by a large off-chip DDR memory. From the programmable logic side, the SDSoC environment creates a hardware design that might contain a Direct Memory Access (DMA) block to allow a hardware function to directly read and/or write to the processor system memory via the system interface ports.

As shown in the simplified diagram below, the processing system (PS) block in Zynq devices has three kinds of system ports that are used to transfer data from processor memory to the Zynq device programmable logic (PL) and back. They are Accelerator Coherence Port (ACP) which allows the hardware to directly access the L2 Cache of the processor in a coherent fashion, High Performance ports 0-3 (HP0-3), which provide direct buffered access to the DDR memory or the on-chip memory from the hardware bypassing the processor cache using Asynchronous FIFO Interface (AFI), and General-Purpose IO ports (GP0/GP1) which allow the processor to read/write hardware registers.

**Figure 3–1: Simplified Zynq + DDR Diagram Showing Memory Access Ports and Memories**



X14709\_060515

When the software running on the ARM A9 processor “calls” a hardware function, it actually calls an SDSoC environment generated stub function that calls underlying drivers to send data from the processor memories to the hardware function and to get data back from the hardware function to the processor memories over the three types of system ports shown: GPx, ACP, and AFI.

The table below shows the different system ports and their properties. The SDSoC environment automatically chooses the best possible system port to use for any data transfer, but allows you to override this selection by using pragmas.

System Port	Properties
ACP	Processor and Hardware function access the same fast cache memory as shared memory.
AFI (HPx)	Driver must flush cache to DDR before Hardware function can read the data from DDR.
GPx	Processor directly writes/reads data to/from hardware function. Inefficient for large data transfers.

## Learning Objectives

After you complete the tutorial (lab2), you should be able to:

- Use pragmas to select ACP or AFI ports for data transfer
- Observe the error detection and reporting capabilities of the SDSoC environment.

If you go through the additional exercises, you can also learn to:

- Use pragmas to select different data movers for your hardware function arguments
- Understand the use of `sds_alloc()`
- Use pragmas to control the number of data elements that are transferred to/from the hardware function.

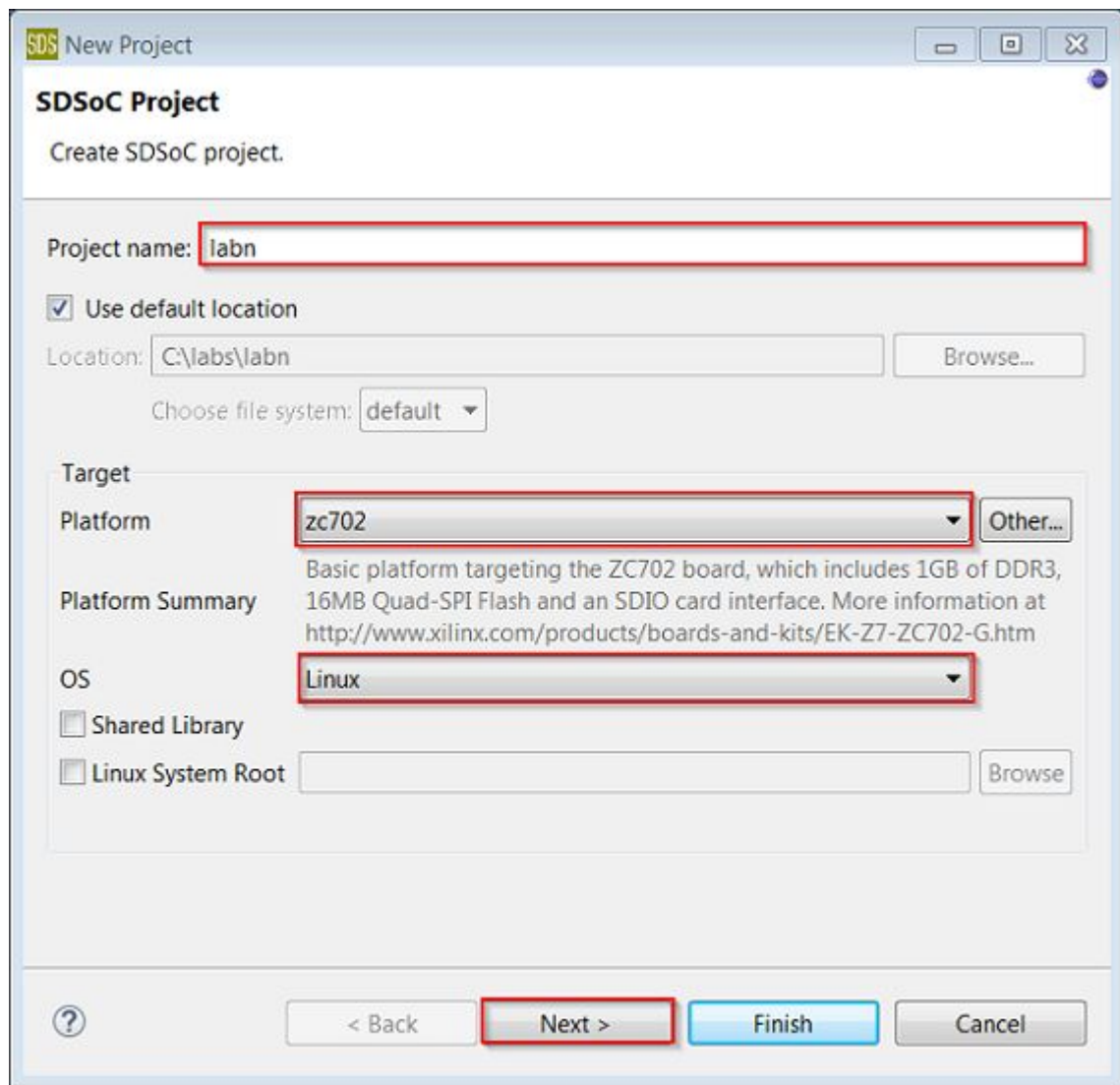
---

## Creating a New Project

To create a project in the SDSoC™ environment that performs matrix multiplication and addition:

1. Launch the SDSoC environment using the desktop icon or the **Start** menu.
2. When you launch the SDSoC environment, the **Workspace Launcher** dialog appears. Click **Browse** to enter a workspace folder used to store your projects (you can use workspace folders to organize your work), then click **OK** to dismiss the **Workspace Launcher** dialog.
3. The SDSoC environment window opens with the **Welcome** tab visible when you create a new workspace. The tab includes links for quickly getting started, for example **Create SDSoC Project**, and for accessing documentation and tutorials, for example **SDSoC User Guide**. The **Welcome** tab can be dismissed by clicking the X icon or minimized if you do not wish to use it.

4. In the **Welcome** tab click **Create SDSoC Project** or in the SDSoC menu bar select **File > New > SDSoC Project**.

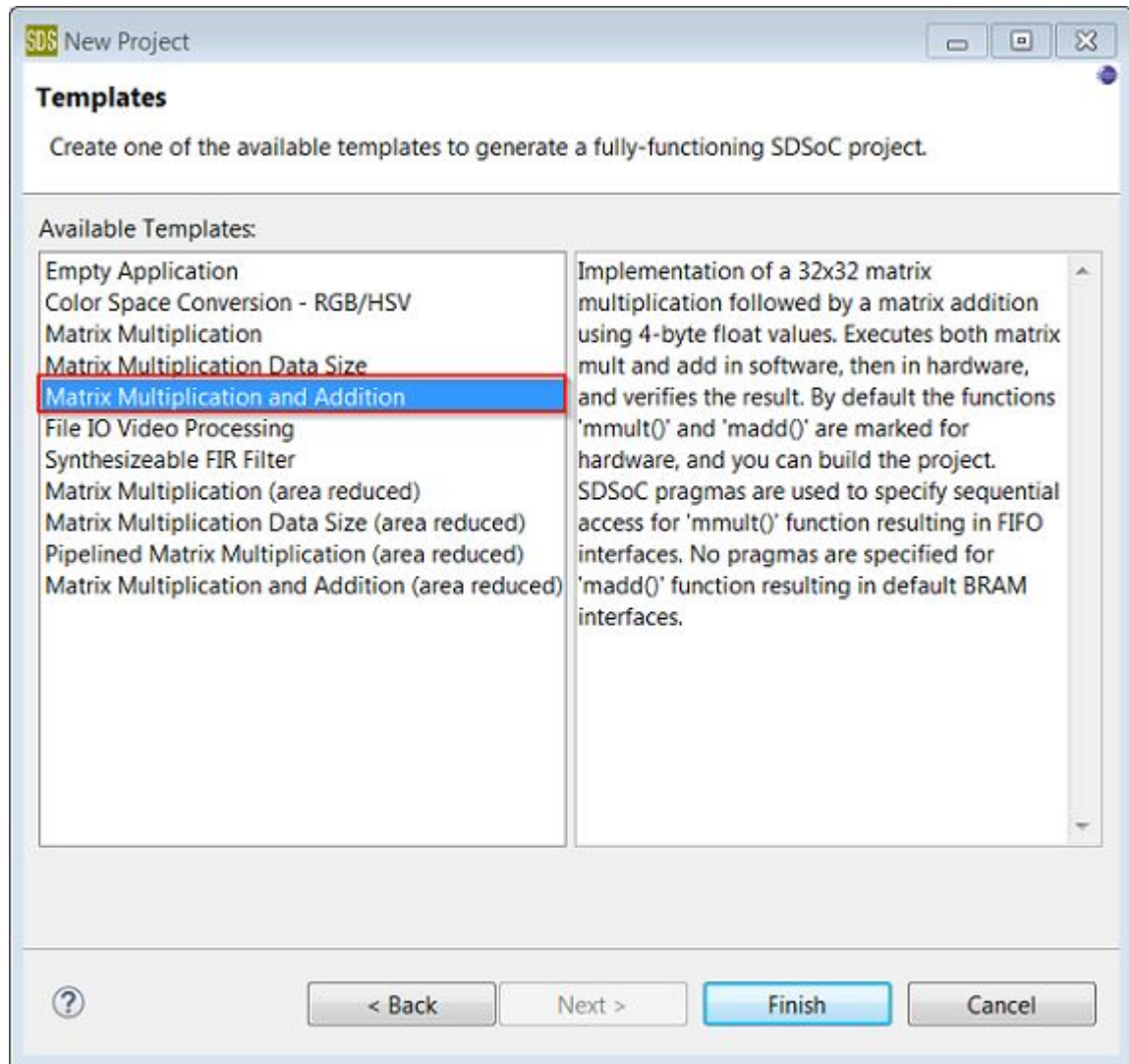


5. Specify the name of the project. The figure shows `labn` as the **Project name**, but the tutorial steps use `lab1` for the first tutorial, `lab2` for the second tutorial, etc.
6. From the **Platform** drop-down list of available platforms, select **zc702**.
7. From the **OS** drop-down list for the selected platform, select **Linux**.
8. Click **Next**.

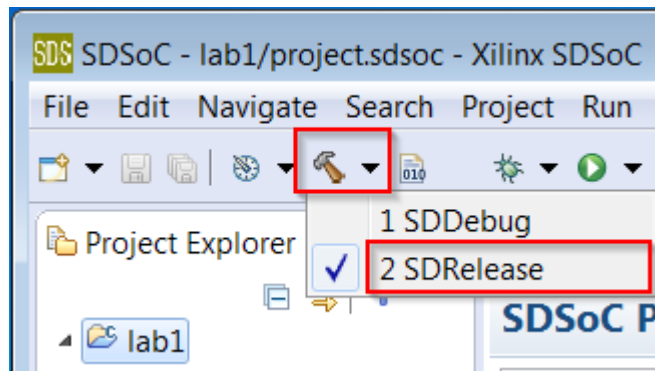
The Templates page appears, containing source code examples for the selected platform.



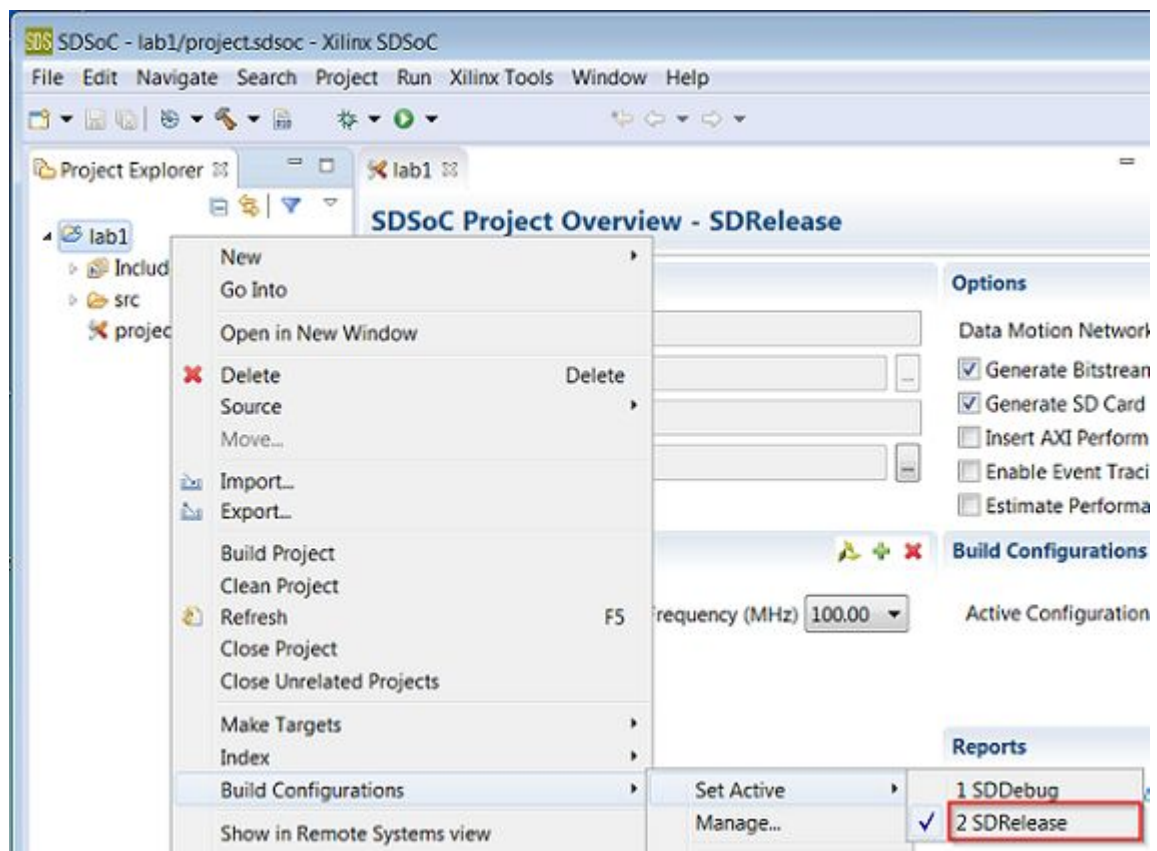
9. From the list of application templates, select **Matrix Multiplication and Addition** and click **Finish**.



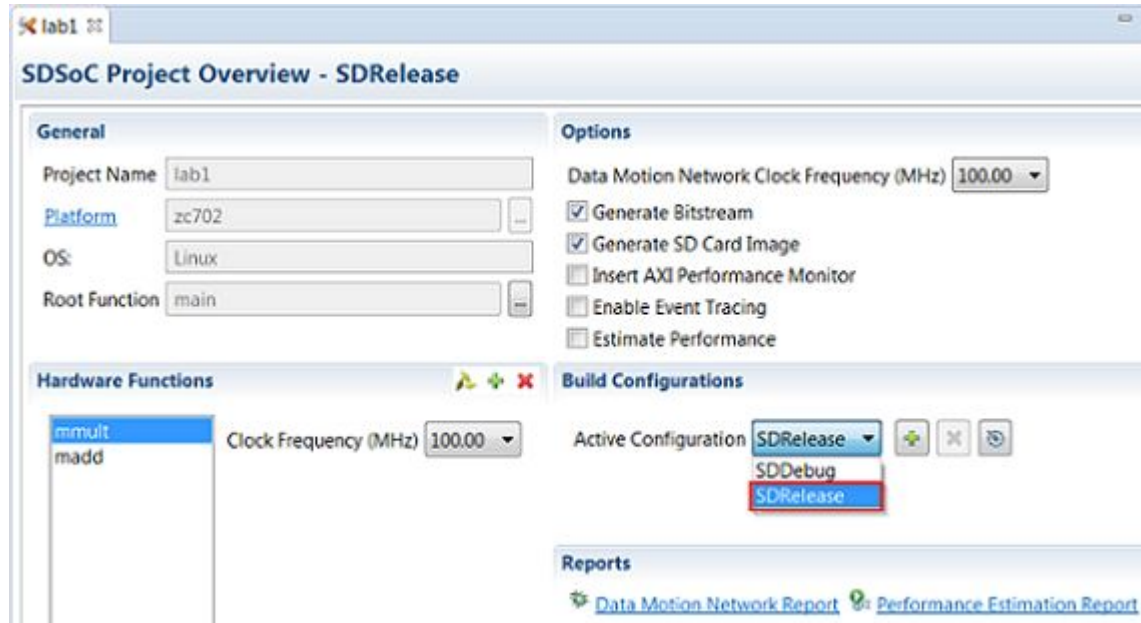
10. The standard build configurations are SDDebug and SDRelease, and you can create additional build configurations. To get the best runtime performance, switch to use the SDRelease configuration by clicking on the project and selecting **SDRelease** from the **Build** icon, or by right-clicking the project and selecting **Build Configurations > Set Active > SDRelease**. The SDRelease build configuration uses a higher compiler optimization setting than the SDDebug build configuration. The SDSoc Project Overview window also allows you to select the active configuration or create a build configuration. The **Build** icon provides a drop-down menu for selecting the build configuration and building the project. Clicking on the **Build** icon builds the project.



In the Project Explorer you can right-click on the project to select the build configuration.



The SDSoC Project Overview window includes a Build Configurations panel where you can select the active configuration or create a build configuration.




The SDSoC Project Overview window provides a summary of the project settings.

When you build an SDSoC application, you use a build configuration (a collection of tool settings, folders and files). Each build configuration has a different purpose. SDDebug builds the application with extra information in the ELF (compiled and linked program) that you need to run the debugger. The debug information in an ELF increases the size of the file and makes your application information visible. The SDRelease option provides the same ELF file as the SDDebug option with the exception that it has no debug information. The Estimate Performance option can be selected in any build configuration and is used to run the SDSoC environment in a mode used to estimate the performance of the application (how fast it runs), which requires different settings and steps (see [Tutorial: Estimating System Performance](#)).

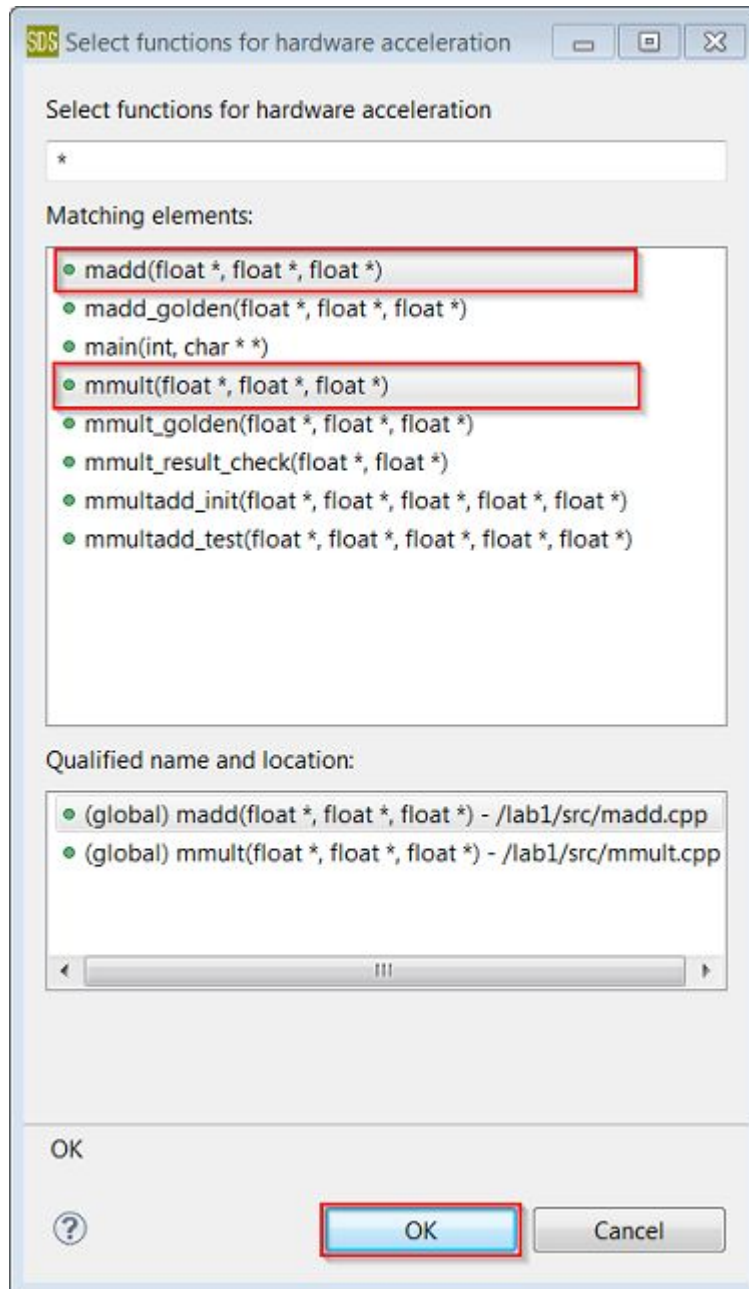
## Marking Functions for Hardware Implementation

This application has two hardware functions. One hardware function, `mmult`, multiplies two matrices to produce a matrix product, and the second hardware function, `madd`, adds two matrices to produce a matrix sum. These hardware functions are combined to compute a matrix multiply-add function. Both hardware functions `mmult` and `madd` are specified to be implemented in hardware.

