SDSoC Environment Tutorial

Introduction

UG1028 (v2016.3) November 30, 2016

XILINX ALL PROGRAMMABLE.™

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
11/30/2016	2016.3	Initial documentation release for SDx™ IDE 2016.3, which includes both the SDSoC™ Environment and the SDAccel™ Environment. Due to this major change in tool architecture, this document has undergone substantial changes in structure and content since the previous release.

Table of Contents

Chapter 1

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 2016.3 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. For more information, visit the SDSoC development environment web page.

Chapter 2

Flow Overview

Lab 1: Introduction to the SDSoC Development Environment

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 even 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.*

Learning Objectives

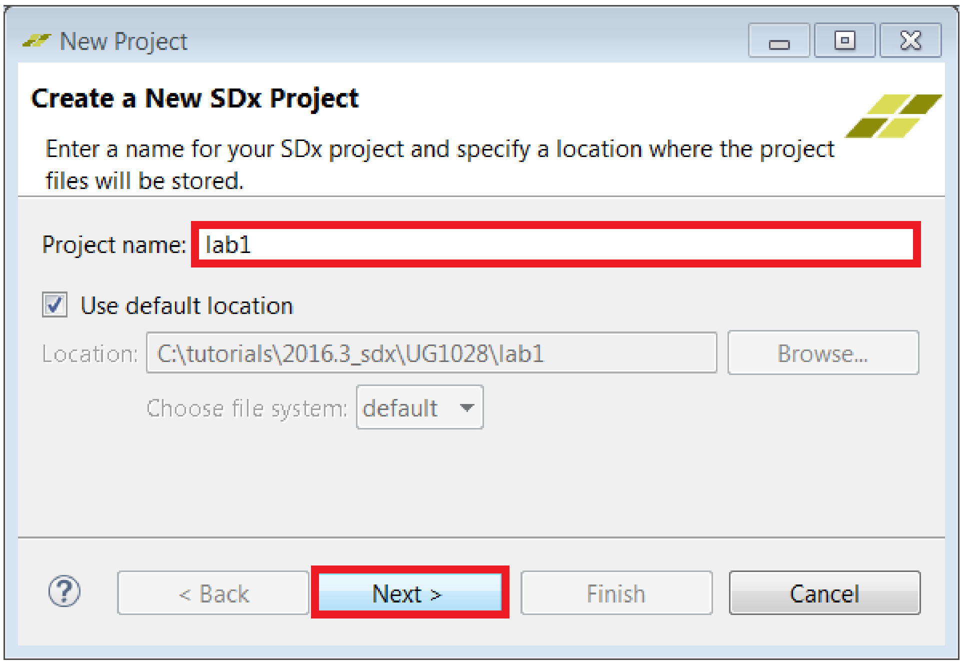
After you complete the tutorial (`lab1`), you should be able to:

- 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.

Creating a New Project

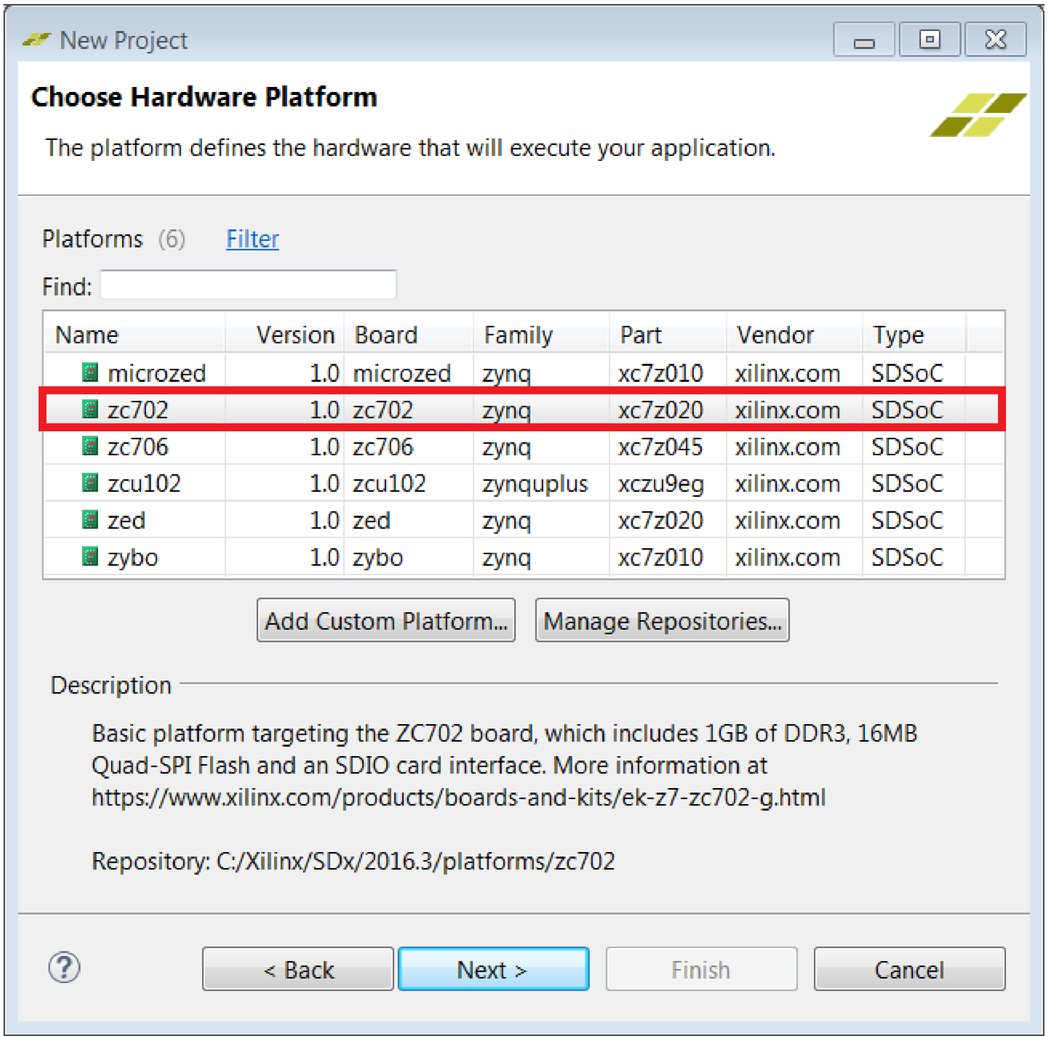
1. Launch the SDx IDE 2016.3 using the desktop icon or the **Start** menu.
2. When you launch the SDx IDE, the **Workspace Launcher** dialog appears. Click **Browse** to enter a workspace folder used to store your projects (you can use workspace folders to organize your work), then click **OK** to dismiss the **Workspace Launcher** dialog.

3. The SDx IDE window opens with the **Welcome** tab visible when you create a new workspace. The tab includes links for **Creating a new Xilinx SDx Project**, **Importing an existing project**, **Tutorials**, and **Web Resources**. Clicking any of these links takes you to further options available under each link. For example, to access documentation and tutorials, clicking on **Tutorials** takes you to the Tutorials page which has links for SDSoC and SDAccel related documents. The **Welcome** tab can be dismissed by clicking the X icon or minimized if you do not wish to use it.
4. From the SDx IDE menu bar select **File→New→Xilinx SDx Project.** The New Project dialog box opens.



5. Specify the name of the project, for example **lab1**.
6. Click **Next**.

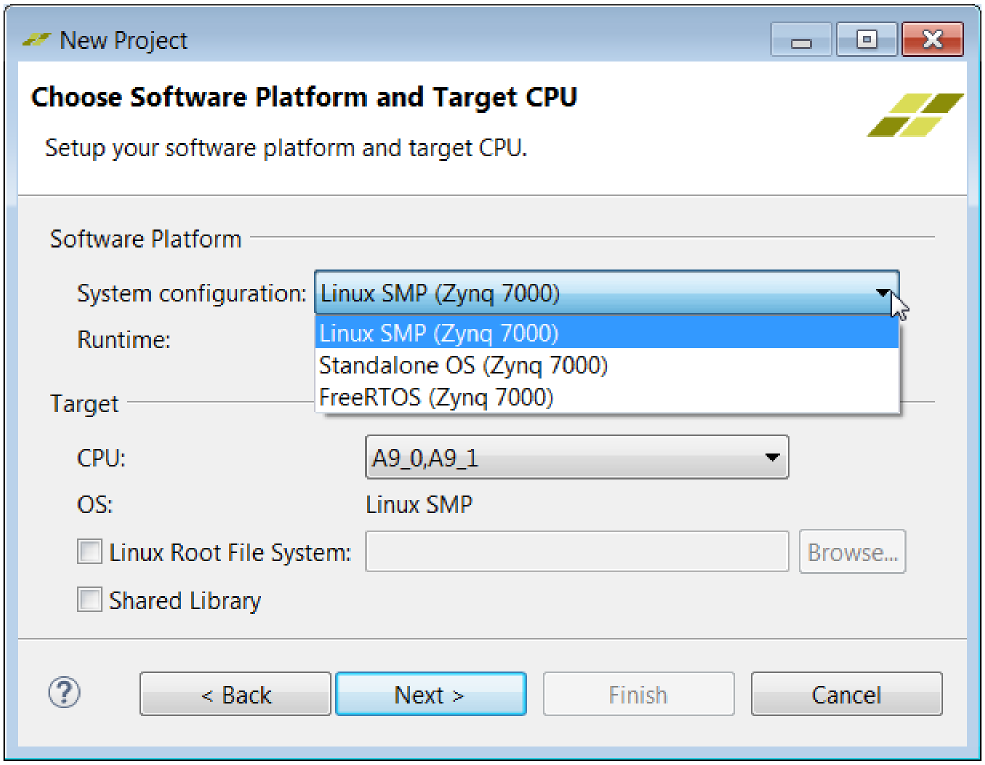
7. From the **Choose Hardware Platform** page, select the **zc702** platform.



NOTE: *If a custom platform is being used that is not in the list of supported platforms, click **Add Custom Platform** to add the custom platform.*

8. Click **Next**.

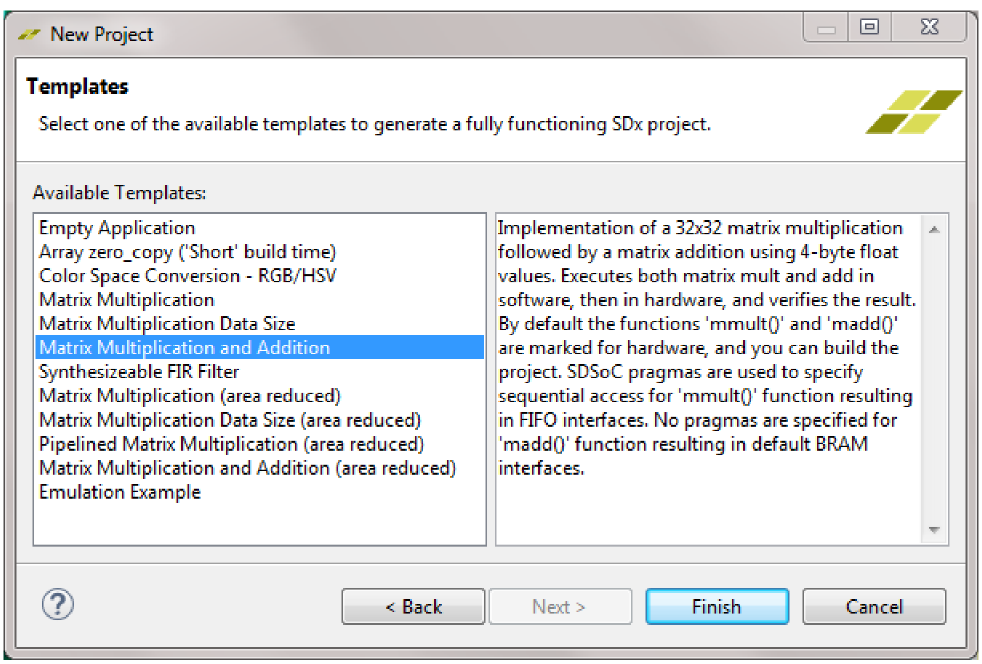
9. From the **System configuration** drop-down list for the selected platform, select **Linux SMP (Zynq 7000)**. Leave all other fields at their default values.