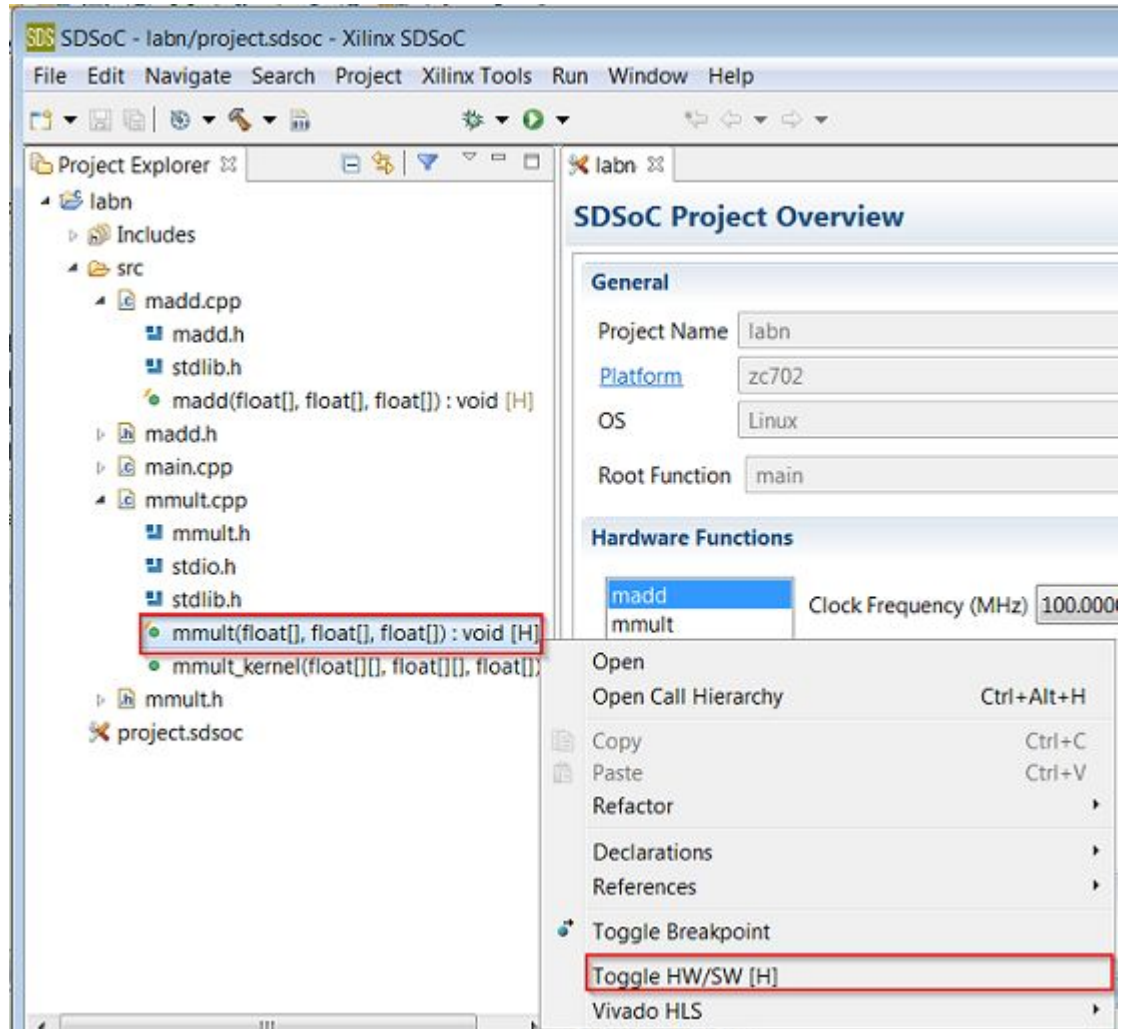
When the SDSoC environment creates the project from a template, it specifies the hardware functions for you. In cases where hardware functions have been removed or have not been specified, follow the steps below to add hardware functions (for this lab, you do not need to mark functions for hardware – the SDSoC environment has already marked them).

1. The SDSoC Project Overview window provides a central location for setting project values. Click on the tab labeled **<name of project>** (if the tab is not visible, double-click on the **project.sdsoc** file in the Project Explorer tab) and in the **Hardware Functions** panel, click on the Add Hardware Function icon  to invoke a dialog to specify hardware functions.

2. Ctrl-click (press the Ctrl key and left click) on the `mmult` and `madd` functions to select them in the "Matching elements" list. Click OK, and observe that both functions have been added to the Hardware Functions list.



Alternatively, you can expand `mmult.cpp` and `madd.cpp` in the Project Explorer, right click on `mmult` and `madd` functions, and select **Toggle HW/SW** (when the function is already marked for hardware, you will see **Toggle HW/SW [H]**). When you have a source file open in the editor, you can also select hardware functions in the Outline window.



**CAUTION!** Not all functions can be implemented in hardware. See the [SDSoC Environment User Guide \(UG1027\)](#), [Coding Guidelines](#) for more information.

## Specifying System Ports

The `sys_port` pragma allows you to override the SDSoC system compiler port selection to choose the ACP or one of the AFI ports on the Zynq-7000 AP SoC Processing System (PS) to access the processor memory.

1. You do not need to generate an SD card boot image to inspect the structure of the system generated by the SDSoC system compiler, so set project linker options to prevent generating the bit stream, boot image and build.
  - a. Click on the **lab2** tab to select the SDSoC Project Overview.
  - b. Deselect the **Generate Bit Stream** and **Generate SD Card** check boxes.



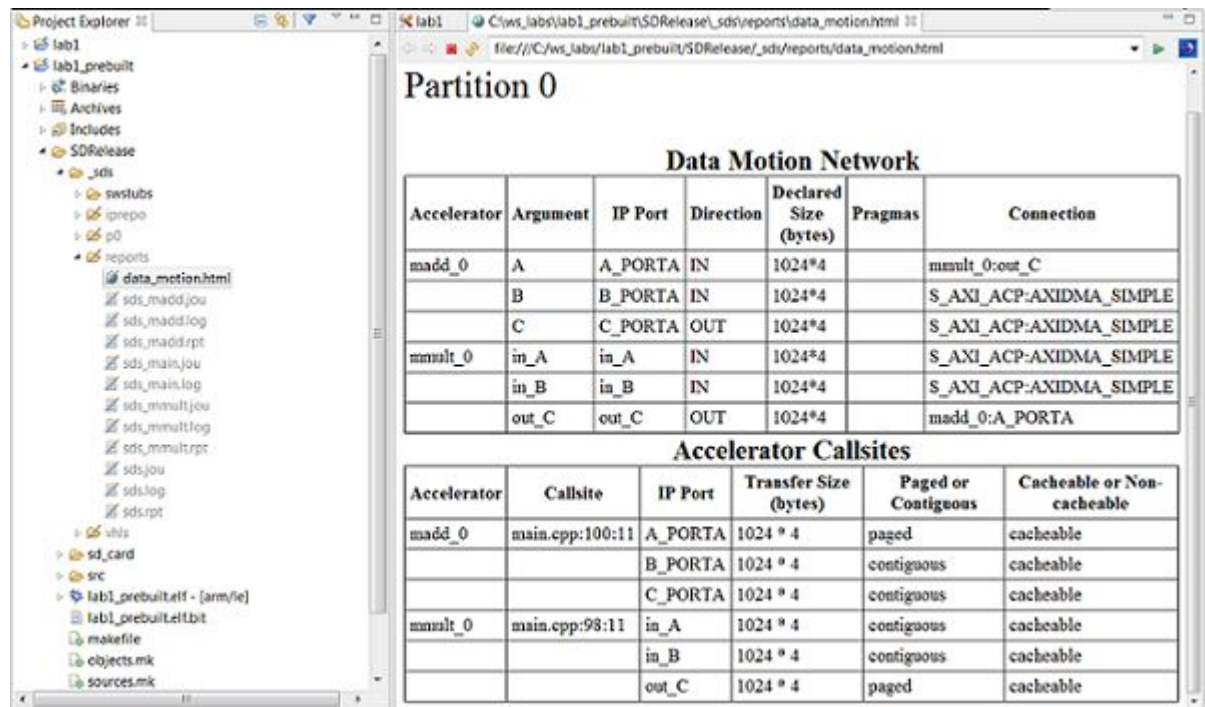
2. Right-click on the top level folder for the project in Project Explorer and select **Build Project**.



**IMPORTANT:** The build process can take approximately 5-10 minutes to complete.

3. When the build completes, in the **SDSoC Project Overview** window, under the **Reports** panel, click on **Data motion** to view the Data Motion Network report . The report contains a table describing the hardware/software connectivity for each hardware function.

The right-most column (Connection) shows the type of DMA assigned to each input array of the matrix multiplier (AXIDMA\_SIMPLE= simple DMA), and the Processing System 7 IP port used. The table below displays a partial view of the data\_motion.html file, before adding the sys\_port pragma.



**Partition 0**

**Data Motion Network**

Accelerator	Argument	IP Port	Direction	Declared Size (bytes)	Pragmas	Connection
madd_0	A	A_PORTA	IN	1024*4		mmult_0:out_C
	B	B_PORTA	IN	1024*4		S_AXI_ACP:AXIDMA_SIMPLE
	C	C_PORTA	OUT	1024*4		S_AXI_ACP:AXIDMA_SIMPLE
mmult_0	in_A	in_A	IN	1024*4		S_AXI_ACP:AXIDMA_SIMPLE
	in_B	in_B	IN	1024*4		S_AXI_ACP:AXIDMA_SIMPLE
	out_C	out_C	OUT	1024*4		madd_0:A_PORTA

**Accelerator Callsites**

Accelerator	Callsite	IP Port	Transfer Size (bytes)	Paged or Contiguous	Cacheable or Non-cacheable
madd_0	main.cpp:100:11	A_PORTA	1024 * 4	paged	cacheable
		B_PORTA	1024 * 4	contiguous	cacheable
		C_PORTA	1024 * 4	contiguous	cacheable
mmult_0	main.cpp:98:11	in_A	1024 * 4	contiguous	cacheable
		in_B	1024 * 4	contiguous	cacheable
		out_C	1024 * 4	paged	cacheable

4. Add sys\_port pragma.
  - a. Double-click mmult.h file in the Project Explorer view, to open the file in the source editor.
  - b. Immediately preceding the declaration for the mmult function, insert the following to specify a different system port for each of the input arrays.
 

```
#pragma SDS data sys_port(in_A:ACP, in_B:AFI)
```
  - c. Save the file.
5. Right-click the top-level folder for the project and click on **Clean Project** in the menu.
6. Right-click the top-level folder for the project and click on **Build Project** in the menu.



**IMPORTANT:** The build process can take approximately 5-10 minutes to complete.

7. When the build completes, open the `data_motion.html` file.

The connection column shows the system port assigned to each input/output array of the matrix multiplier.

---

## Error Reporting

You can introduce errors as described in each of the following steps and note the response from the SDSoC environment.

1. Open the source file `main.cpp` and remove the semicolon at the end of the `std::cout` statement near the bottom of the file.  
Notice how a yellow box shows up on the left edge of the line.
2. Move your cursor over the yellow box and notice that it tells you that you have a missing semicolon.
3. Insert the semicolon at the right place and notice how the yellow box disappears.
4. Now change `std::cout` to `std::cou` and notice how a pink box shows up on the left edge of the line.
5. Move the cursor over the pink box to see a popup displaying the "corrected" version of the line with `std::cout` instead of `std::cou`.
6. Correct the previous error by changing `std::cou` to `std::cout`.
7. Introduce a new error by commenting out the line that declares all the variables used in `main()`.
8. Save and build the project. Do not wait for the build to complete.
9. You can see the error messages scrolling by on the console.  
Open the `SDRelease/_sds/reports/sds.log` and `SDRelease/_sds/reports/sds_mmult.log` files to see the detailed error reports.

---

## Additional Exercises

**NOTE:** Instructions provided in this section are optional.

When Linux is used as the target OS for your application, memory allocation for your application is handled by Linux and the supporting libraries. If you declare an array on stack within a scope (`int a[10000];`) or allocate it dynamically using the standard `malloc()` function, what you get is a section of memory that is contiguous in the Virtual Address Space provided by the processor and Linux. This buffer is typically split over multiple non-contiguous pages in the Physical Address Space, and Linux automatically does the Virtual-Physical address translation whenever the software accesses the array. However, the hardware functions and DMAs can only access the physical address space, and so the software drivers have to explicitly translate from the Virtual Address to the Physical Address for each array, and provide this physical address to the DMA or hardware function. As each array may be spread across multiple non-contiguous pages in Physical Address Space, the driver has to provide a list of physical page addresses to the DMA. DMA that can handle a list of pages for a single array is known as Scatter-Gather DMA. A DMA that can handle only single physical addresses is called Simple DMA. Simple DMA is cheaper than Scatter-Gather DMA in terms of the area and performance overheads, but it requires the use of a special allocator called `sds_alloc()` to obtain physically contiguous memory for each array.

[Tutorial: Creating, Building and Running a Project](#) used `sds_alloc()` to allow the use of Simple DMA. In the following exercises you force the use of other data movers such as Scatter-Gather DMA or AXIFIFO using pragmas, modify the source code to use `malloc()` instead of `sds_alloc()` and observe how Scatter-Gather DMA is automatically selected.

## Controlling Data Mover Selection

In this exercise you add data mover pragmas to the source code from [Tutorial: Creating, Building and Running a Project](#) (lab1) to specify the type of data mover used to transfer each array between hardware and software. Then you build the project and view the generated report (`data_motion.html`) to see the effect of these pragmas. Remember to prevent generation of bit stream and boot files, so that your build does not synthesize the hardware.

To add data mover pragmas to specify the type of data mover used for each array:

1. Double click `mmult.h` in the folder view under `lab1/src` to bring up the source editor panel.
2. Just above the `mmult` function declaration, insert the following line to specify a different data mover for each of the arrays and save the file.

```
#pragma SDS data data_mover(in_A:AXIDMA_SG, in_B:AXIDMA_SIMPLE, out_C:AXIFIFO)
```

3. Right-click the top-level folder for the project and click **Clean Project** in the menu.
4. Right-click the top-level folder for the project and click **Build Project** in the menu.




---

**IMPORTANT:** *The build process can take approximately 5 to 10 minutes to complete.*

---

5. When the build completes, in the Project Explorer view, double-click to open `SDRelease/_sds/reports/data_motion.html`.

The right-most column (Connection) shows the data mover assigned to each input/output array of the matrix multiplier.

**NOTE:** The Pragmas column lists the pragmas that were used. Also, the `AXIFIFO` data mover has been assigned the `M_AXI_GP0` port, while the other two data movers are associated with `S_AXI_ACP`.



## Using malloc() instead of sds\_alloc()

For this exercise you start with the source used in [Tutorial: Creating, Building and Running a Project](#) (lab1), modify the source to use `malloc()` instead of `sds_alloc()`, and observe how the data mover changes from Simple DMA to Scatter-Gather DMA. Disable generation of the bitstream and boot files by unselecting **Generate Bit Stream** and **Generate SD Card**. If you are continuing from the previous section, you must delete the data mover pragma on the `mmult` declaration in `lab1/src/mmult.h`.

1. Double-click the `main.cpp` in the Project Explorer view to bring up the source editor view.
2. Find all the lines to where buffers are allocated with `sds_alloc()`, and replace `sds_alloc()` with `malloc()` everywhere. Also remember to replace all calls to `sds_free()` with `free()`.
3. Save your file.
4. Right-click the top-level folder for the project and click **Clean Project** in the menu.
5. Right-click the top-level folder for the project and click **Build Project** in the menu.




---

**IMPORTANT:** *The build process can take approximately 5 to 10 minutes to complete.*

---

6. When the build completes, in the Project Explorer view, double-click to open `SDRelease/_sds/reports/data_motion.html`.
7. The right-most column (Connection) shows the type of DMA assigned to each input/output array of the matrix multiplier (`AXIDMA_SG` = scatter gather DMA), and which Processing System 7 IP port is used (`S_AXI_ACP`). You can also see on the Accelerator Call sites table whether the allocation of the memory that is used on each transfer is contiguous or paged.

## Using Vivado HLS based Accelerator Optimizations

In this exercise, you modify the source from Lab 1 to observe the effects of Vivado HLS pragmas on the performance of generated hardware. See [SDSoC Environment User Guide \(UG1027\)](#), [A Programmer's Guide to Vivado High-Level Synthesis](#) for more information on this topic. Enable generation of the bitstream and boot files by selecting **Generate Bit Stream** and **Generate SD card**. If you are continuing from the previous section, change the `malloc()` calls back to `sds_alloc()` and `free()` to `sds_free()`.

1. Double click the `mmult.cpp` in the Project Explorer view to bring up the source editor view.
2. Find the lines where the pragmas `HLS pipeline` and `HLS array_partition` are located.
3. Remove these pragmas by commenting out the lines.
4. Save your file.
5. Right click the top-level folder for the project and click **Clean Project** in the menu.
6. Right click the top-level folder for the project and click **Build Project** in the menu.

7. After the build completes, copy the `sd_card` folder to an SD card and run it on the board.

Observe the performance and compare it with the performance that was seen with the commented out pragmas present. Note that the `array_partition` pragmas increase the memory bandwidth for the inner loop by allowing array elements to be read in parallel. The pipeline pragma on the other hand performs pipelining of the loop and allows multiple iterations of a loop to run in parallel.

## Adding Pragmas to Control the Amount of Data Transferred

For this step, you use a different design template to show the use of the copy pragma. In this template an extra parameter called `dim1` is passed to the matrix multiply function. This parameter allows the matrix multiplier function to multiply two square matrices of any size `dim1*dim1` up to a maximum of `32*32`. The top level allocation for the matrices creates matrices of the maximum size `32x32`. The `dim1` parameter tells the matrix multiplier function the size of the matrices to multiply, and the data copy pragma tells the SDSoc™ environment that it is sufficient to transfer a smaller amount of data corresponding to the actual matrix size instead of the maximum matrix size.

1. Launch the SDSoc environment and create a new project for the zc702, Linux platform using the matrix multiplication with variable data size design template:
  - a. Select **File > New > SDSocProject**.
  - b. In the new project dialog box, type in a name for the project (for example `lab2a`)
  - c. Select **zc702** and **Linux**.
  - d. Click **Next**.
  - e. Select **Matrix Multiplication Data Size** as the application and click **Finish**.
  - f. Note that the `mmult_accel` function has been marked for hardware acceleration.
2. Set up the project to prevent building the bitstream and boot files.
3. Add data copy pragmas by double-clicking **mmult\_accel.h** in the Project Explorer view to bring up the source editor view.

Note the pragmas that specify a different data copy size for each of the arrays. In the pragmas, you can use any of the scalar arguments of the function to specify the data copy size. In this case, `dim1` is used to specify the size.

```
#pragma SDS data copy(in_A[0:dim1*dim1])
#pragma SDS data copy(in_B[0:dim1*dim1])
#pragma SDS data copy(out_C[0:dim1*dim1])
void mmult_accel (float in_A[A_NROWS*A_NCOLS],
                  float in_B[A_NCOLS*B_NCOLS],
                  float out_C[A_NROWS*B_NCOLS],
                  int dim1);
```

4. Right-click the top-level folder for the project and click **Clean Project** in the menu.
5. Right-click the top-level folder for the project and click **Build Project** in the menu.