10. Click **Next**.

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

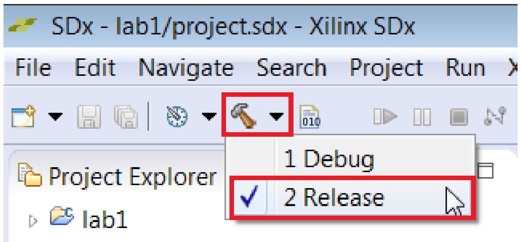
11. From the list of application templates, select **Matrix Multiplication and Addition** and click **Finish**.



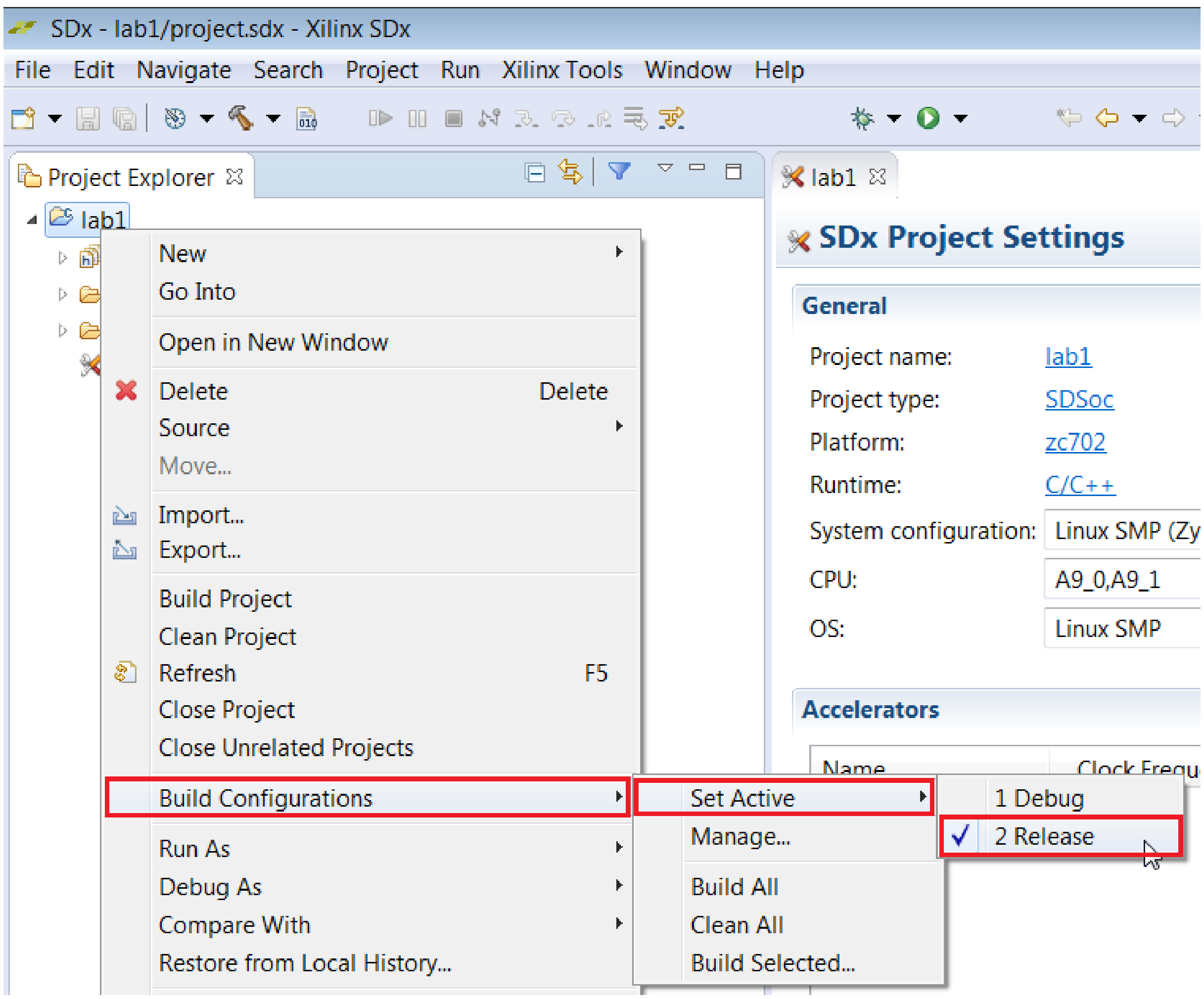
12. The standard build configurations are Debug and Release, and you can create additional build configurations. To get the best runtime performance, switch to use the Release configuration using one of the three methods illustrated below. The Release build configuration uses a

higher compiler optimization setting than the Debug build configuration. The SDx Project Settings window also allows you to select the active configuration or create a build configuration.

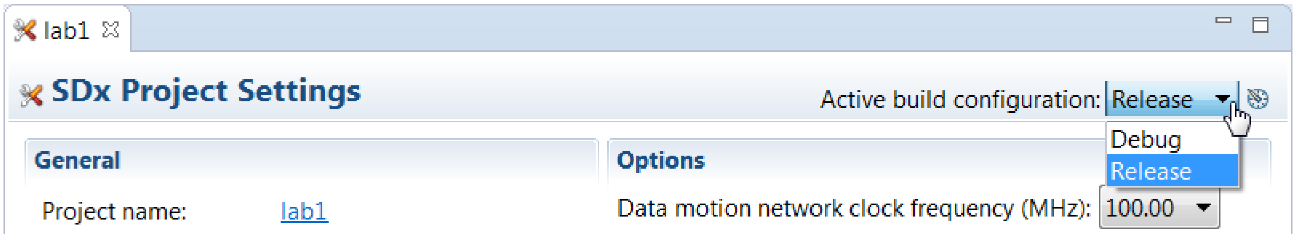
The **Build** icon provides a drop-down menu for selecting the build configuration and building the project. Clicking on the **Build** icon builds the project.



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



The SDx Project Settings window includes a Build Configurations drop-down, where you can select the active configuration or create a build configuration.



The SDx Project Settings window provides a summary of the project settings.

When you build an SDx application, you use a build configuration (a collection of tool settings, folders and files). Each build configuration has a different purpose. **Debug** 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 **Release** configuration provides the same ELF file as the Debug configuration 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 Performance Estimation).

Marking Functions for Hardware Implementation

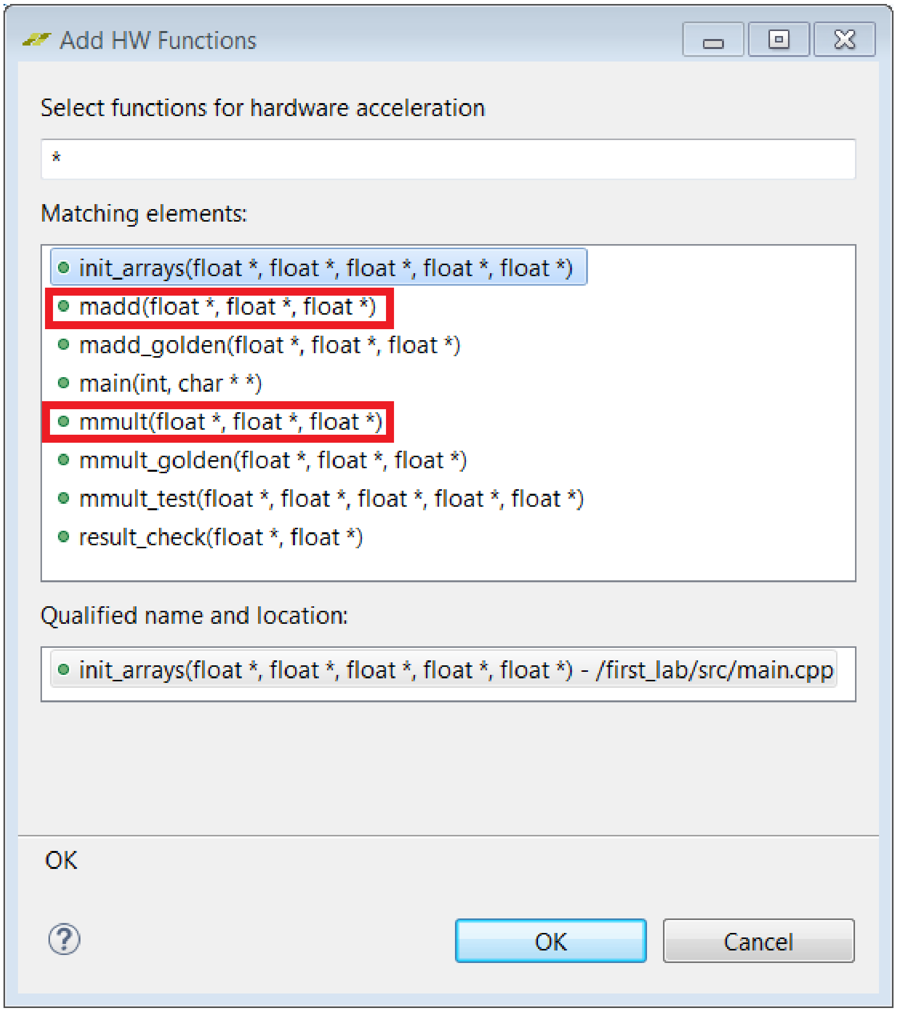
This application has two hardware functions. One hardware function, `mmult`, multiplies two matrices to produce a matrix product, and the second hardware function, `madd`, adds two matrices to produce a matrix sum. These hardware functions are combined to compute a matrix multiply-add function. Both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

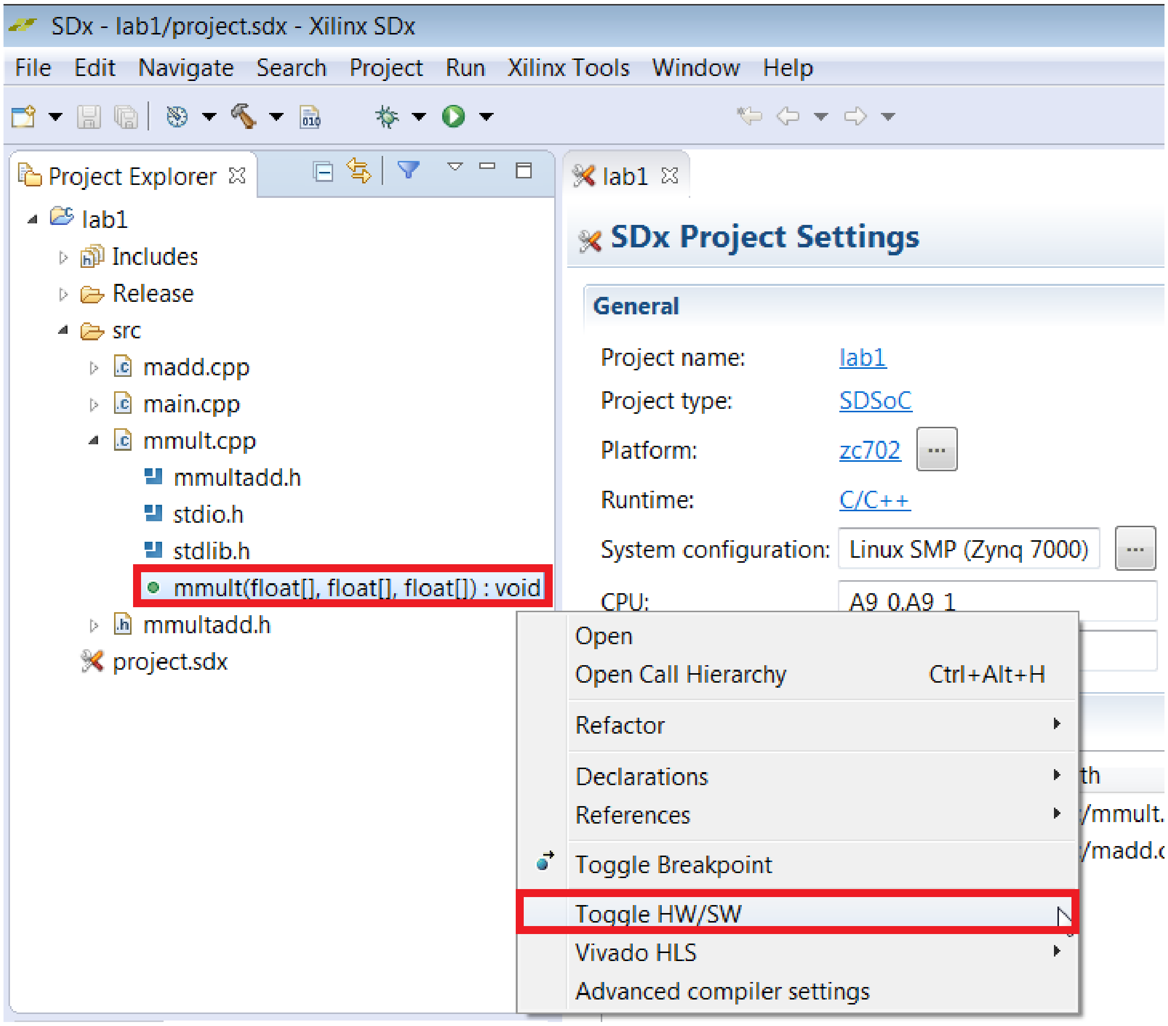
NOTE: *For this lab, you do not need to mark functions for hardware – the template code for matrix multiplication and addition has already marked them. If you don't have the* `madd` *and* `mmult` *functions marked as HW Functions, you could do the following to mark them as HW Functions.*