---

**IMPORTANT:** *The build process may take approximately 5 to 10 minutes to complete.*

---

6. When the build completes, in the Project Explorer view, double-click to open `SDDebug/_sds/reports/data_motion.html`.

7. Observe the second column from the right, titled **Pragmas**, to view the length of the data transfer for each array. The second table shows the transfer size for each hardware function call site.

## Partition 0

### Data Motion Network

Accelerator	Argument	IP Port	Direction	Declared Size(bytes)	Pragmas	Connection
mmult_accel_0	in_A	in_A	IN	1024*4	• length:(dim1*dim1)	S_AXI_ACP:AXIDMA_SIMPLE
	in_B	in_B	IN	1024*4	• length:(dim1*dim1)	S_AXI_ACP:AXIDMA_SIMPLE
	out_C	out_C	OUT	1024*4	• length:(dim1*dim1)	S_AXI_ACP:AXIDMA_SIMPLE
	dim1	dim1	IN	4		M_AXI_GP0:AXILITE:0x80

### Accelerator Callsites

Accelerator	Callsite	IP Port	Transfer Size(bytes)	Paged or Contiguous	Cacheable or Non-cacheable
mmult_accel_0	mmult.cpp:78:5	in_A	(dim1*dim1) * 4	contiguous	cacheable
		in_B	(dim1*dim1) * 4	contiguous	cacheable
		out_C	(dim1*dim1) * 4	contiguous	cacheable
		dim1	4	paged	cacheable

# Tutorial: Debugging Your System

This tutorial demonstrates how to use the interactive debugger in the SDSoC™ environment.

First, you target your design to a standalone operating system or platform, run your standalone application using the Xilinx SDSoC environment, and debug the application.

You then create a Linux application and use the interactive debugger to step through your code.

In this tutorial you are debugging applications running on an accelerated system.

**NOTE:** This tutorial is separated into steps, followed by general instructions and supplementary detailed steps allowing you to make choices based on your skill level as you progress through it. If you need help completing a general instruction, go to the detailed steps, or if you are ready, simply skip the step-by-step directions and move on to the next general instruction.

**NOTE:** You can complete this tutorial if you do not have a ZC702 board. When creating the SDSoC environment project, select your board and one of the available applications if the suggested template **Matrix Multiplication and Addition** is not found. For example, boards such as the MicroZed with smaller Zynq-7000 devices offer the **Matrix Multiplication and Addition (area reduced)** application as an available template. Any application can be used to learn the objectives of this tutorial. A pre-built project is only available for the ZC702 board.

---

## Learning Objectives

After you complete the tutorial, you should be able to:

- Use the SDSoC environment to download and run your standalone application.
- Optionally step through your source code in the SDSoC environment (debug mode) and observe various registers and memories. Note that this is limited to code running on the ARM A9, and does not apply to code that has been converted into hardware functions.

---

## Setting Up the Board

You need a mini USB cable to connect to the UART port on the board, which talks to a serial terminal in the SDSoC environment. You also need a micro USB cable to connect to the Digilent port on the board to allow downloading the bitstream and binaries. Finally, you need to ensure that the jumpers to the side of the SD card slot are set correctly to allow booting from an SD card.

1. Connect the mini USB cable to the UART port. See [Connecting the Board to a Serial Terminal](#).

2. Ensure that the JTAG mode is set to use the Digilent cable and that the micro USB cable is connected.



3. Set the jumpers to SD-boot mode but do not plug in an SD card. See [Configuring the Board for SD Card Boot](#).
4. Power on the board.

Ensure that you allow Windows to install the USB-UART driver and the Digilent driver to enable the SDSoc environment to communicate with the board.



**IMPORTANT:** Make sure that the jumper settings on the board correspond to SD-boot or JTAG-boot. Otherwise the board may power up in some other mode such as QSPI boot, and attempt to load something from the QSPI device or other boot device, which is not related to this lab.

## Creating a Standalone Project

Create a new SDSoc™ environment project (lab3) for the ZC702 platform and Standalone OS using the design template for Matrix Multiplication and Addition.


To create a standalone project in the SDSoc environment:

1. Launch the SDSoc environment.
2. Select **File > New > SDSocProject**.
3. Specify the name of the project in the **Project name** field. For example, lab3.
4. From the **Platform** drop-down list, select **zc702**.
5. From the **OS** drop-down list, select **Standalone**.



6. Click **Next**.

The Templates page appears.

7. From the list of application templates, select **Matrix Multiplication and Addition** and click **Finish**.
8. Click on the tab labeled **lab3** to select the **SDSoC Project Overview** (if the tab is not visible, double click on the `project.sdsoc` file in the **Project Explorer**) and in the **Hardware Functions** panel, observe that the `mmult` and `madd` functions were marked as hardware functions when the project was created.
9. If hardware functions were removed or not marked, you would click on the **Add Hardware** icon  to invoke the dialog box to specify hardware functions. Ctrl-click (press the Ctrl key and left click) on the `mmult` and `madd` functions to select them in the **Matching Elements** list. Click **OK** and observe that both functions have been added to the **Hardware Functions** list.
10. In the **Project Explorer** right-click the project and select **Build Project** from the context menu that appears.

SDSoC builds the project. A dialog box displaying the status of the build process appears.




---

**IMPORTANT:** *The build process might take approximately 30 to 45 minutes to complete. Instead of building the project you can save time and instead use the pre-built project. (To minimize disk usage in the SDSoC installation, the imported project might contain fewer files than a project you build, but it includes the files required to complete the tutorial.) To import a pre-built project: select **File > Import** and then select **General > Existing Projects into Workspace** and click **Next**. Click **Select archive file** and browse to find the `lab3_standalone_prebuilt.zip` file provided in the project files folder (`<path to install>/SDSoC/2016.2/docs/labs/lab3_standalone_prebuilt.zip`). Click **Open**. Click **Finish**.*

---

**NOTE:** If the project is imported, its binary ELF file does not have the correct paths for source debugging. You would need to rebuild the ELF but you do not want to rebuild the programmable logic bitstream. In the **Project Explorer** expand the **lab3\_standalone\_prebuilt** project and double-click `project.sdsoc` to display the **SDSoC Project Overview**. In the **Options** panel, uncheck the **Generate Bit Stream** box and leave the **Generate SD Card** box checked. Clean the project (right click on **lab3\_standalone\_prebuilt** and select **Clean Project**) and rebuild it (right click on **lab3\_standalone\_prebuilt** and select **Build Project**).

## Setting up the Debug Configuration

To set up the debug configuration:


1. In the Project Explorer view click on the ELF (.elf) file in the SDDebug folder in the **lab3\_standalone\_prebuilt** project and in the toolbar click on the **Debug** icon or use the **Debug** icon pull-down menu to select **Debug As > Launch on Hardware (SDSoC Debugger)**. Alternatively, right-click the project and select **Debug As > Launch on Hardware (SDSoC Debugger)**. The **Confirm Perspective** Switch dialog box appears.




---

**IMPORTANT:** *Ensure that the board is switched on before debugging the project.*

---


2. Click **Yes** to switch to the debug perspective.  
 You are now in the **Debug** Perspective of the SDSoC environment. Note that the debugger resets the system, programs and initializes the device, then breaks at the `main` function. The source code is shown in the center panel, local variables in the top right corner panel and the SDK log at the bottom right panel shows the Debug configuration log.
3. Before you start running your application you need to connect a serial terminal to the board so you can see the output from your program. In this example, we are using the SDSoC environment Terminal view invoked by **Window > Show View > Other** and selecting **Terminal > Terminal**. Click the **Terminal** tab near the bottom of the **Debug** perspective (configured with **Connection Type:** Serial, **Port:** COM<n>, **Baud Rate:** 115200 baud), and then click the **Connect** icon  to connect the terminal to the board (which should be powered up already).

---

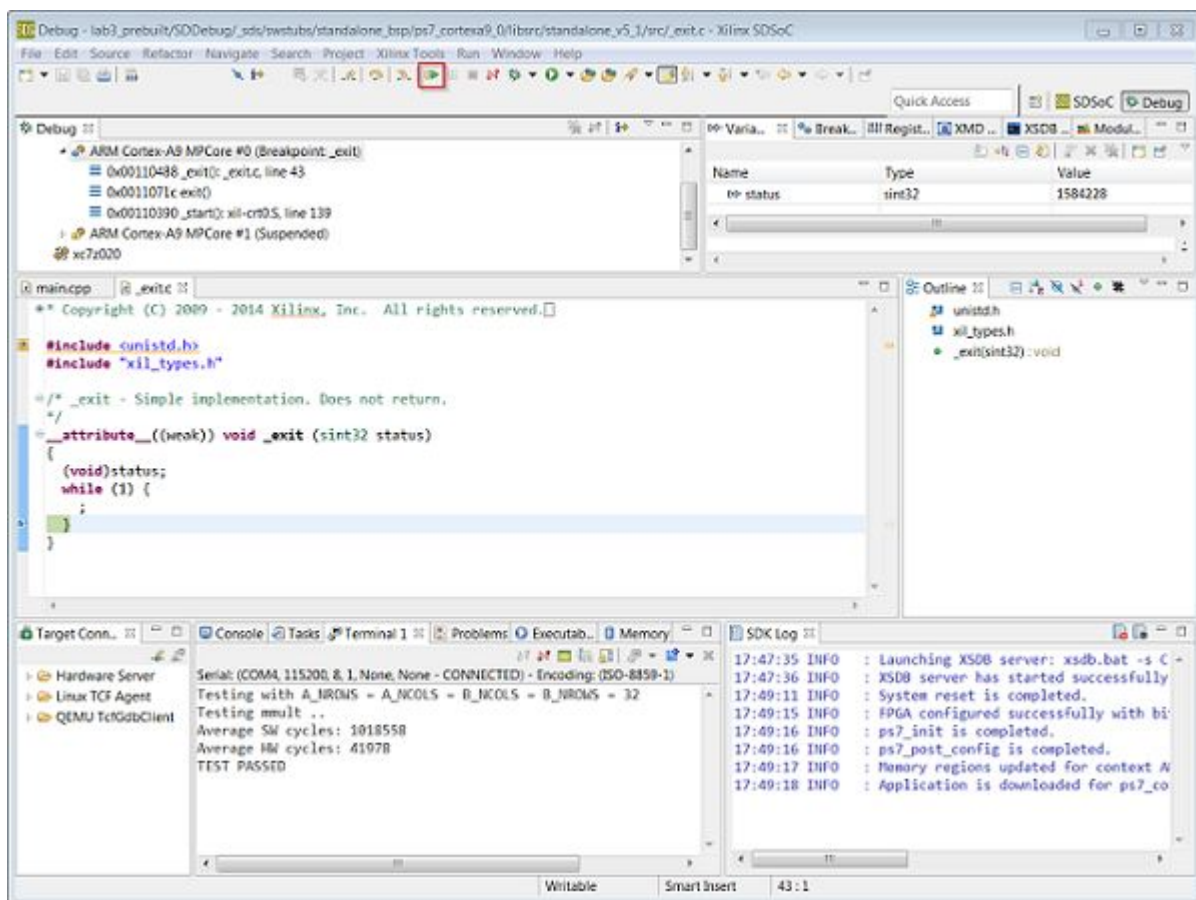
## Running the Application

To run your application:



Click the **Resume** icon  to run your application, and observe the output in the terminal window.

**NOTE:** The source code window shows the `_exit` function, and the terminal tab shows the output from the matrix multiplication application.



## Additional Exercises

**NOTE:** Instructions provided in this section are optional.

You can learn how to debug/step through the application, run in SD Boot mode and debug a Linux application

## Stepping Through the Code

The Debug perspective has many other capabilities that have not been explored in this lab. The most important is the ability to step through the code to debug it.

1. Right-click the folder at the top of the debug hierarchy in the Debug view, and click **Disconnect** in the menu.

2. Right-click the top-level debug folder again, and click **Remove all Terminated** in the menu.
3. Click on the BUG icon to launch the debugger. Then step through the code using the step-into, step-over, and step-return buttons.
4. As you step through the code, examine the values of different variables.




---

**IMPORTANT:**

*If you try to use the terminate and relaunch buttons, the SDSoc™ environment might display an error message saying that it failed to launch. If that happens, restart the SDSoc environment, and cycle power on the board.*

*If you missed [Setting up the Debug Configuration](#) and did not disable bitstream generation, the SDSoc environment attempts to clean and rebuild the application when you try to launch the debugger, and this can take up to 30 mins for this example.*

---

## Boot From SD Card

1. Copy the `BOOT.BIN` file from the `sd_card` folder inside the Debug folder to an SD card using Windows file copy/paste.
2. Plug in the SD card on the ZC702 board, confirm the jumpers are set to sd-boot mode and power on the board.
3. Ensure that the terminal tab in the SDSoc IDE is still connected, and view the output of your application on that terminal.

## Debugging Linux Applications

To debug a Linux application in the SDSoc environment:

1. Create or select a project targeted to ZC702 and Linux.  
For details, see [Creating a New Project](#).
2. Mark a function for hardware implementation.  
For details, see [Marking Functions for Hardware Implementation](#).

3. Build a project and generate executable, bitstream, and SD Card boot image.

For details, see [Building a Design with Hardware Accelerators](#).





**IMPORTANT:** *Building the executable can take 30 to 60 minutes depending on your machine. Instead of building the project you can save time and instead use the pre-built project. (To minimize disk usage in the SDSoC installation, the imported project might contain fewer files than a project you build, but it includes the files required to complete the tutorial.) To import a pre-built project: select **File > Import** and then select **General > Existing Projects into Workspace** and click **Next**. Click **Select archive file** and browse to find the `lab3_linux_prebuilt.zip` file provided in the project files folder (`<path to install>/SDSoC/2016.2/docs/labs/lab3_linux_prebuilt.zip`). Click **Open**. Click **Finish**.*

**NOTE:** If the project is imported, its binary ELF file does not have the correct paths for source debugging. You would need to rebuild the ELF but you do not want to rebuild the programmable logic bitstream. In the **Project Explorer** expand the **lab3\_linux\_prebuilt** project and double-click `project.sdsoc` to display the **SDSoC Project Overview**. In the **Options** panel, uncheck the **Generate Bit Stream** box and leave the **Generate SD Card** box checked. Clean the project (right click on **lab3\_linux\_prebuilt** and select **Clean Project**) and rebuild it (right click on **lab3\_linux\_prebuilt** and select **Build Project**).

4. Here we are using the SDSoC environment Terminal view invoked from **Window > Show View > Other** and selecting **Terminal > Terminal**. Click the **Terminal** tab near the bottom of the Debug window and confirm the settings (**Connection Type:** Serial, **Port:** COM<n>, **Baud Rate:** 115200 baud).

For the COM port settings to be visible, the board must be powered up:

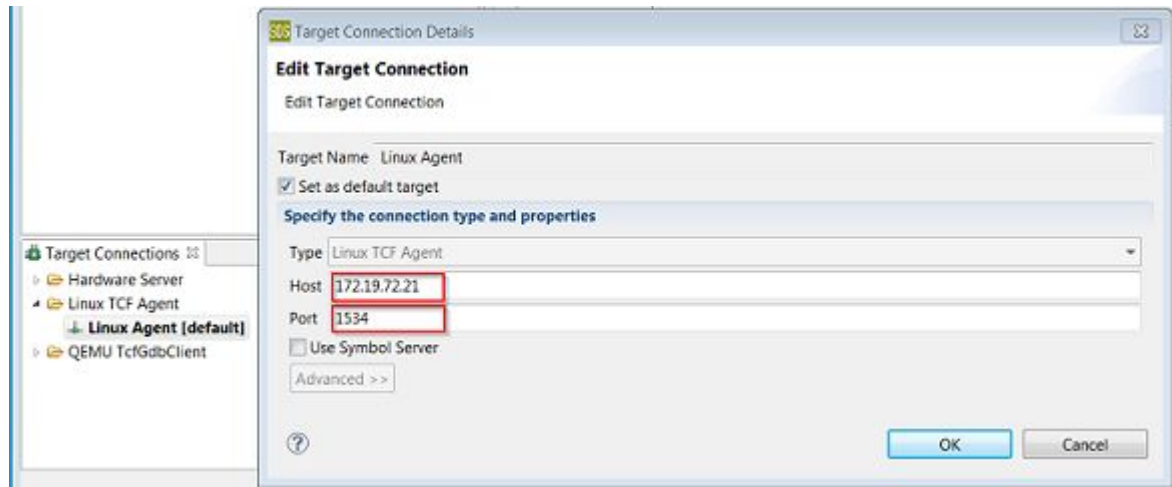
- Power up the board without an SD card plugged in.
- Click on the Terminal Settings icon , set the configuration and click **OK**.
- The terminal indicates it is connected. Click the red disconnect icon  to disconnect the terminal from the board, and power off the board.

5. Copy the contents of the generated `sd_card` directory to an SD card, and plug the SD card into the ZC702 board.
6. Ensure that the board is connected to an Ethernet router using an Ethernet cable. Power on the board. Click on the Terminal tab and click the green connection icon to connect the terminal to the board. The Linux boot log is displayed on the terminal. Look for a line that says `Sending select for 172.19.73.248...Lease of 172.19.73.248 obtained`, where the IP address assigned to your board is reported. Take a note of this address for use in the next step.

If you do not see the IP address, type the command `ifconfig` in the terminal and take a note the IP address.

7. Back in the SDSoC environment in the **Target Connections** view, expand **Linux TCF Agent** and right-click on **Linux Agent (default)**, then select **Edit**.

8. In the Target Connection Details dialog set up the IP address and port (1534).



9. Click **OK**.
10. In the Project Explorer click on the ELF file to select it and click on the **Debug** icon in the toolbar (or use the **Debug** icon pull-down menu to select **Debug As > Launch on Hardware (SDSoC Debugger)**) to go to the Debug perspective, and run or step through your code.

**NOTE:** Your application output displays in the Console view instead of the Terminal view.

# Tutorial: Estimating System Performance

This tutorial demonstrates how to obtain an estimate of the expected performance of an application, without going through the entire build cycle.

**NOTE:** This tutorial is separated into steps, followed by general instructions and supplementary detailed steps, allowing you to make choices based on your skill level as you progress through it. If you need help completing a general instruction, go to the detailed steps, or if you are ready, simply skip the step-by-step directions and move on to the next general instruction.

**NOTE:** You can complete this tutorial if you do not have a ZC702 board. When creating the SDSoC environment project, select your board and one of the available applications if the suggested template **Matrix Multiplication and Addition** is not found. For example, boards such as the MicroZed with smaller Zynq-7000 devices offer the **Matrix Multiplication and Addition (area reduced)** application as an available template. A different application can be used to learn the objectives of this tutorial, as long as the application exits (this is a requirement to run the instrumented application on the board to collect software runtime data). Consult your board documentation for setup information.

---

## Learning Objectives

After you complete the tutorial, you should be able to use the SDSoC environment to obtain an estimate of the speedup that you can expect from your selection of functions to accelerate.

[Additional Exercises](#)

---

## Setting Up the Board

You need a mini USB cable to connect to the UART port on the board, which talks to a serial terminal in the SDSoC environment. You also need a micro USB cable to connect to the Digilent port on the board to allow downloading the bitstream and binaries. Finally, you need to ensure that the jumpers to the side of the SD card slot are set correctly to allow booting from an SD card.

1. Connect the mini USB cable to the UART port. See [Connecting the Board to a Serial Terminal](#).



2. Ensure that the JTAG mode is set to use the Digilent cable and that the micro USB cable is connected.



3. Set the jumpers to SD-boot mode but do not plug in an SD card. See [Configuring the Board for SD Card Boot](#).
4. Power on the board.

Ensure that you allow Windows to install the USB-UART driver and the Digilent driver to enable the SDSoc environment to communicate with the board.




**IMPORTANT:** Make sure that the jumper settings on the board correspond to SD-boot or JTAG-boot. Otherwise the board may power up in some other mode such as QSPI boot, and attempt to load something from the QSPI device or other boot device, which is not related to this lab.

## Setting up the Project for Performance Estimation

To create a project and use the Estimate Performance option in a build configuration:

1. Create a new project in the SDSoc™ environment (lab4) for the ZC702 platform and Standalone using the design template for Matrix Multiplication and Addition.
2. Click on the tab labeled **lab4** to view the **SDSoc Project Overview**. If the tab is not visible, in the **Project Explorer** double click on the `project.sdsoc` file under the **lab4** project.
3. In the **Hardware Functions** panel, observe that the `madd` and `mmult` functions already appear in the list of functions marked for hardware – template projects in the SDSoc environment include information for automating the process of marking hardware functions.

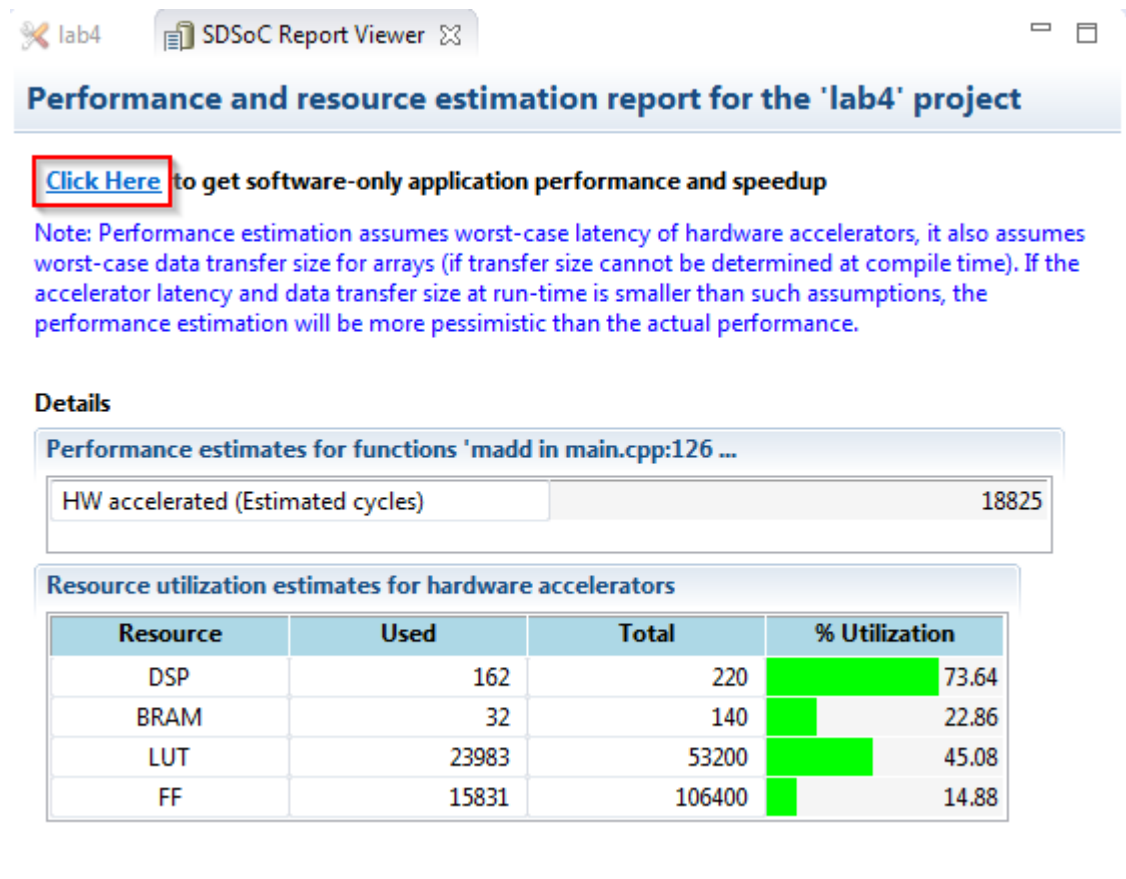
4. If the **Hardware Functions** panel did not list any functions, you would click on the **Add Hardware Function** icon (  ) to invoke a dialog for specifying hardware functions. Ctrl-click (press the Ctrl key and left click simultaneously) on the `madd` and `mmult` functions in the **Matching elements:** list and notice that they appear in the **Qualified name and location:** list. Click **OK**.
5. Performance estimation can be run using any build configuration. Instead of selecting `SDDebug` or `SDRelease` as the Active Configuration in the **Build Configurations** panel, you could instead click on the **Create New Configuration** icon next to the active configuration. When creating the build configuration, specify an existing build configuration to use as a starting point and select it as the active configuration after creating it. Using the `SDDebug` build configuration or another build configuration copied from `SDDebug` will compile the code with `-O0` using GCC, so the software performance will be significantly degraded.
6. In the **SDSoC Project Overview** in the **Actions** panel, check the **Estimate Performance** box. This enables the estimation flow.



- The **Build** icon provides a drop-down menu for selecting the build configuration and building the project. Clicking on the **Build** icon builds the project and with the **Estimate Performance** option checked, the performance estimation flow runs.

The SDSoC environment builds the project. A dialog box displaying the status of the build process appears.

After the build is over, you can see an initial report. This report contains a hardware-only estimate summary and has a link that can be clicked to obtain the software run data, which updates the report with comparison of hardware implementation versus the software-only information.



**lab4** SDSoC Report Viewer

### Performance and resource estimation report for the 'lab4' project

[Click Here](#) to get software-only application performance and speedup

**Note:** Performance estimation assumes worst-case latency of hardware accelerators, it also assumes worst-case data transfer size for arrays (if transfer size cannot be determined at compile time). If the accelerator latency and data transfer size at run-time is smaller than such assumptions, the performance estimation will be more pessimistic than the actual performance.

**Details**

Performance estimates for functions 'madd in main.cpp:126 ...

HW accelerated (Estimated cycles)	18825
-----------------------------------	-------

Resource utilization estimates for hardware accelerators

Resource	Used	Total	% Utilization
DSP	162	220	73.64
BRAM	32	140	22.86
LUT	23983	53200	45.08
FF	15831	106400	14.88

## Comparing Software and Hardware Performance

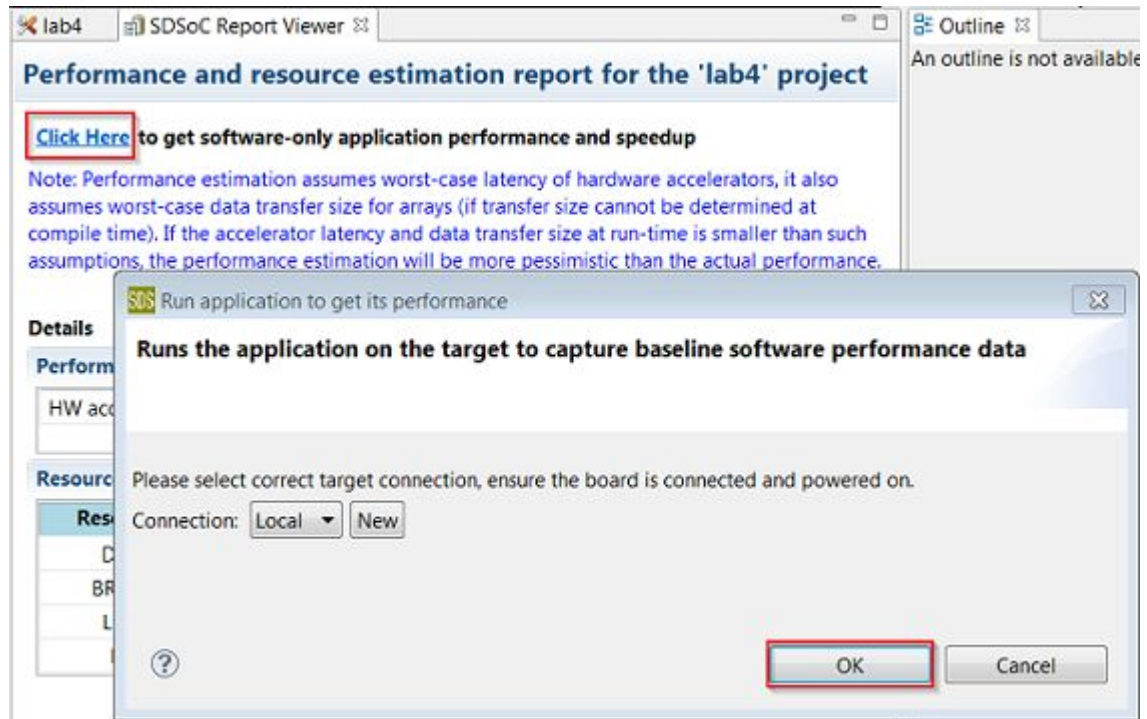


**IMPORTANT:** Ensure that the board is switched on before performing the instructions provided in this section.

To collect software run data and generate a performance estimation report:

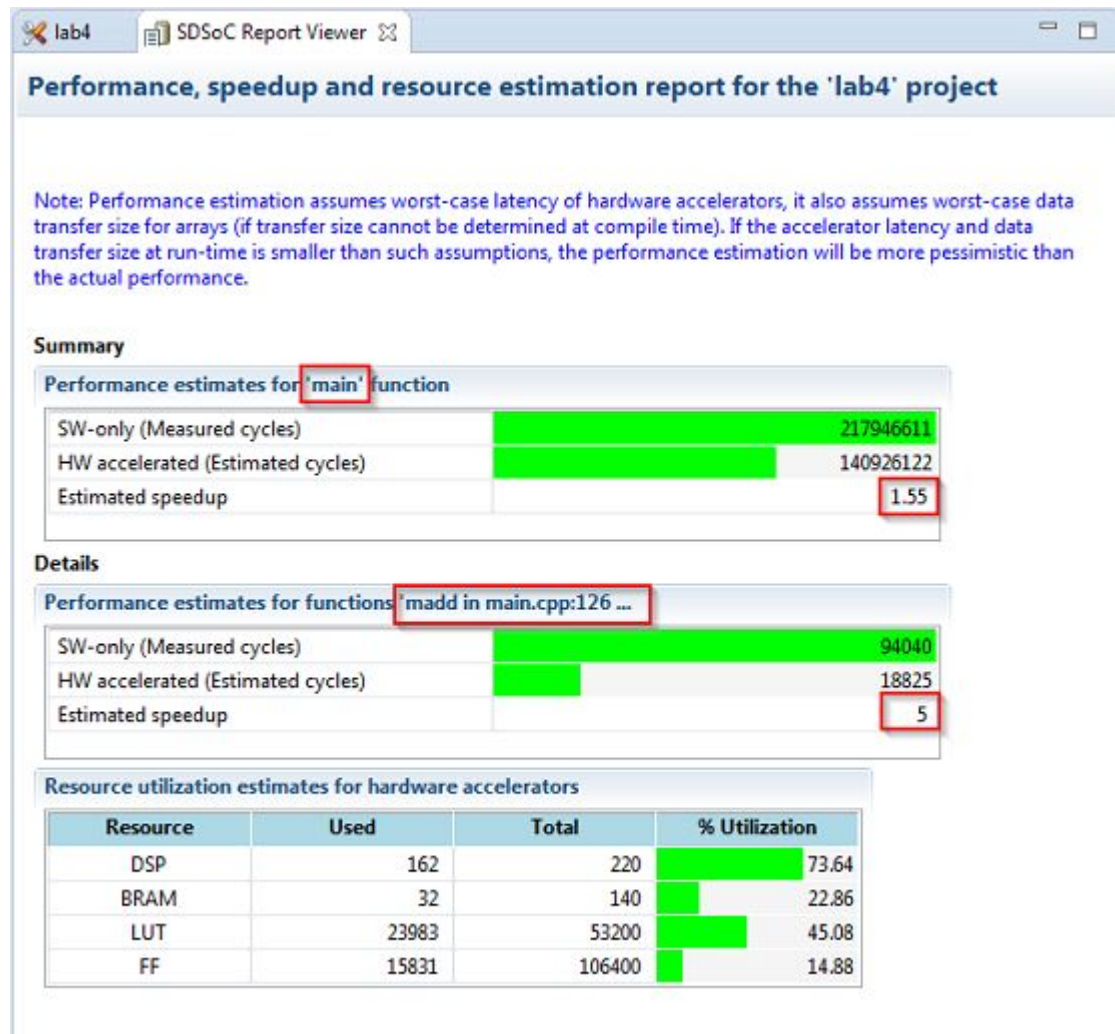
- Open the SDSoC Report Viewer.

2. Click the **Click Here** hyperlink on the viewer to launch the application on the board.  
The Run application to dialog box appears.
3. Select a pre-existing connection, or create a new connection to connect to the target board.



4. Click **OK**.

The debugger resets the system, programs and initializes the FPGA, runs a software-only version of the application. It then collects performance data and uses it to display the performance estimation report.

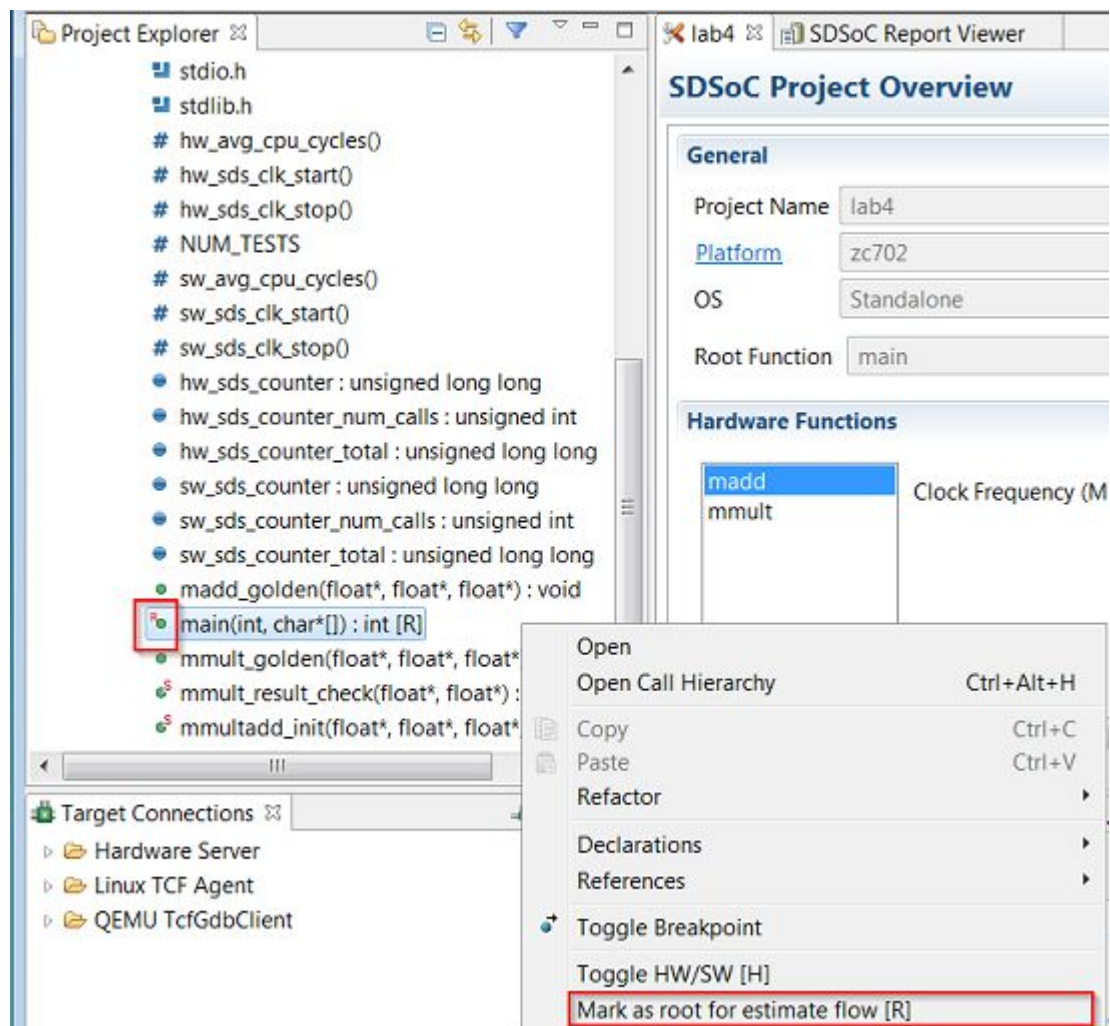


## Changing Scope of Overall Speedup Comparison

In the Performance Estimation Report, the first line shows the estimated speedup for the top-level function (referred to as perf root). This function is set to "main" by default. However, there might be code that you would like to exclude from this comparison, for example allocating buffers, initialization and setup. If you wish to see the overall speedup when considering some other function, you can do this by specifying a different function as the root for performance estimation flow. The flow works with the assumption that all functions selected for hardware acceleration are children of the root.

1. To change the top-level function used for estimating performance speedup (perf root), click on **SDSoC** in the upper right corner of the SDSoC IDE to return to the SDSoC perspective and in the **Project Explorer**, right-click the function that you are interested in selecting as root and click the menu item **Mark as root for estimate flow**.

A small R icon appears on the top left of that function listed as shown below. The selected function is a parent of the functions that are selected for hardware acceleration.



2. In the **Project Explorer**, right click on the project and select **Clean Project**, then **Build Project**. In the **SDSoC Project Overview**, click on **Estimate speedup** to generate the estimation report again and you get the overall speedup estimate based on the function that you selected.


## Additional Exercises

**NOTE:** Instructions provided in this section are optional.

You can learn how to use the performance estimation flow when Linux is used as the target OS for your application.

## Using the Performance Estimation Flow With Linux

To use the performance estimation flow with Linux:

1. Create a new project in the SDSoC™ environment (lab4\_linux) for the ZC702 platform and Linux OS using the design template for Matrix Multiplication and Addition.
2. Click on the tab labeled **lab4\_linux** (if the tab is not visible, in the **Project Explorer** tab under the **lab4\_linux** project double click on `project.sdsoc`). In the **Hardware Functions** panel, observe that the `madd` and `mmult` functions already appear in the list of functions marked for hardware – template projects in the SDSoC environment include information for automating the process of marking hardware functions.
3. If the **Hardware Functions** panel did not list any functions, you would click on the **Add Hardware Function** icon  to invoke a dialog for specifying hardware functions. Ctrl-click (press the Ctrl key and left click simultaneously) on the `madd` and `mmult` functions in the **Matching elements:** list, and notice that they appear in the **Qualified name and location:** list below. Click **OK**.
4. In the **SDSoC Project Overview** in the **Options** panel, check the **Estimate Performance** box. This enables the performance estimation flow for the current build configuration.
5. The **Build** icon provides a drop-down menu for selecting the build configuration and building the project. Clicking on the **Build** icon builds the project and with the **Estimate Performance** option checked, the performance estimation flow runs.

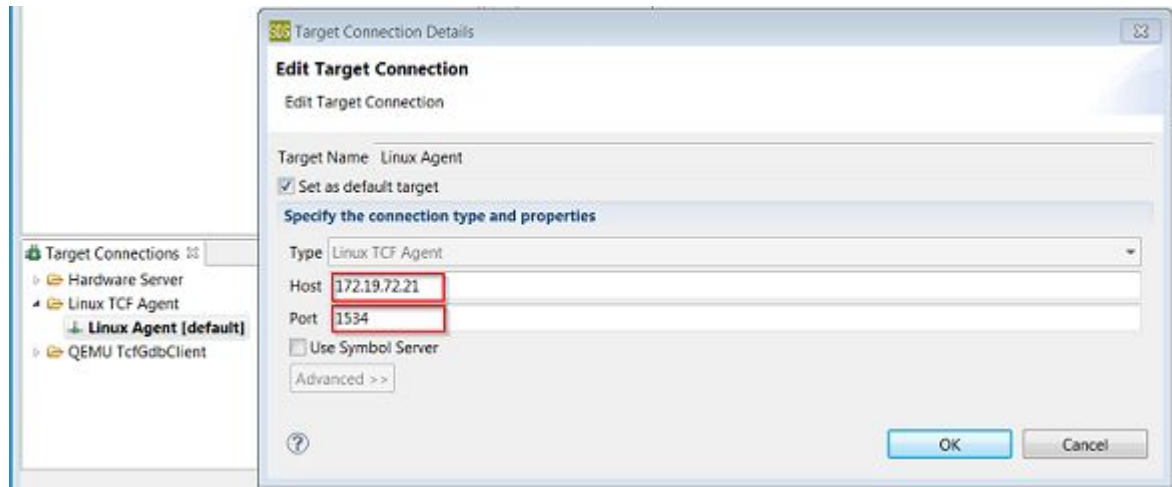
The SDSoC environment builds the project. A dialog box displaying the status of the build process appears.