1. The SDx Project Settings window provides a central location for setting project values. Click on the tab labeled **lab1** (if the tab is not visible, double-click on the **project.sdx** file in the Project Explorer tab) and in the **HW Functions** window, click on the **Add HW Functions** 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 the function `mmult(float[], float[], float[]): void [H]` in the Project Explorer tab). 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) for more information.*

Building a Design with Hardware Accelerators

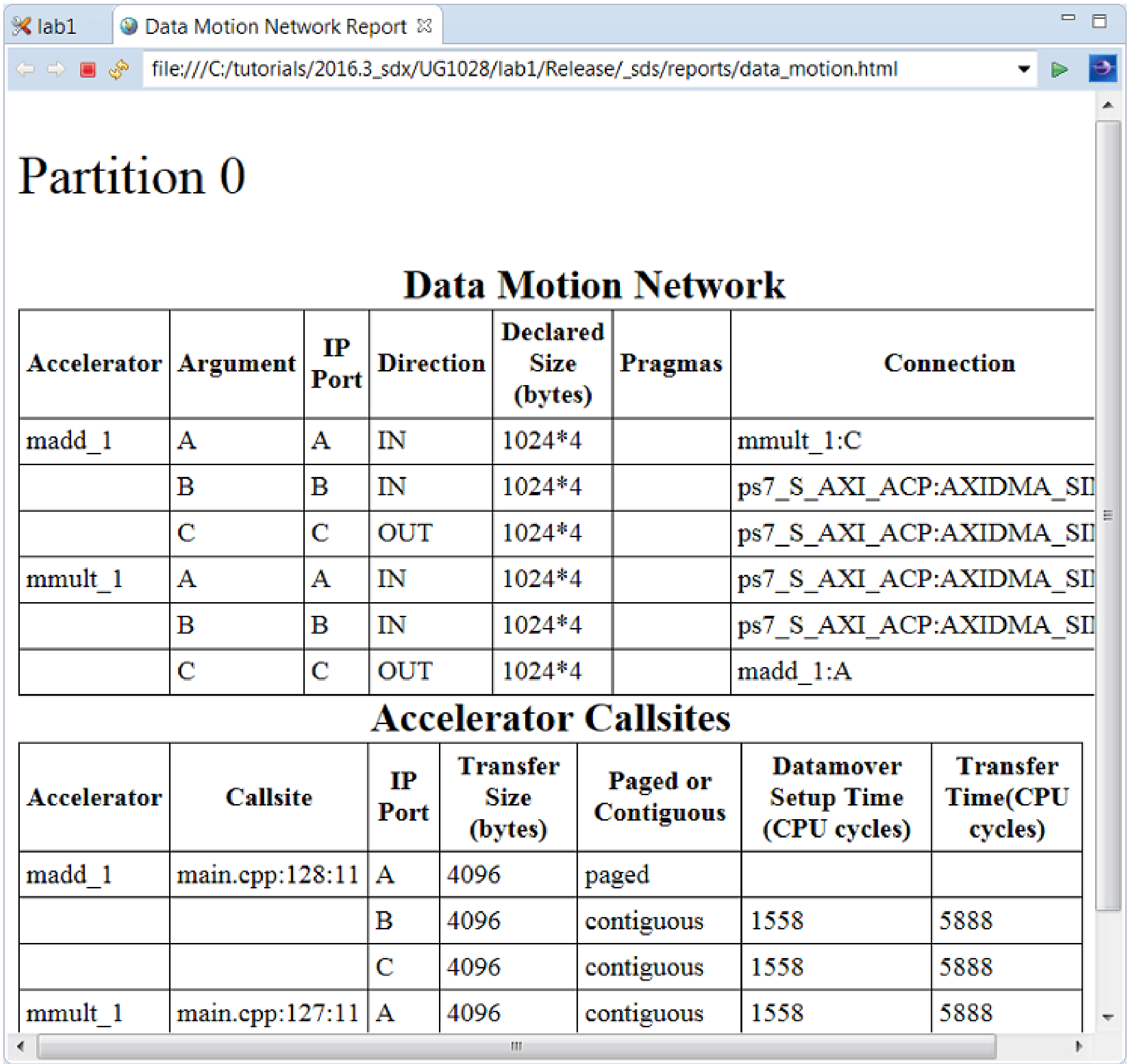
To build a project and generate an executable, bitstream, and SD Card boot image:

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.