6. Copy the contents of the `sd_card` folder under the build configuration to an sd card and boot up the board. Ensure that the board is connected to an Ethernet router using an Ethernet cable. Ensure that a serial terminal is connected.
7. Power on the board, and notice the Linux boot log displayed on the terminal. Look for a line that says `Sending select for 172.19.73.248...Lease of 172.19.73.248 obtained` or something similar, where the IP address assigned to your board is reported.

**NOTE:** This address is for use in the next step. If you miss this statement in the log as it scrolls by, you can obtain the IP address of the board by running the command `ifconfig`

8. Back in the SDSoC environment in the **Target Connections** view, expand **Linux TCF Agent** and right-click on **Linux Agent (default)**, then select **Edit**.

9. In the Target Connection Details dialog set up the IP address and port (1534) and click **OK**.



10. Open the **SDSoC Report Viewer**.
11. Click the **Click Here** hyperlink on the viewer to launch the application on the board.  
The Run application to dialog box appears.
12. Select the **Linux Agent** connection and click **OK**.

The SDSoC environment runs a software-only version of the application. It then collects performance data and uses it to display the performance estimation report.



## Tutorial: Task Pipelining Optimizations

This tutorial demonstrates how to modify your code to optimize the hardware-software system generated by the SDSoC environment using task-level pipelining. You can observe the impact of pipelining on performance.

**NOTE:** This tutorial is separated into steps, followed by general instructions and supplementary detailed steps allowing you to make choices based on your skill level as you progress through it. If you need help completing a general instruction, go to the detailed steps, or if you are ready, simply skip the step-by-step directions and move on to the next general instruction.

**NOTE:** You can complete this tutorial if you do not have a ZC702 board. When creating the SDSoC environment project, select your board. The tutorial instructions ask you to add source files created for an application created for the ZC702. If your board contains a smaller Zynq-7000 device, after adding source files you need to edit the file `mmult_accel.cpp` to reduce resource usage (in the accelerator source file you will see `#pragma HLS array partition` which sets `block factor=16`; instead, set `block factor=8`). A pre-built project is available only for the ZC702.

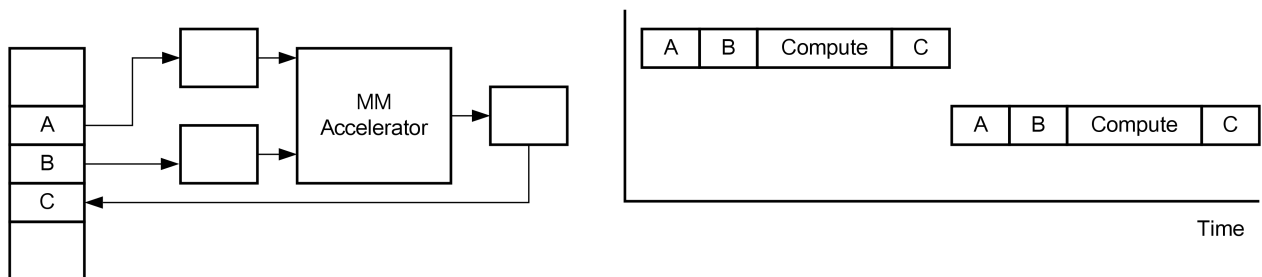
### Task Pipelining

If there are multiple calls to an accelerator in your application, then you can structure your application such that you can pipeline these calls and overlap the setup and data transfer with the accelerator computation. In the case of the matrix multiply application, the following events take place:

1. Matrices A and B are transferred from the main memory to accelerator local memories.
2. The accelerator executes.
3. The result, C, is transferred back from the accelerator to the main memory.

The following figure illustrates the matrix multiply design on the left side and on the right side a time-chart of these events for two successive calls that are executing sequentially.

**Figure 6–1: Sequential Execution of Matrix Multiply Calls**

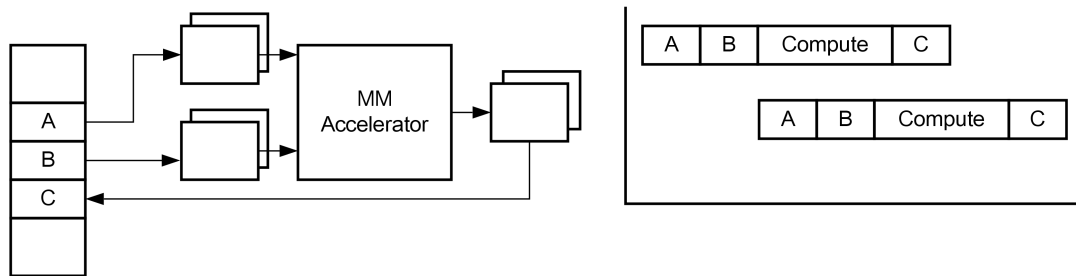


X14705\_060515



The following figure shows the two calls executing in a pipelined fashion. The data transfer for the second call starts as soon as the data transfer for the first call is finished and overlaps with the execution of the first call. To enable the pipelining, however, we need to provide extra local memory to store the second set of arguments while the accelerator is computing with the first set of arguments. The SDSoC environment generates these memories, called **multi-buffers**, under the guidance of the user.

**Figure 6–2: Pipelined Execution of Matrix Multiply Calls**



X14706\_060515

Specifying task level pipelining requires rewriting the calling code using the pragmas `async(id)` and `wait(id)`. The SDSoC environment includes an example that demonstrates the use of `async` pragmas and this Matrix Multiply Pipelined example is used in this tutorial.

## Learning Objectives

After you complete the tutorial, you should be able to:

- Use the SDSoC environment to optimize your application to reduce runtime by performing task-level pipelining.
- Observe the impact on performance of pipeline calls to an accelerator when overlapping accelerator computation with input and output communication.

## Task Pipelining in the Matrix Multiply Example

The SDSoC environment includes a matrix multiply pipelined example that demonstrates the use of `async` pragmas to implement task-level pipelining. This exercise allows you to see the runtime improvement that comes from using this technique.

1. Create a new SDSoC environment project (lab5) by selecting **File > New > SDSoC Project**. Enter the project name `lab5`, select the ZC702 **Platform** and Linux **OS**, and click **Next**.
2. The Templates page appears, containing source code examples for the selected platform. From the list of application templates, select **Empty Application** and click **Finish**.
3. Using your operating system file manager, navigate to `<path to install>/SDSoC/2016.2/samples/mmult_pipelined` and copy the source files in that directory (`mmult_accel.cpp`, `mmult_accel.h`, and `mmult.cpp`) into the `src` folder of the newly created project.
4. Change the build configuration to **SDRelease**.

5. Mark the function `mmult_accel` in the file `mmult_accel.cpp` for hardware using the **Add Hardware Function** icon in the **SDSoC Project Overview** or **Toggle HW/SW** in the **Project Explorer**.
6. Build the project.



---

**IMPORTANT:** *The build process might take approximately 30 to 45 minutes to complete. Instead of building the project you can save time and instead use the pre-built project. (To minimize disk usage in the SDSoC installation, the imported project might contain fewer files than a project you build, but it includes the files required to complete the tutorial.) To import a pre-built project: select **File > Import** and then select **General > Existing Projects into Workspace** and click **Next**. Click **Select archive file** and browse to find the `lab5_prebuilt.zip` file provided in the project files folder (`<path to install>/SDSoC/2016.2/docs/labs/lab5_prebuilt.zip`). Click **Open**. Click **Finish**.*

---

7. Copy the files obtained in the `sd_card` folder to an SD card, set up a terminal and run the generated application on the board. You need to specify the pipeline depth as an argument to the application. Run the application with pipeline depth of 1, 2 and 3 and note the performance obtained.

## Tutorial: Hardware/Software Event Tracing

This chapter provides step-by-step instructions to create a project, enable trace, run the application, and view the trace visualization. This tutorial assumes that the host PC is connected directly to the Zynq-7000 board, and that the board is a Xilinx ZC702 board. This tutorial is applicable to other boards and configurations. However, the details of the steps might differ slightly. The tutorial assumes you have already installed and started the SDSoC GUI and chosen a workspace.

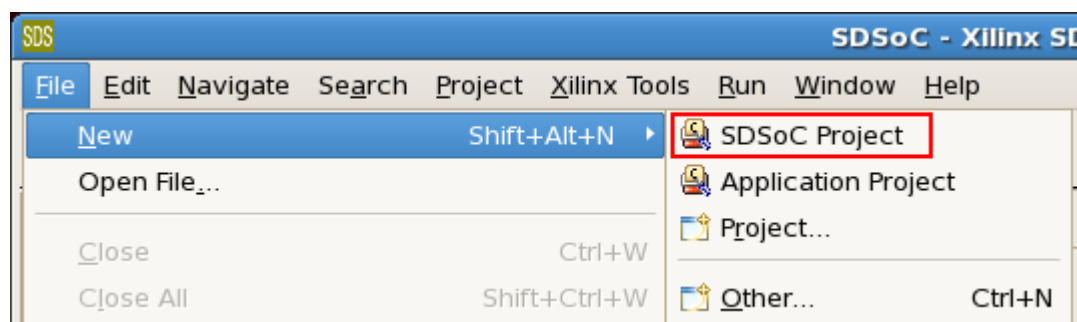
**NOTE:** This tutorial is separated into steps, followed by general instructions and supplementary detailed steps, allowing you to make choices based on your skill level as you progress through it. If you need help completing a general instruction, go to the detailed steps, or if you are ready, simply skip the step-by-step directions and move on to the next general instruction.

**NOTE:** You can complete this tutorial if you do not have a ZC702 board. When creating the SDSoC environment project, select your board and one of the available applications if the suggested template **Matrix Multiplication** is not found. For example, boards such as the MicroZed with smaller Zynq-7000 devices offer the **Matrix Multiplication (area reduced)** application as an available template. Any application can be used to learn the objectives of this tutorial. A pre-built project is only available for the ZC702 board.

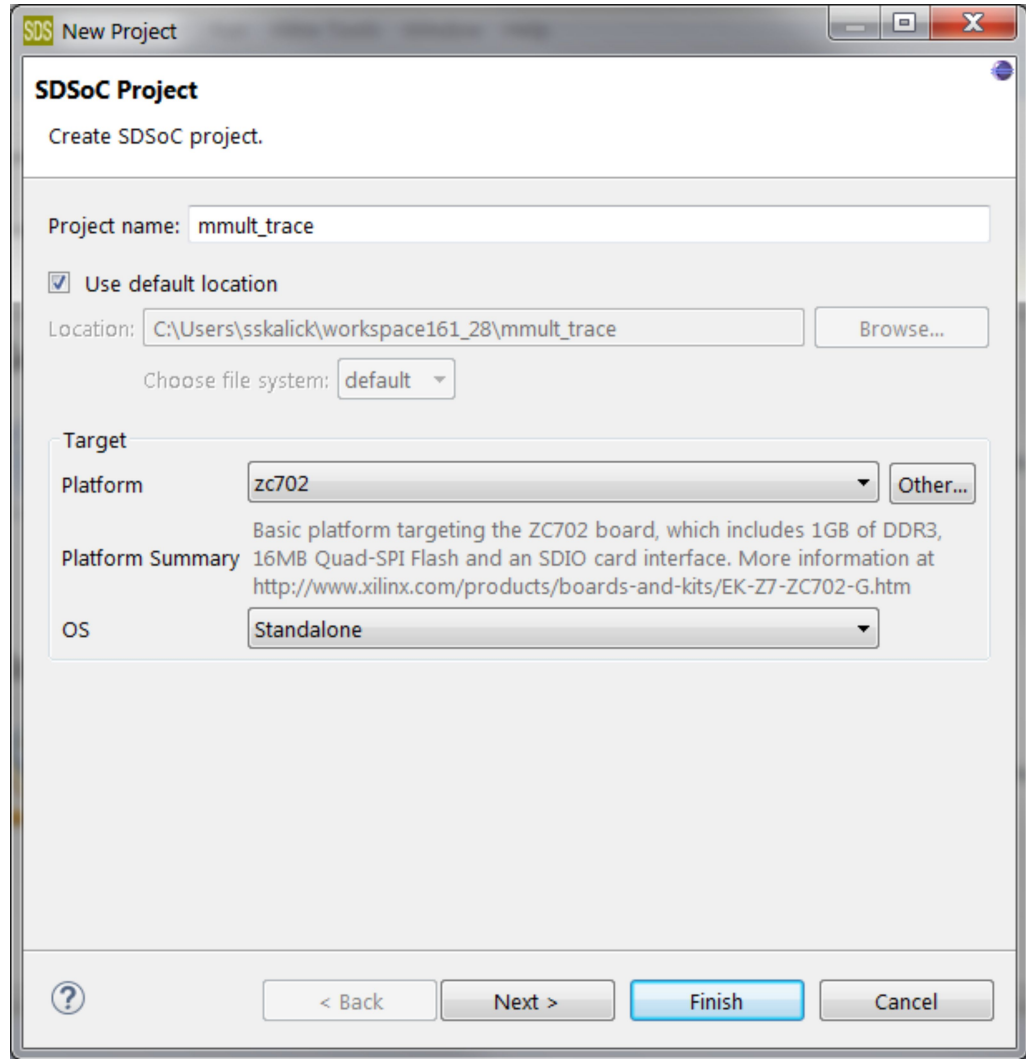
---

### Tracing a Standalone or Bare-Metal Project

1. Create a new project.
  - a. Select **File > New > SDSoC Project**.

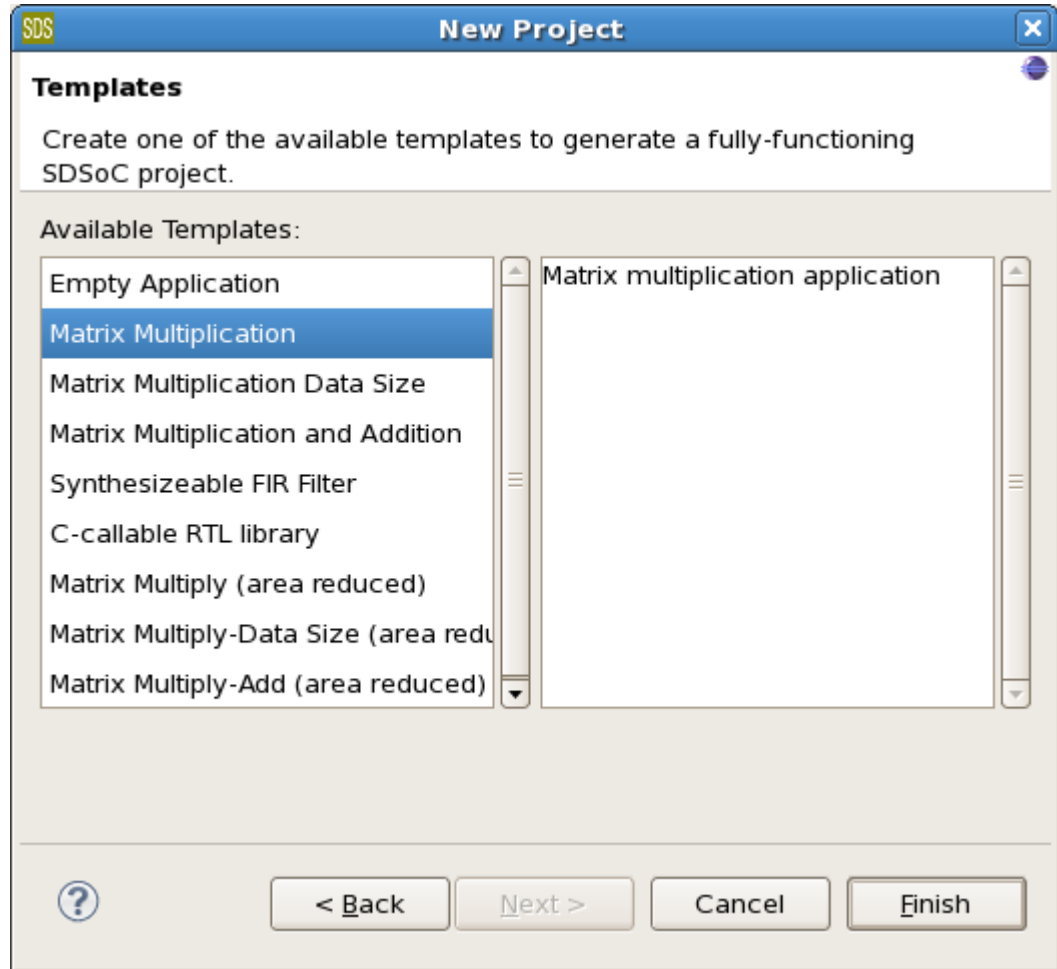



- b. In the New Project wizard, name the project `mmult_trace` and select **Standalone** as the OS. Select the appropriate platform if you are using something other than the ZC702 board.

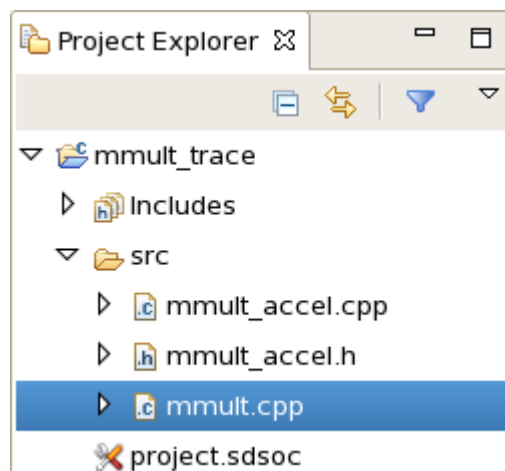


X16921-050316

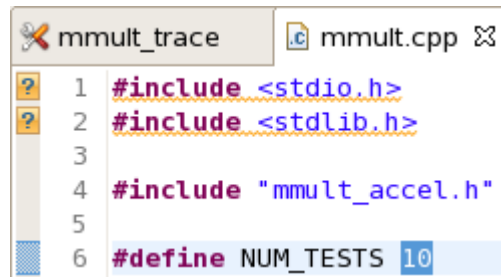
- c. Click **Next**.
- d. Select **Matrix Multiplication** as the template for this project and click **Finish**.



- e. In the Project Explorer, expand the various folders by clicking on the triangle , then open the `mmult.cpp` file.



- f. Change the number of tests symbol **NUM\_TESTS** from 1024 to **10**, then save and close the file.

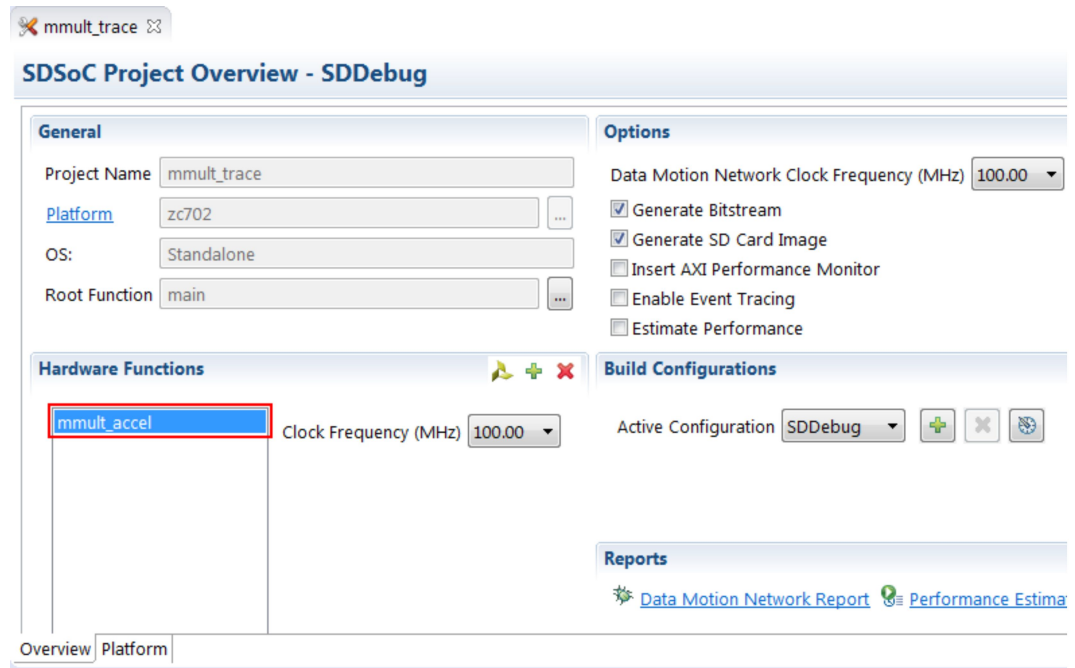


```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #include "mmult_accel.h"
5
6 #define NUM_TESTS 10

```

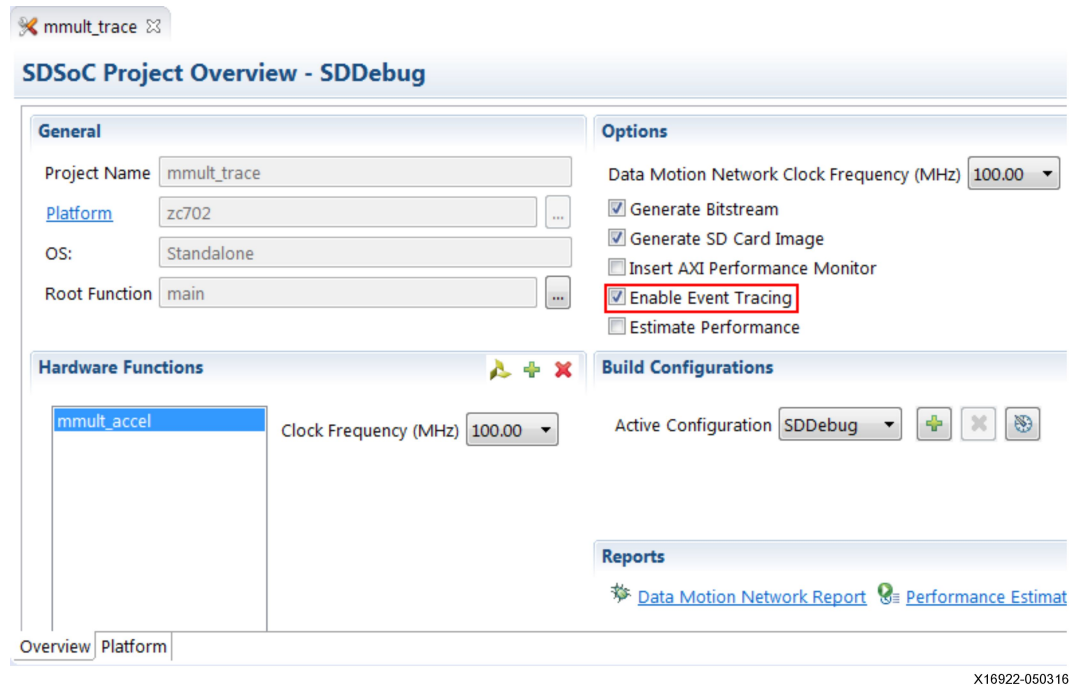
- g. In the SDSoC Project Overview (in the **mmult\_trace** tab), notice that **mmult\_accel** in the Hardware Functions section of the project overview is already marked for implementation in hardware.



X16928-050316

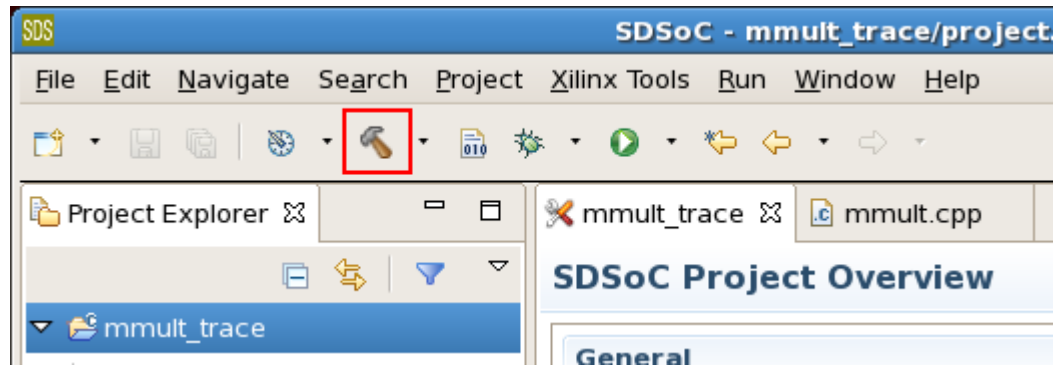


2. Configure the project to enable the Trace feature in the SDSoC environment.
  - a. In the Project Overview window, click the checkbox for **Enable Event Tracing**.



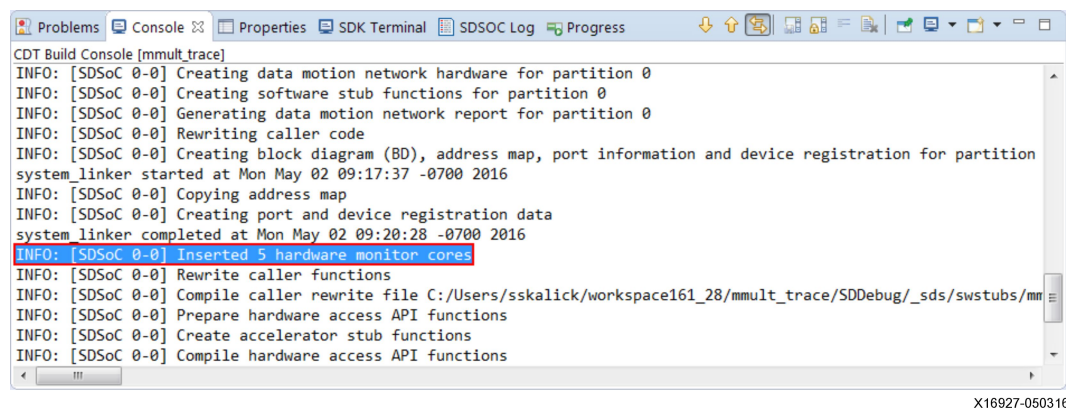
X16922-050316

3. Build the project.
  - a. Click the **Build** button to start building the project. (This will take a while.)



**IMPORTANT:** The build process might take approximately 30 to 45 minutes to complete. Instead of building the project you can save time and instead use the pre-built project. (To minimize disk usage in the SDSoc installation, the imported project might contain fewer files than a project you build, but it includes the files required to complete the tutorial.) To import a pre-built project: select **File > Import** and then select **General > Existing Projects into Workspace** and click **Next**. Click **Select archive file** and browse to find the `lab7a_mmult_trace.zip` file provided in the project files folder (`<path to install>/SDSoC/2016.2/docs/labs/lab7a_mmult_trace.zip`). Click **Open**. Click **Finish**.

After all the hardware functions are implemented in Vivado HLS, and after the Vivado IP Integrator design is created, you will see Inserted # hardware monitor cores displayed in the console. This message validates that the trace feature is enabled for your design and tells you how many hardware monitor cores have been inserted automatically for you.



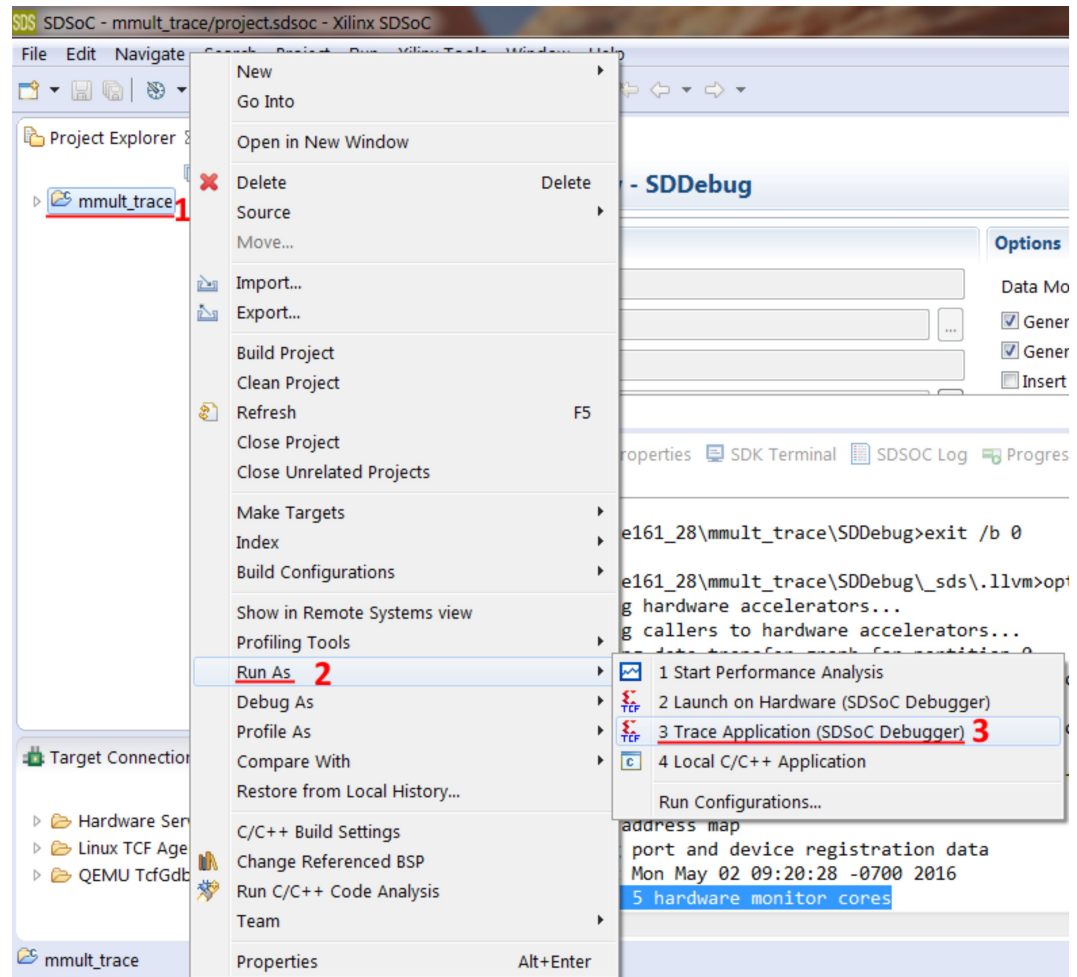
4. Run the application on the board.
  - a. When the build is finished, right-click on the project in the Project Explorer and select **Run As > Trace Application (SDSoC Debugger)**.

**NOTE:** Be sure not to select Debug As, because it will enable breakpoints. If your program breakpoints during execution, the timing will not be accurate (because the

software will stop, the hardware will continue running, and the trace timer used for timestamping will continue to run).

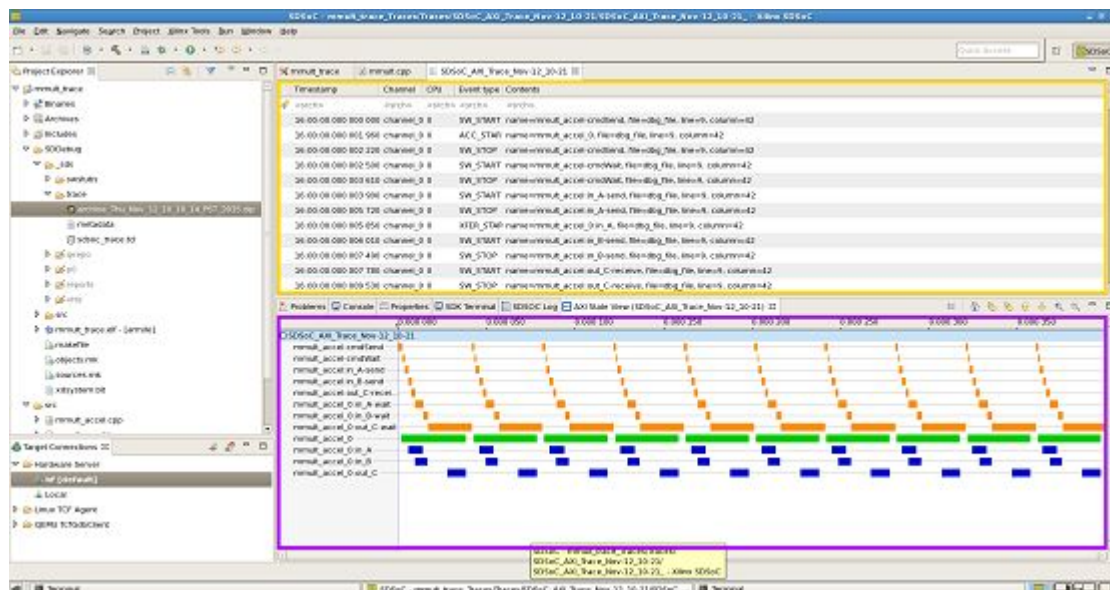
When you click on the **Trace Application (SDSoC Debugger)** option, the GUI downloads the bitstream to the board followed by the application ELF, starts the application, and then begins collecting the trace data produced until the application exits. After the application finishes (or any error in collecting the trace data occurs) the trace data collected is displayed.

**NOTE:** The application must exit successfully for trace data to be collected successfully. If the application does not exit normally (i.e., hangs in hardware or software, or the Linux kernel crashes), the trace data might not be collected correctly.

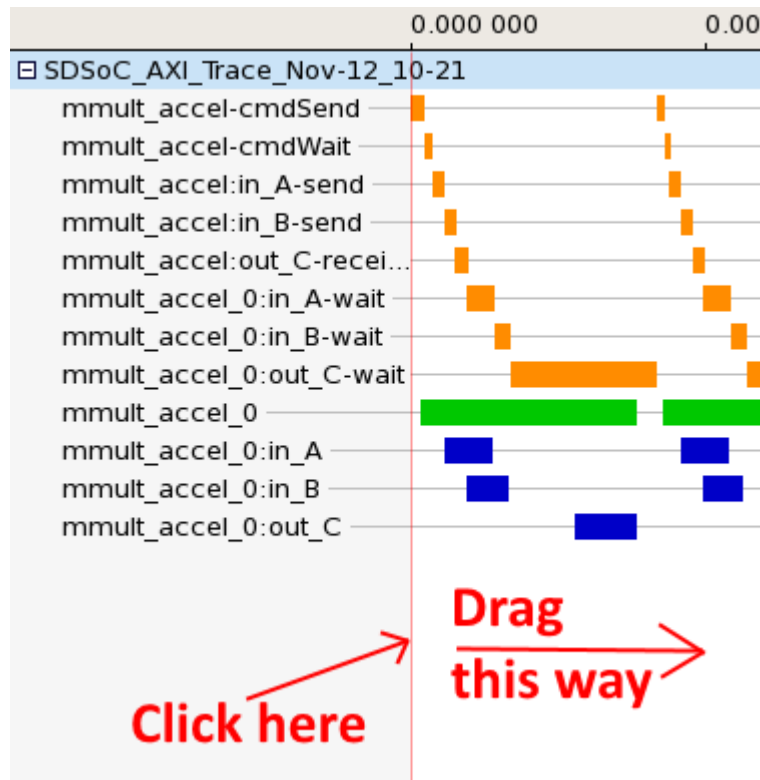


X16924-050316

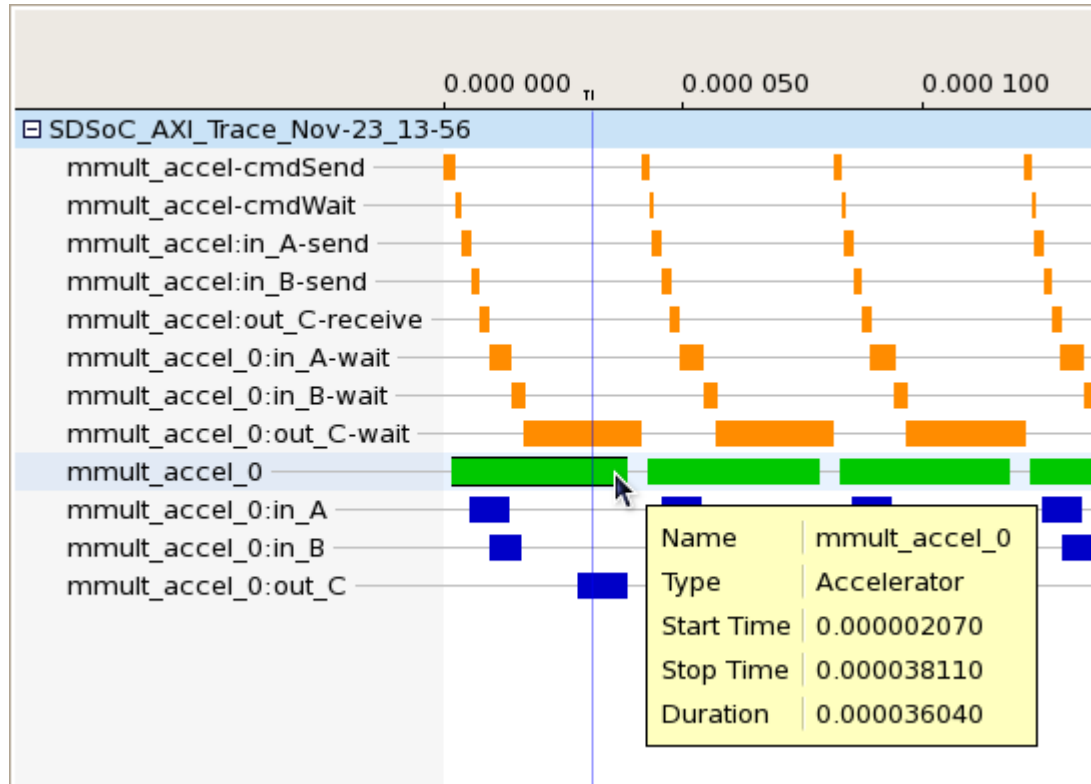
After the application exits, and all trace data is collected and displayed, you will see two main areas in the trace visualization: the event textual listing on top (yellow highlighted border), and the event timeline on the bottom (purple highlighted border). Both areas display the same information. The top textual listing orders event by time in a descending order. The bottom event timeline shows the multiple axes for each trace point in the design (either a monitor core or a region of software that is being traced).



- The first thing you should notice is that the 10 iterations of the application are clearly visible as repeated groups of events.
  - Orange events are software events.
  - Green events are accelerator events.
  - Blue events are data transfer events.
- b. If the names of the trace points in the event timeline are abbreviated with an ellipsis ("...") you can expand the panel by clicking on the border between the grey on the left and the white on the right (the border turns red when you hover the cursor over the right spot), and then clicking and dragging to the right.

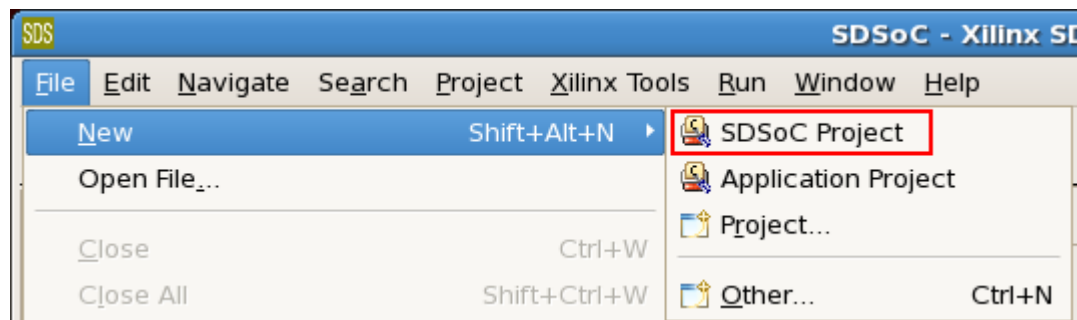


If you hover the cursor over one of the events, you will see a detailed tool-tip appear displaying the detailed information about each trace. The example below shows the first accelerator event, which corresponds to the start/stop of the `mmult_accel` function that we chose to implement in hardware (via Vivado HLS). The start time is at 0.000002070 seconds (2,070 ns) and the stop time is at 0.000038110 seconds (38,110 ns). It also shows the duration of the event (which is the runtime of the accelerator in this case) as 0.000036040 seconds (36,040 ns).