2. In the **SDx Project Settings** window, under the **Reports** tab, below the **Project Explorer** tab, double-click to open the **Data Motion Network Report**.

   This report shows the connections done by the SDx environment and the types of data transfers for each function implemented in hardware. For details, see Application Code Optimization.



lab1 | Data Motion Network Report

file:///C:/tutorials/2016.3_sdx/UG1028/lab1/Release/_sds/reports/data_motion.html

Partition 0

Data Motion Network

Accelerator	Argument	IP Port	Direction	Declared Size (bytes)	Pragmas	Connection
madd_1	A	A	IN	1024*4		mmult_1:C
	B	B	IN	1024*4		ps7_S_AXI_ACP:AXIDMA_SI
	C	C	OUT	1024*4		ps7_S_AXI_ACP:AXIDMA_SI
mmult_1	A	A	IN	1024*4		ps7_S_AXI_ACP:AXIDMA_SI
	B	B	IN	1024*4		ps7_S_AXI_ACP:AXIDMA_SI
	C	C	OUT	1024*4		madd_1:A

Accelerator Callsites

Accelerator	Callsite	IP Port	Transfer Size (bytes)	Paged or Contiguous	Datamover Setup Time (CPU cycles)	Transfer Time(CPU cycles)
madd_1	main.cpp:128:11	A	4096	paged		
		B	4096	contiguous	1558	5888
		C	4096	contiguous	1558	5888
mmult_1	main.cpp:127:11	A	4096	contiguous	1558	5888

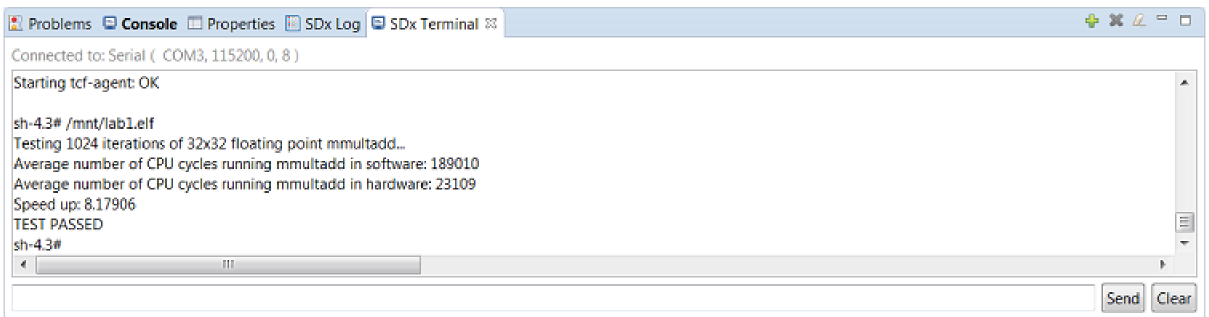
3. Open the `lab1/Release/_sds/swstubs/mmult.cpp` file, to see how the SDx system compiler replaced the original `mmult` function with one named `_p0_mmult_1_noasync` that performs transfers to and from the FPGA using `cf_send_i` and `cf_wait` functions. The SDx system compiler also replaces calls to `mmult` with `_p0_mmult_1_noasync` in `lab1/Release/_sds/swstubs/main.cpp`. The SDx 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/Release` 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 SDx environment (or connect via Putty/Teraterm with **Baud Rate**: 115200, **Data bits**: 8, **Stop bits**: 1, **Parity**: None and **Flow Control**: None). Click the icon to open the settings.

4. Keep the default settings in the Connect to Serial Port window and click **OK**.
5. After the board boots up, you can execute the application at the Linux prompt. Type `/mnt/lab1.elf`.



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 SDx™ system compiler?
- Examine the contents of the `Release/_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 `Release/_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 8.1 times faster. The application running on the processor takes about 190k cycles while the application running on both the processor and FPGA takes about 23k 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

Chapter 3

Performance Estimation

Lab 2: Performance Estimation

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 even if you do not have a ZC702 board. When creating the SDSoC environment project, select your board and one of the available templates, 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

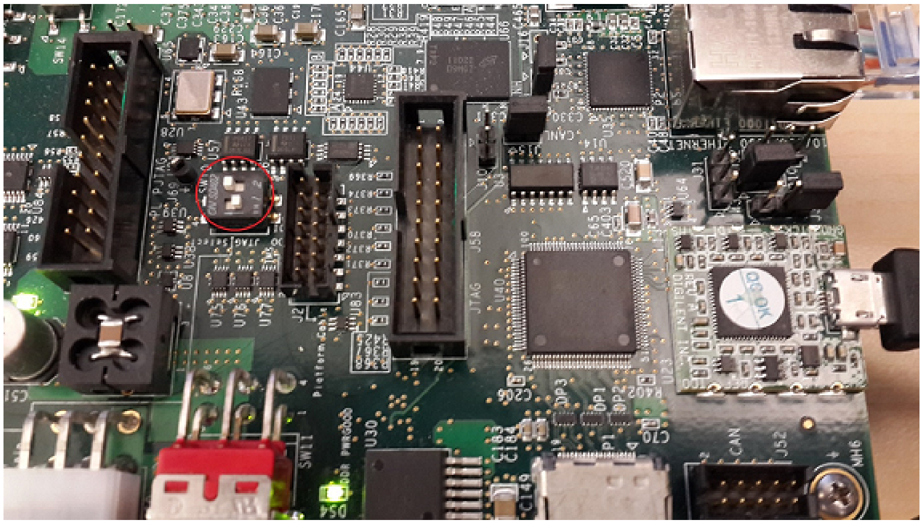
After you complete the tutorial, you should be able to use the SDSoC environment to obtain an estimate of the speedup that you can expect from your selection of functions implemented in hardware.

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 SDx IDE. 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.

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.
4. Power on the board.

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

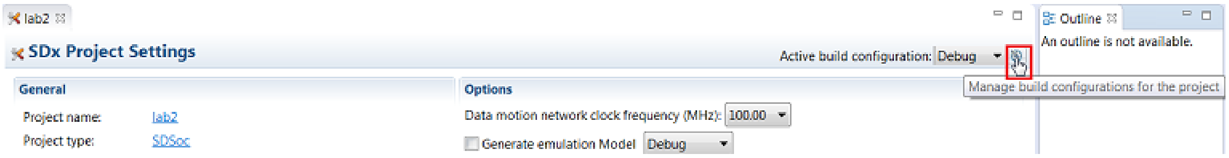
IMPORTANT: *Make sure that the jumper settings on the board correspond to SD-boot or JTAG-boot. Otherwise the board may power up in some other mode such as QSPI boot, and attempt to load something from the QSPI device or other boot device, which is not related to this lab.*

Setting up the Project for Performance Estimation

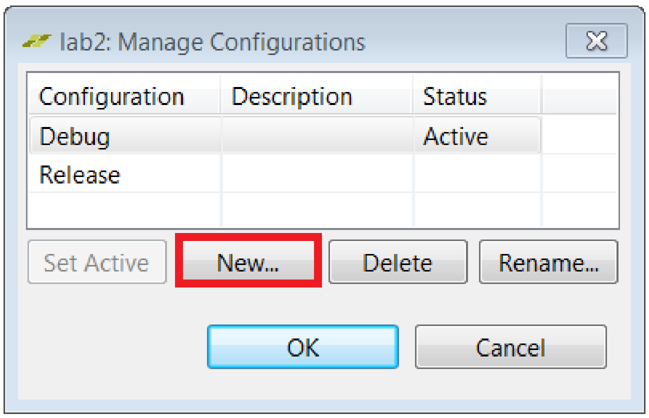
To create a project and use the Estimate Performance option in a build configuration:

1. Create a new project in the SDx™ IDE 2016.3 (lab2) for the **ZC702** platform and **Standalone OS (Zynq 7000)** as System configuration using the design template for **Matrix Multiplication and Addition**.
2. Click on the tab labeled **lab2** to view the **SDx Project Settings**. If the tab is not visible, in the **Project Explorer** double click on the `project.sdx` file under the **lab2** project.
3. In the **HW Functions** panel, observe that the `madd` and `mmult` functions already appear in the list of functions marked for hardware – template projects in the SDx environment include information for automating the process of marking hardware functions.
4. If the **HW Functions** panel did not list any functions, you would click on the **Add HW 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.

5. Performance estimation can be run using any build configuration. Instead of selecting **Debug** or **Release** as the Active Configuration, you could instead click on the **Manage build configuration for the project** icon next to the active configuration.



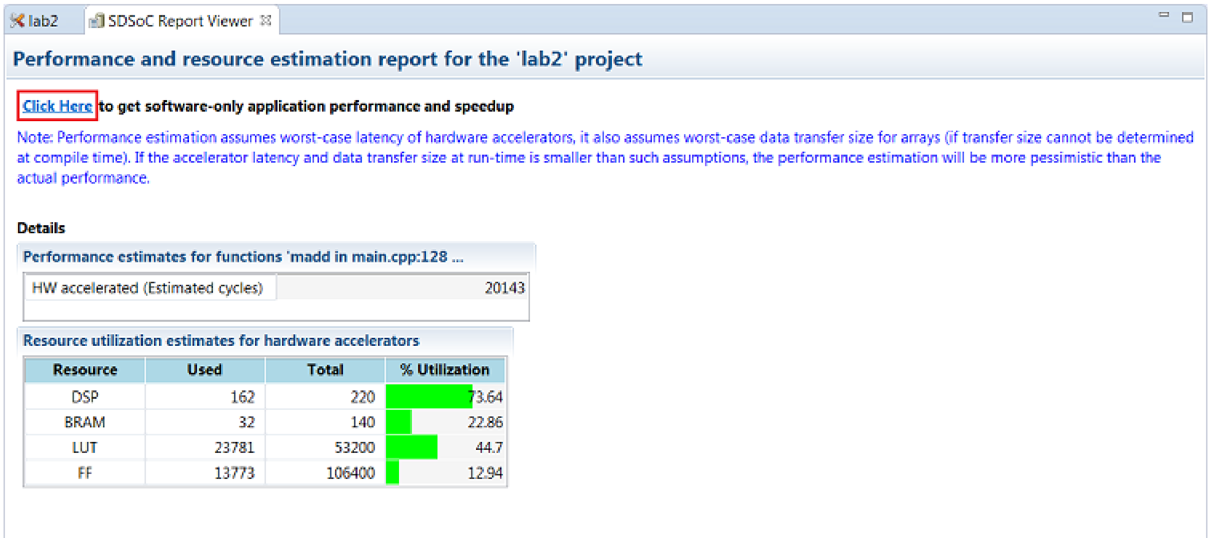
6. You can choose an available configuration or you can create a new configuration. New configuration can be created from an existing configuration (as a starting point) or it can be created from scratch. Using the Debug build configuration or another build configuration copied from Debug will compile the code with -O0 using GCC, so the software performance will be significantly degraded.



7. In the **SDx Project Settings** in the **Options** panel, check the **Estimate Performance** box. This enables the estimation flow.
8. The **Build** toolbar button provides a drop-down menu for selecting the build configuration and building the project. Clicking the **Build** icon builds the project. If the **Estimate Performance** option is checked, then performance estimation also occurs. Click the **Build** button on the toolbar.

   The SDx IDE 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.