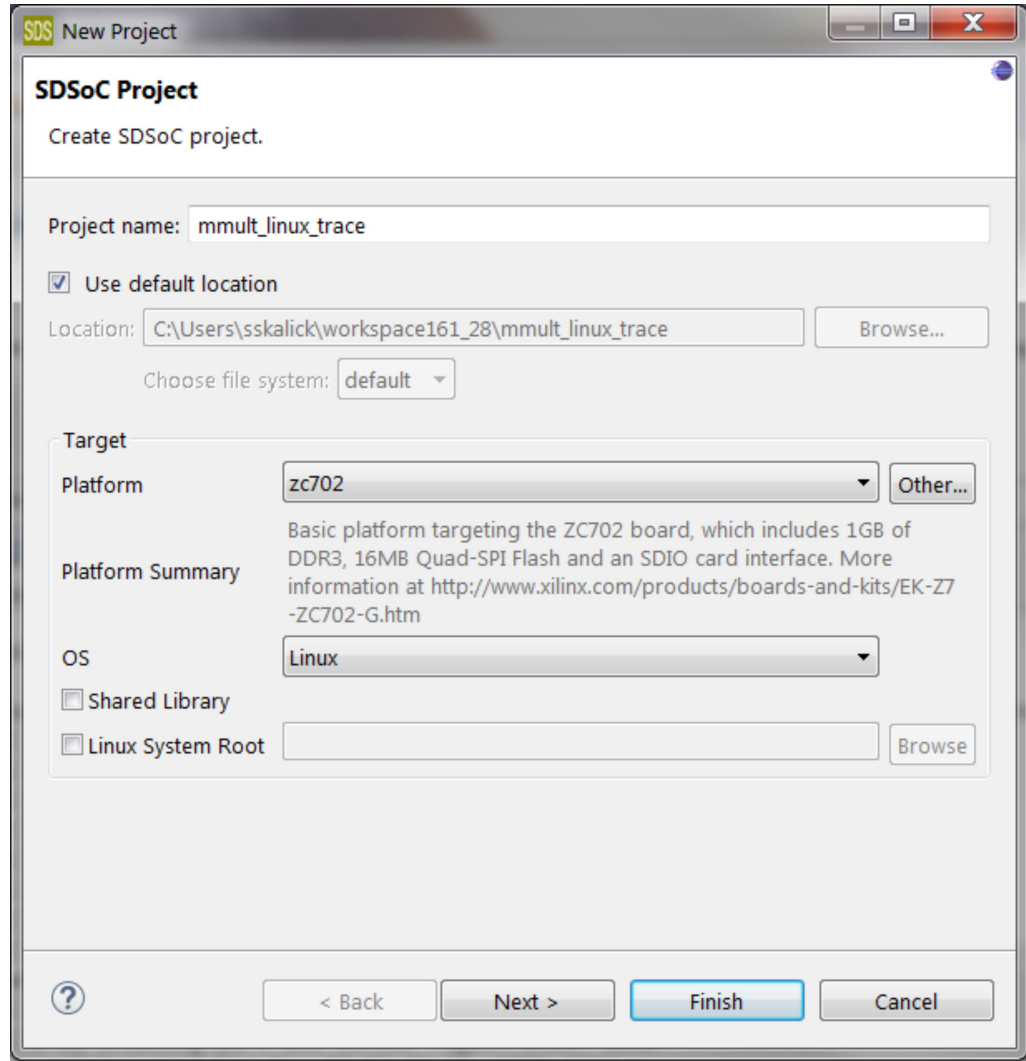
## Tracing a Linux Project

1. Create a new project.
  - a. Select **File > New > SDSoC Project**.



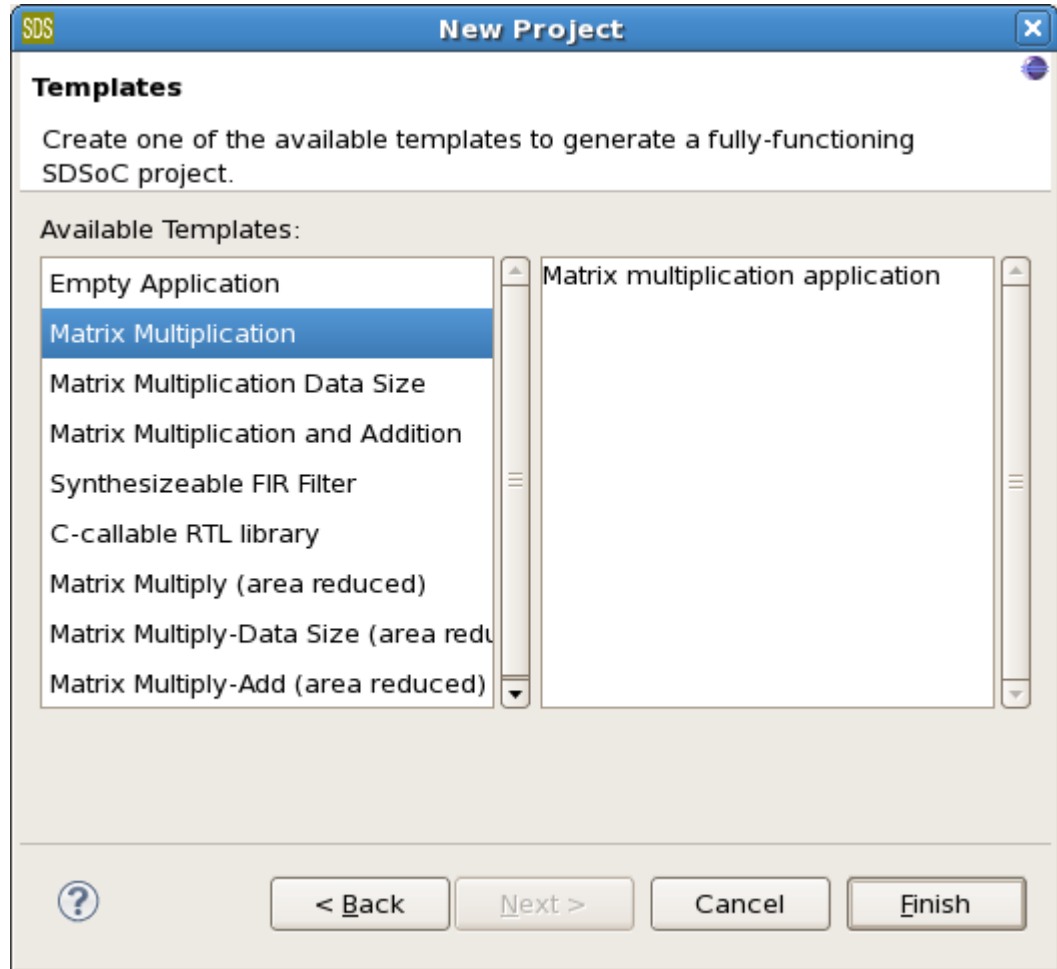
- b. In the New Project wizard, name the project `mmult_linux_trace` and select **Linux** as the OS. Select the appropriate platform if you are using something other than the ZC702 board.




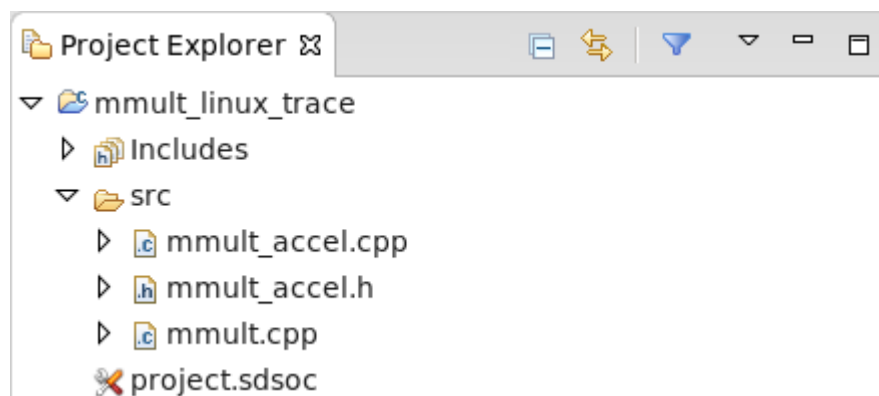


X16920-050316

- c. Click **Next**.
- d. Select **Matrix Multiplication** as the template for this project and click **Finish**.



- e. In the Project Explorer, expand the various folders by clicking on the triangle , then open the `mmult.cpp` file.



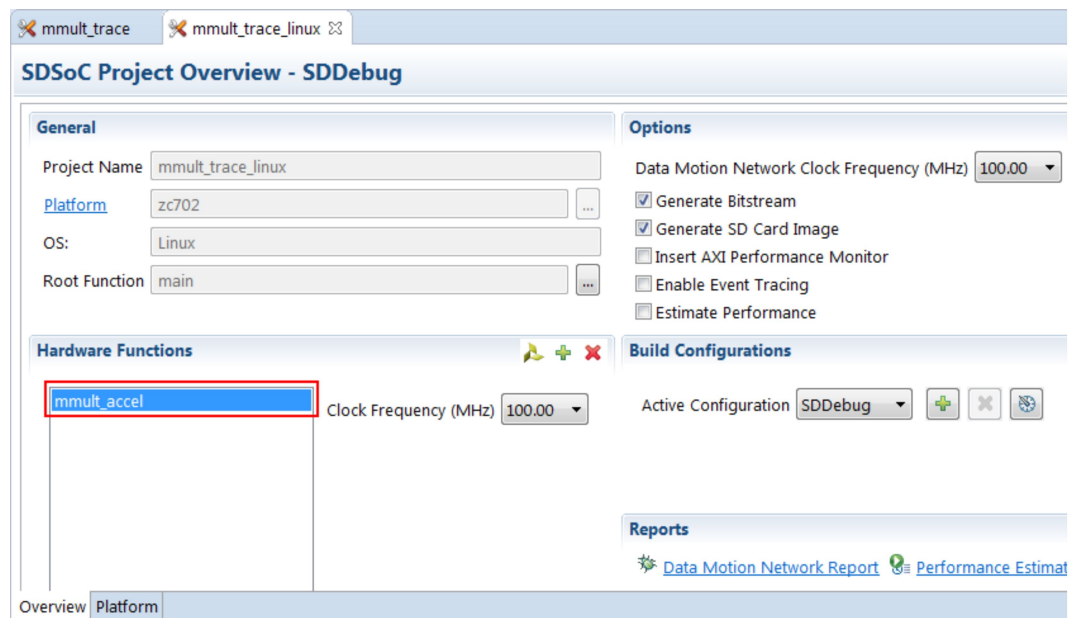
- f. Change the number of tests symbol **NUM\_TESTS** from 1024 to **10**, then save and close the file.

```
mmult_linux_trace  *mmult.cpp
#include <iostream>
#include <stdlib.h>
#include <stdint.h>

#include "sds_lib.h"
#include "mmult_accel.h"

#define NUM_TESTS 10
```

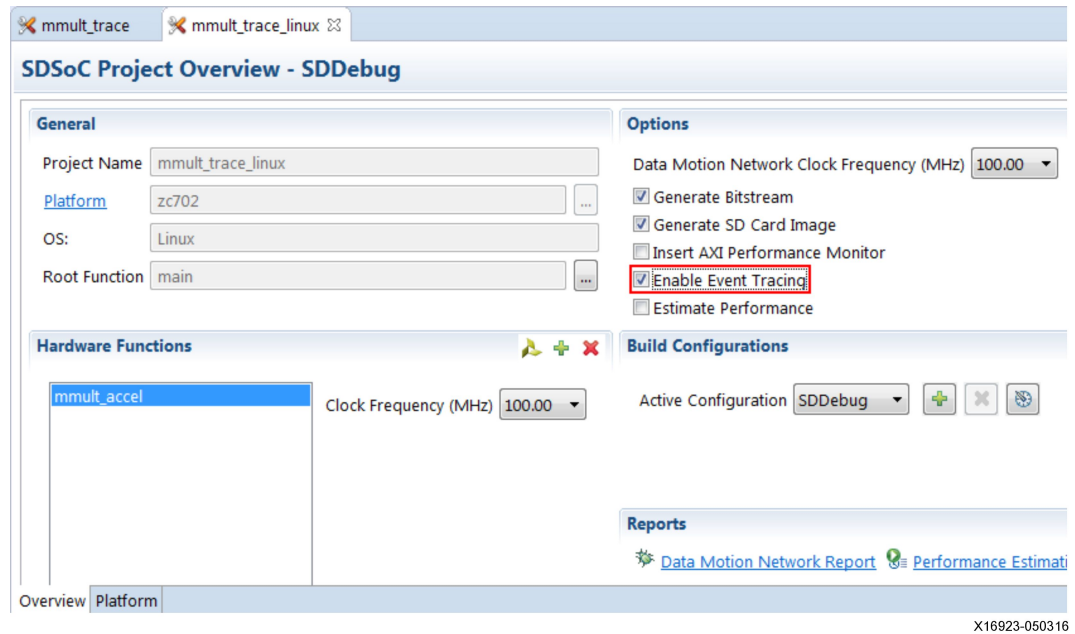
- g. In the SDSoC Project Overview (in the **mmult\_linux\_trace** tab), notice that the **mmult\_accel** in the Hardware Functions section of the project overview is already marked for implementation in hardware.



The screenshot shows the 'SDSoC Project Overview - SDDDebug' window. The 'General' tab is active, displaying project details for 'mmult\_trace\_linux'. The 'Platform' is 'zc702', 'OS' is 'Linux', and 'Root Function' is 'main'. The 'Hardware Functions' section shows 'mmult\_accel' selected, indicating it is marked for implementation in hardware. The 'Options' section includes checkboxes for 'Generate Bitstream', 'Generate SD Card Image', 'Insert AXI Performance Monitor', 'Enable Event Tracing', and 'Estimate Performance'. The 'Build Configurations' section shows 'Active Configuration' set to 'SDDDebug'. The 'Reports' section includes links for 'Data Motion Network Report' and 'Performance Estimat'.

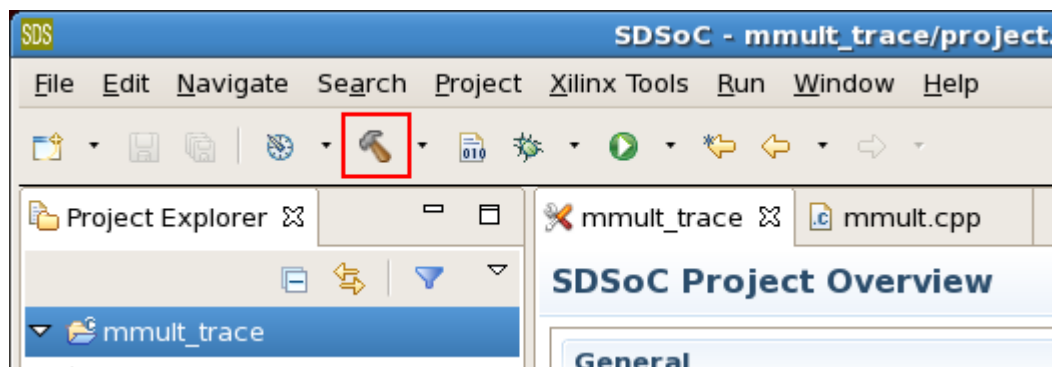
X16929-050316

2. Configure the project to enable the Trace feature in the SDSoC environment.
  - a. In the Project Overview window, click the checkbox for **Enable Event Tracing**.



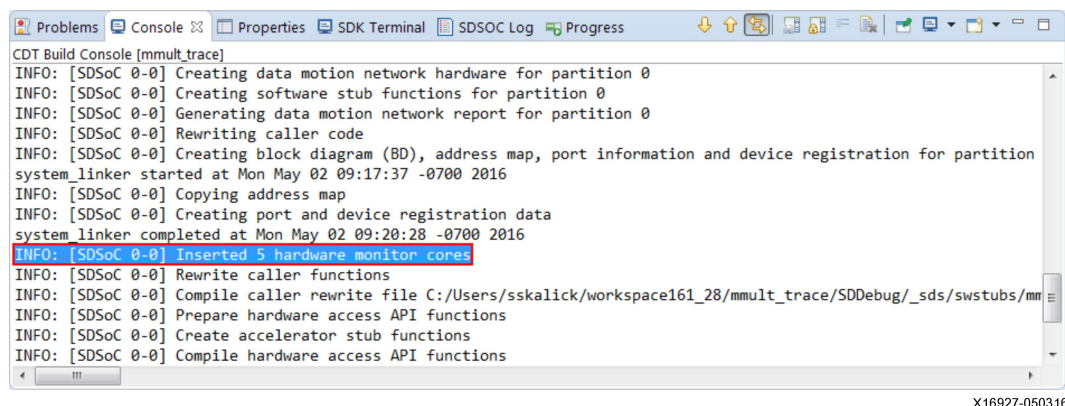
X16923-050316

3. Build the project.
  - a. Click the **Build** button to start building the project. (This will take a while.)



**IMPORTANT:** The build process might take approximately 30 to 45 minutes to complete. Instead of building the project you can save time and instead use the pre-built project. (To minimize disk usage in the SDSoC installation, the imported project might contain fewer files than a project you build, but it includes the files required to complete the tutorial.) To import a pre-built project: select **File > Import** and then select **General > Existing Projects into Workspace** and click **Next**. Click **Select archive file** and browse to find the `lab7b_mmult_trace_linux.zip` file provided in the project files folder (<path to install>/SDSoC/2016.2/docs/labs/lab7b\_mmult\_trace\_linux.zip). Click **Open**. Click **Finish**.

After all the hardware functions are implemented in the Vivado HLS, and after the Vivado IP Integrator design is created, you will see Inserted # hardware monitor cores displayed in the console. This message validates that the trace feature is enabled for your design and tells you how many hardware monitor cores have been inserted automatically for you.



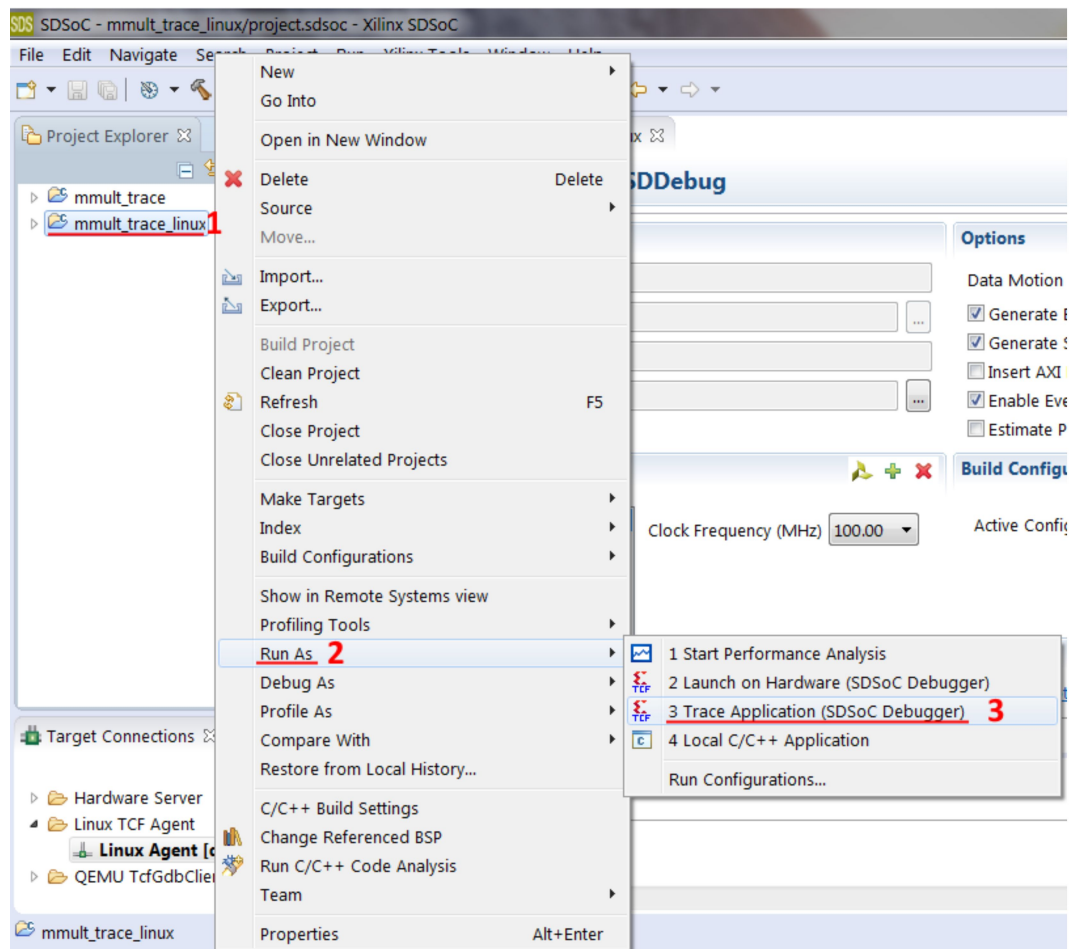
4. Run the application on the board.
  - a. When the build is finished, copy the files in the `sd_card` directory onto an SD card and insert into the SD card socket on the board.
  - b. Connect an Ethernet cable to the board (connected to your network, or directly to the PC).

- c. Connect the USB/UART port to the PC and open a serial console (TeraTerm or putty).
- d. Connect the USB/JTAG port to the PC and boot Linux on the board.
- e. From the **Target Connections** view, set up the **Linux TCL Agent** in the same manner as in [Tutorial: Debugging Your System](#).
- f. Right-click on the project in the Project Explorer and select **Run As > Trace Application (SDSoC Debugger)**.

**NOTE:** Be sure not to select Debug As, because it will enable breakpoints. If your program breakpoints during execution, the timing will not be accurate (because the software will stop, the hardware will continue running, and the trace timer used for timestamping will continue to run).

When you click on the **Trace Application (SDSoC Debugger)** option, the GUI downloads the ELF over the Ethernet TCF Agent connection, starts the application, and then begins collecting the trace data produced until the application exits. After the application finishes (or any error in collecting the trace data occurs) the trace data collected is displayed.

**NOTE:** The application must exit successfully for trace data to be collected successfully. If the application does not exit normally (i.e., hangs in hardware or software, or the Linux kernel crashes), the trace data might not be collected correctly.



X16925-050316



5. View the trace data.
  - a. After the application exits, all trace data is collected and displayed.

---

## Viewing Traces

1. After you have run the application and collected the trace data, an archive of the trace is created and stored in the build directory for that project in `<build_config>/_sds/trace`.
2. To open this trace archive, right click on it and select **Import and Open AXI Trace**.

The other files in the `_sds/trace` folder are `metadata` and `sdsoc_trace.tcl`. These files are produced during the build. They are used to extract the trace data and create the trace visualization archive. If you remove or change these files, you will not be able to collect the trace data and will need to perform a Clean and Build to regenerate them.

# Tutorial: Communication to Platform I/O Streams

This chapter describes how to interface with AXI-Stream interfaces coming from the platform within an application. This is an advanced tutorial demonstrating concepts that assume proficiency in using the SDSoC environment and Vivado® HLS as a prerequisite.

**NOTE:** This tutorial is separated into steps, followed by general instructions and supplementary detailed steps, allowing you to make choices based on your skill level as you progress through it. If you need help completing a general instruction, go to the detailed steps, or if you are ready, simply skip the step-by-step directions and move on to the next general instruction.


**NOTE:** You can complete this tutorial if you do not have a ZC702 board or a Zybo board (when the tutorial asks you to locate the platform, navigate to `<install_dir>/SDSoC/<version>/samples/xc7z010/zybo_axis_io`). If you have a different board, the platform described in the tutorial is not available, but you can follow the steps where possible (not running on the board) to satisfy a portion of the learning objectives. You can recreate the platform for your board by examining the existing platforms for other boards and using the technical references described in the tutorial.

---

## Advanced Concepts: AXI4-Stream to Memory

This section describes how to take data coming from an AXI4-Stream interface in the platform, and write that data directly into memory. This example demonstrates how to use the `zero_copy` datamover which employs the Vivado HLS capability to generate an AXI-Memory Mapped (AXIMM) master interface in the accelerator to move data directly to DDR memory. This example uses the `zc702_axis_io` platform referenced in [SDSoC Environment User Guide: Platforms and Libraries \(UG1146\)](#). Because this platform delivers a data stream over the AXI4-Stream transport interface, the hardware function in the application is configured to have an AXI4-Stream slave interface and write the data to an AXIMM interface that masters a bus to DDR.

1. Create a new project.
  - a. Select **File > New > SDSoC Project**.
  - b. In the New Project wizard, name the project `stream2mem` and select **Standalone** as the OS. Click the **Other** button to select a custom platform and navigate to the `zc702_axis_io` platform located in `<install_dir>/SDSoC/<version>/samples/platforms/zc702_axis_io`.
  - c. Click **Next**.
  - d. Select **Unpacketized AXI4-Stream to DDR** as the template for this project and click **Finish**.

- e. In the Project Explorer, expand the various folders by clicking on the triangle , then open the `main.cpp` file.

This example is configurable for different buffer sizes (using the `BUF_SIZE` symbol) and number of buffers (using the `NUM_BUFFERS` symbol).

```
#include <stdio.h>
#include "sds_lib.h"
#include "zc702_axis_io.h"
```

The `s2mm_data_copy` ("stream to memory-mapped data copy") function has two SDSoC pragmas: `data mem_attribute`, which specifies that the `buf` argument is cacheable and physically contiguous (because it will be allocated with `sds_alloc`), and `data zero_copy`, which specifies that the `buf` argument should be implemented as an AXI memory-mapped interface. The Vivado HLS pragma `interface axis` is used to specify the `fifo` argument as an AXI4-Stream. In general, you do not need to write HLS interface pragmas, but in this case, because the hardware function is customizing AXI transport signaling, the use of HLS pragmas is required. (By default, AXI4-Streams are packetized in the SDSoC environment whereas HLS by default treats them as unpacked.) The copy loop is pipelined using the Vivado HLS pragma `pipeline` which tries to achieve `II=1` for the loop (i.e., copy a data element every clock cycle).

```
// s2mm "DMA" accelerator
#pragma SDS data mem_attribute (buf:CACHEABLE|PHYSICAL_CONTIGUOUS)
#pragma SDS data zero_copy(buf)
int s2mm_data_copy(unsigned *fifo, unsigned buf[BUF_SIZE])
{
    #pragma HLS interface axis port=fifo
    for(int i=0; i<BUF_SIZE; i++) {
        #pragma HLS pipeline
        buf[i] = *fifo;
    }
    return 0;
}
```

The `s2mm_data_copy` function is called within the `s2mm_data_copy_wrapper` function, which defines the hardware/software interface through its data flow. The `pf_read_stream` function is a platform function that is the source of the input AXI4-Stream data. The `rbuf0` variable is "written" by the platform function `pf_read_stream` and "read" by the `s2mm_data_copy` function. This def-use instance is parsed by the `sdscc` compiler which determines that `pf_read_stream` is a platform function and `s2mm_data_copy` is a hardware function and therefore `rbuf0` is a direct connection between two hardware IP cores. As such, the length of the array `rbuf0` is inconsequential and in reality the software variable is never actually used (because data goes straight from one hardware IP to another without going through software).

```
// 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);
}
```

In the SDSoC Project Overview (in the **stream2mem** tab), `s2mm_data_copy` in the Hardware Functions section of the project overview is already marked for implementation in hardware.

2. Build the project.
  - a. Click the **Build** button to start building the project. (This will take a while.)
  - b. After the build finishes, the `s2mm_data_copy` function is implemented in hardware and directly connected to the platform AXI4-Stream for its input interface, and the ACP port for its output interface. The data motion report shows this information.

### Data Motion Network


Argument	IP Port	Connection
<code>rbuf</code>	<code>stream_fifo_0_M_AXIS</code>	<code>s2mm_data_copy_0:fifo</code>
<code>fifo</code>	<code>fifo</code>	<code>zc702_axis_io_0:stream_fifo_0_M_AXIS</code>
<code>buf</code>	<code>buf_r</code>	<code>ps7_S_AXI_ACP:AXIMM:0x80</code>
<code>return</code>	<code>ap_return</code>	<code>ps7_M_AXI_GP0:AXILITE:0xC0</code>

3. Run the application on the board.
  - a. When the build is finished, right-click the project in the Project Explorer and select **Run As > Launch on Hardware (SDSoC Debugger)**.
  - b. The output on the console from the board should print `TEST PASSED` which indicates that the application ran successfully and all buffers were filled with sequential data samples without any data drops or bubbles.

## Advanced Concepts: AXI4-Stream to Accelerator

This section describes how to consume an unpackitized data stream from a platform AXI4-Stream interface (i.e., a stream without the TLAST signal) in an application within the SDSoC environment. In practice, a stream of data might be continuous and have no logical grouping into packets, e.g., data produced by an analog-to-digital converter. However, within the SDSoC environment, streams are packetized by default, in part to comply with the requirements of some of the underlying IP cores like the AXI DMA. The application in this section demonstrates how to packetize an incoming stream so that it can be consumed by functions or data movers requiring a packetized stream input. As was the case in the previous example, some of the hardware functions manipulate arguments at the AXI transport level, which requires the use of explicit HLS interface pragmas and transport level coding. While this is not recommended in general when you are working in the SDSoC environment, it is useful when interfacing to external I/O. This example uses the `zc702_axis_io` platform referenced in [SDSoC Environment User Guide: Platforms and Libraries \(UG1146\)](#).

1. Create a new project.
  - a. Select **File > New > SDSoC Project**.

- b. In the New Project wizard, name the project `stream2acc` and select **Standalone** as the OS. Click the **Other** button to select a custom platform and navigate to the `zc702_axis_io` platform located in `<install_dir>/SDSoC/<version>/samples/platforms/zc702_axis_io`.
- c. Click **Next**.
- d. Select **Unpacketized AXI-Stream Converter** as the template for this project and click **Finish**.
- e. In the Project Explorer, expand the various folders by clicking on the triangle , then open the `main.cpp` file.

In the `main()` function, `pf_read_stream` is the platform function that produces the initial data stream and the `packetize` function consumes the raw stream `rbuf` and produces the packetized stream `pbuf`. This packetized stream is then consumed by the `minmax` function, which is the hardware function destined to actually compute on the input data stream.

```
int main() {
    unsigned min,max;
    //buffers in hardware
    unsigned rbuf[1];
    unsigned pbuf[1];
    //read from platform function
    pf_read_count(rbuf);
    //call hardware packetize function to convert unpacketized stream
    packetize(rbuf,pbuf);
    //call hardware function to process the data
    minmax(pbuf,&min,&max);

    printf("min value is %u and max is %u in packet of %u elements\n\r",min,max,BUF_SIZE);
    return 0;
}
```

The `packetize` and `minmax` functions, once marked for acceleration using Vivado HLS, become directly connected. This results in a processing pipeline with a data source from the platform, through the `packetize` function (which produces the TLAST signal), and finally consumed by the `minmax` function. If the original AXI4-Stream source in the platform were packetized (producing the TLAST signal), the `packetize` function would not be needed and the `minmax` function would consume `rbuf` as its input.

- f. Open the `packetize.cpp` file. In the `#ifdef __SDSVHLS__` at the top of the file, the `__SDSVHLS__` symbol is defined by `sdsc` only when running through Vivado HLS, so the code that is specific to the HLS process is not compiled by the ARM compiler. The `#else` block in the code below is a function that is representative and is compiled by ARM GCC. The `ap_axiu` struct defines the side-channel signals in the AXI4-Stream interface. These signals are set in the function body. This function is defined with a static variable `cnt` that keeps count of the number of elements read out the B interface. The TLAST signal is set when the count reaches the size of the buffer. This function sets `axis` interface pragma for both arguments, in addition to the `ap_ctrl_none` pragma to make the resulting core run all the time

(withough being started). SDSoC supports accelerators with no control interface as long as all inputs and outputs are streams.

```
#ifndef __SDSVHLS__

/*
 * Packetize an axis stream by adding TLAST sideband signal
 * @param A an unpacktize axis stream
 * @param B a packetized stream of size BUF_SIZE
 */
void packetize(unsigned *A, ap_axiu<32,1,1,1> *B)
{
#pragma HLS pipeline enable_flush
#pragma HLS interface axis port=A
#pragma HLS interface axis port=B
#pragma HLS interface ap_ctrl_none port=return

    static unsigned cnt = 0;
    if (cnt == BUF_SIZE)
        cnt = 0;
    B->data = *A;
    B->strb = 0xF;
    B->keep = 0xF;
    B->user = 0;
    B->last = (++cnt == BUF_SIZE) ? (ap_uint<1>)1 : (ap_uint<1>)0;
    B->id = 0;
    B->dest = 0;
}

#else

void packetize(unsigned *A, unsigned *B) {
    for(int i=0; i<BUF_SIZE; i++)
        B[i] = A[i];
}

#endif
```

2. Build the project.
  - a. Click the **Build** button to start building the project. (This will take a while.)
  - b. After the build finishes, the `packetize` function is implemented in hardware and directly connected to the platform AXI4-Stream for its input interface, and connected to the `minmax` function directly. The output from the `minmax` function is a set of scalars that are accessed over the AXI4-Lite interface. The Data Motion Report shows this information.

**Data Motion Network**

Accelerator	Argument	IP Port	Connection
<code>minmax_0</code>	<code>data</code>	<code>data</code>	<code>packetize_0:B</code>
	<code>min</code>	<code>min</code>	<code>ps7_M_AXI_GP0:AXILITE:0xC0</code>
	<code>max</code>	<code>max</code>	<code>ps7_M_AXI_GP0:AXILITE:0xC4</code>
<code>packetize_0</code>	<code>A</code>	<code>A</code>	<code>zc702_axis_io_0:stream_fifo_0_M_AXIS</code>
	<code>B</code>	<code>B</code>	<code>minmax_0:data</code>
<code>zc702_axis_io_0</code>	<code>rbuf</code>	<code>stream_fifo_0_M_AXIS</code>	<code>packetize_0:A</code>


3. Run the application on the board.
  - a. When the build is finished, right-click the project in the Project Explorer and select **Run As > Launch on Hardware (SDSoC Debugger)**.
  - b. The output on the console from the board should print the minimum value, the maximum value, and the size of the buffer, which indicates that the application ran successfully.

## Advanced Concepts: Lossless Data Capture

This section describes how to capture data that streams from a platform port over an AXI4-Stream transport channel, and store it in DDR memory without dropping any data, using an application developed in the SDSoC environment. In practice, a stream of data might be continuous and need to be processed at a different rate than the rate at which it arrives. This rate matching requires buffering, either in hardware, in DDR memory, or most commonly, both. Often the underlying IP cores that move data introduce task setup or teardown latencies that must be masked through buffering. In this section, the application demonstrates software mechanisms that enable lossless data capture from an AXI4-Stream into storage in DDR memory. This example uses the `zc702_axis_io` platform referenced in [SDSoC Environment User Guide: Platforms and Libraries \(UG1146\)](#).

1. Create a new project.
  - a. Select **File > New > SDSoC Project**.



- b. In the New Project wizard, name the project `capture` and select your preferred OS (Standalone or Linux). Click the **Other** button to select a custom platform and navigate to the `zc702_axis_io` platform located in `<install_dir>/SDSoC/<version>/samples/platforms/zc702_axis_io`.
- c. Click **Next**.
- d. Select **Lossless data capture from AXI4-Stream to DDR** as the template for this project and click **Finish**.
- e. In the Project Explorer, expand the various folders by clicking on the triangle , then open the `main.cpp` file.

In the `main()` function, `pf_read_stream` is the platform function that produces the initial data stream and the `PullPacket` function consumes the raw stream `fifoLink` and writes into the buffer referenced by `p->buffer`. This buffer actually exists in DDR memory, and the `PullPacket` accelerator writes directly into DDR memory using an AXI4-MemoryMapped master interface without the use of an AXI DMA.

```
//enqueue reads
while(pendingReads < MAX_PENDING)
{
    if(emptyBuffers.empty())
    {
        std::cout << "Ran out of buffers, aborting" << std::endl;
        exit(-1);
    }
    Packet* p = emptyBuffers.front();
    emptyBuffers.pop();
    pf_read_stream(fifoLink);
#pragma SDS_async(1)
    PullPacket(fifoLink, p->buffer, p->bufSize, p->validSize);
    fillingBuffers.push(p);
    pendingReads++;
}
```

In the *enqueue reads* loop, the `#pragma SDS async(1)` syntax in the code snippet above denotes that the `PullPacket` function is to be started from software, but should return immediately—without waiting for the accelerator to finish.

```
while(packetsProcessed < iterations)
{
    //Pull finished reads
    while(pendingReads && sds_try_wait(1))
    {
        assert(!fillingBuffers.empty());
        Packet* p = fillingBuffers.front();
        fillingBuffers.pop();
        fullBuffers.push(p);
        pendingReads--;
    }
    //enqueue reads
    while(pendingReads < MAX_PENDING)
    {
        ... //shown above
    }
    //check finished processing
    while(!fullBuffers.empty())
    {
        Packet* p = fullBuffers.front();
        assert(p->validSize == p->bufSize);
        fullBuffers.pop();
        IP_ELEM_TYPE first = p->buffer[0];
        IP_ELEM_TYPE last = p->buffer[p->validSize - 1];
        if(last - first != p->validSize - 1)
        {
            std::cout << "----Bubble detected internal to packet "
                        << packetsProcessed << ", values " << first
                        << "-" << last << std::endl;
        }
        if(lastPacketEnding > 0 && lastPacketEnding + 1 != first)
        {
            std::cout << "----Bubble detected before packet "
                        << packetsProcessed << ", values "
                        << lastPacketEnding << "-" << first << std::endl;
        }
        lastPacketEnding = last;
        emptyBuffers.push(p);
        packetsProcessed++;
    }
}
```

After using the `async` pragma to enqueue several accelerator invocations, the *pull finished reads* loop handles the case when an accelerator is done and adds the buffer to a list for validation. This is done by using the `sds_try_wait()` function which return true (1) if the accelerator has finished its execution, or false (0) if the accelerator has not yet finished. If the accelerator is not done yet, execution in software continues to another loop. In the *check finished processing* loop, the buffers that are done are checked to see if any data was dropped, and if so prints an error message. Execution continues in the main loop checking to see if more instances of the accelerator execution can be enqueued for processing.

2. Build the project.
  - a. Click the **Build** button to start building the project. (This will take a while.)
  - b. After the build finishes, the `PullPacket` function is implemented in hardware and directly connected to the platform AXI4-Stream for its input interface, and the ACP port for its output interface. The data motion report shows this information.

### Data Motion Network

Accelerator	Argument	IP Port	Connection
PullPacket_0	input	input_r	zc702_axis_io_0:stream_fifo_M_AXIS
	outBuf	outBuf	ps7_S_AXI_ACP:AXIMM:0x80
	bufSize	bufSize	ps7_M_AXI_GP0:AXILITE:0x84
	validSize	validSize	ps7_M_AXI_GP0:AXILITE:0xC0
zc702_axis_io_0	rbuf	stream_fifo_M_AXIS	PullPacket_0:input_r

3. Run the application on the board.
  - a. When the build is finished, right-click the project in the Project Explorer and select **Run As > Launch on Hardware (SDSoC Debugger)**.
  - b. The output on the console from the board should print the minimum value, the maximum value, and the size of the buffer, which indicates that the application ran successfully.

# Troubleshooting

If you encounter issues using the SDSoC™ environment after installation, consult this section for potential issues and their resolution.

---

## Path Names Too Long

When using Vivado tools on a Windows platform, path names longer than 260 characters may result in the error message:

```
ERROR: [Common 17-143] Path length exceeds 260-Byte maximum allowed  
by Windows: <LongPathToFile>.
```

Possible solutions to shorten path lengths or to avoid this are described in [Answer Record AR# 52787](#). For example, use shorter path names, map a new drive letter to a lower directory in the path, and other methods.

---

## Use Correct Tool Scripts

In each shell used to run the SDSoC™ environment, use only the environment setup scripts corresponding to the Xilinx tool releases or recommended PATH environment settings.

Running Xilinx design tool environment setup scripts from other or additional releases in the same shell will result in incorrect SDSoC environment behaviors or results.

# 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. *SDSoC Environment User Guide: An Introduction to the SDSoC Environment* ([UG1028](#)), also available in the docs folder of the SDSoC environment.
2. *SDSoC Environment User Guide* ([UG1027](#)), also available in the docs folder of the SDSoC environment.
3. *SDSoC Environment User Guide: Platforms and Libraries* ([UG1146](#)), also available in the docs folder of the SDSoC environment.
4. *UltraFast Embedded Design Methodology Guide* ([UG1046](#))
5. *ZC702 Evaluation Board for the Zynq-7000 XC7Z020 All Programmable SoC User Guide* ([UG850](#))
6. *Vivado Design Suite User Guide: High-Level Synthesis* ([UG902](#))
7. *PetaLinux Tools Documentation: Workflow Tutorial* ([UG1156](#))
8. [Vivado® Design Suite Documentation](#)
9. *Vivado Design Suite User Guide: Creating and Packaging Custom IP* ([UG1118](#))

---

## Please Read: Important Legal Notices

The information disclosed to you hereunder (the “Materials”) is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available “AS IS” and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx’s limited warranty, please refer to Xilinx’s Terms of Sale which can be viewed at [www.xilinx.com/legal.htm#tos](http://www.xilinx.com/legal.htm#tos); IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx’s Terms of Sale which can be viewed at [www.xilinx.com/legal.htm#tos](http://www.xilinx.com/legal.htm#tos).

© Copyright 2015–2016 Xilinx, Inc. Xilinx, the Xilinx logo, Artix, ISE, Kintex, Spartan, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. All other trademarks are the property of their respective owners.