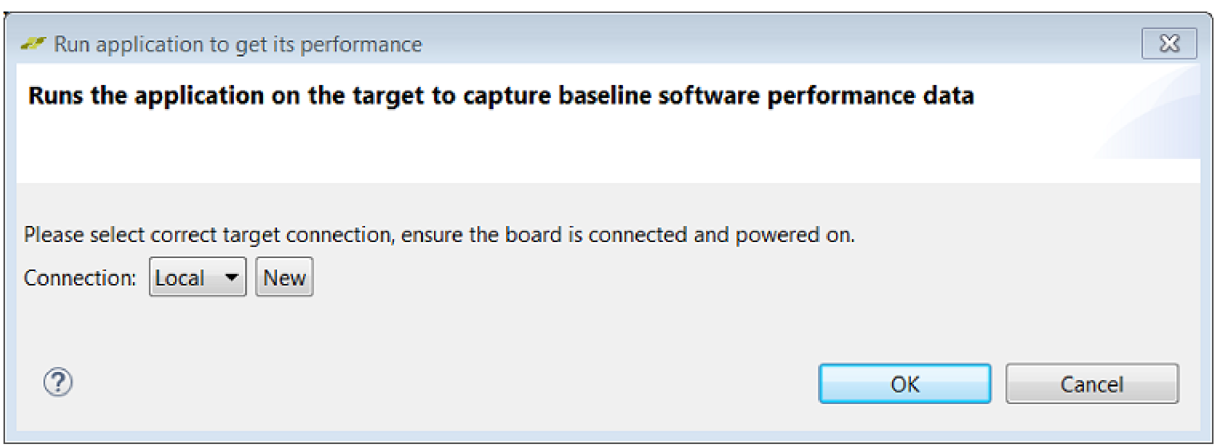
Comparing Software and Hardware Performance

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:

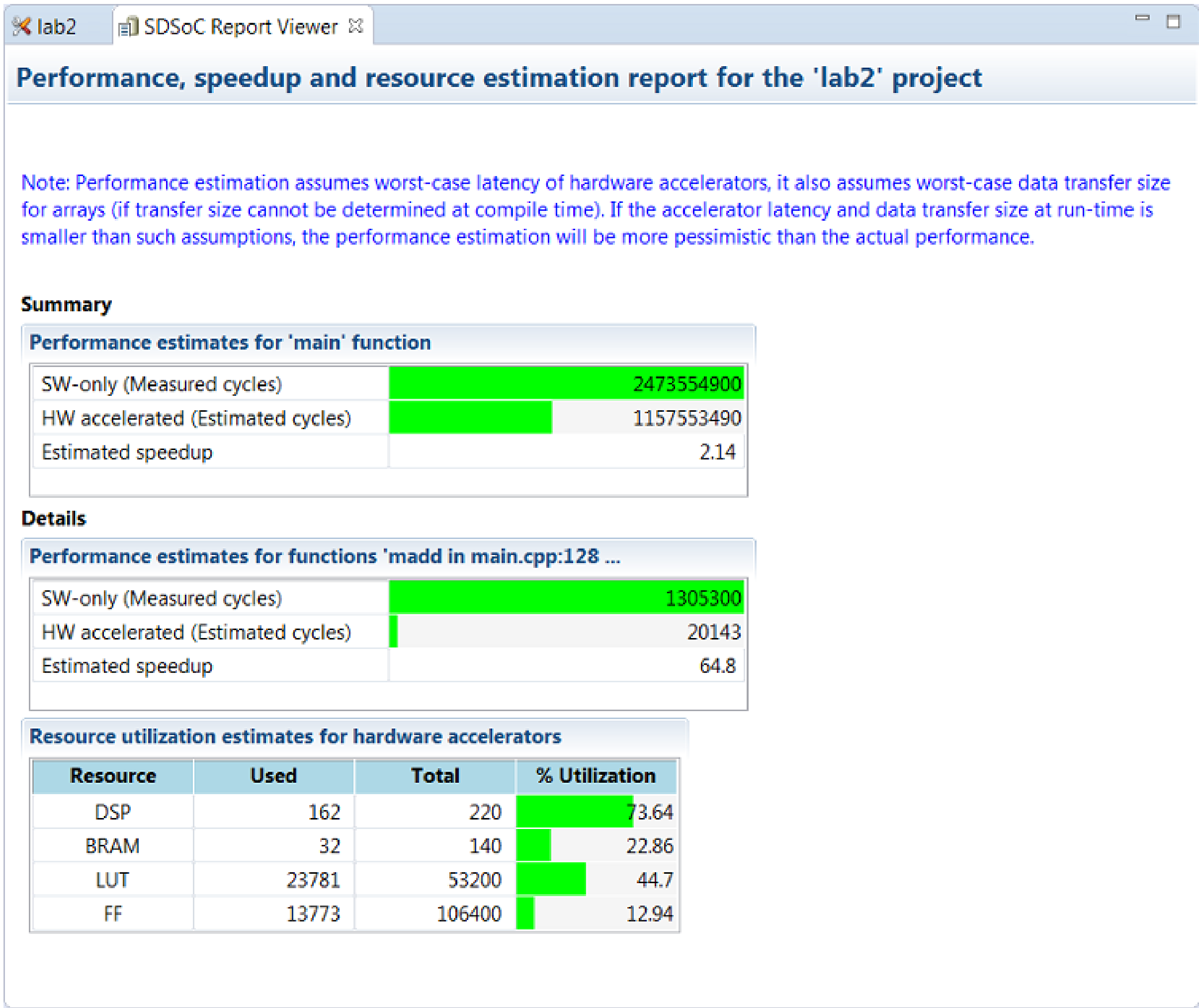
1. After the build completes, the SDSoC Report Viewer tab opens.
2. Click the **Click Here** hyperlink on the viewer to launch the application on the board.

   The Run application to get its performance 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, and runs a software-only version of the application. It then collects performance data and uses it to display the performance estimation report.


lab2 | SDSoC Report Viewer

Performance, speedup and resource estimation report for the 'lab2' project

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.

Summary

Performance estimates for 'main' function

SW-only (Measured cycles)	2473554900
HW accelerated (Estimated cycles)	1157553490
Estimated speedup	2.14

Details

Performance estimates for functions 'madd in main.cpp:128 ...

SW-only (Measured cycles)	1305300
HW accelerated (Estimated cycles)	20143
Estimated speedup	64.8

Resource utilization estimates for hardware accelerators

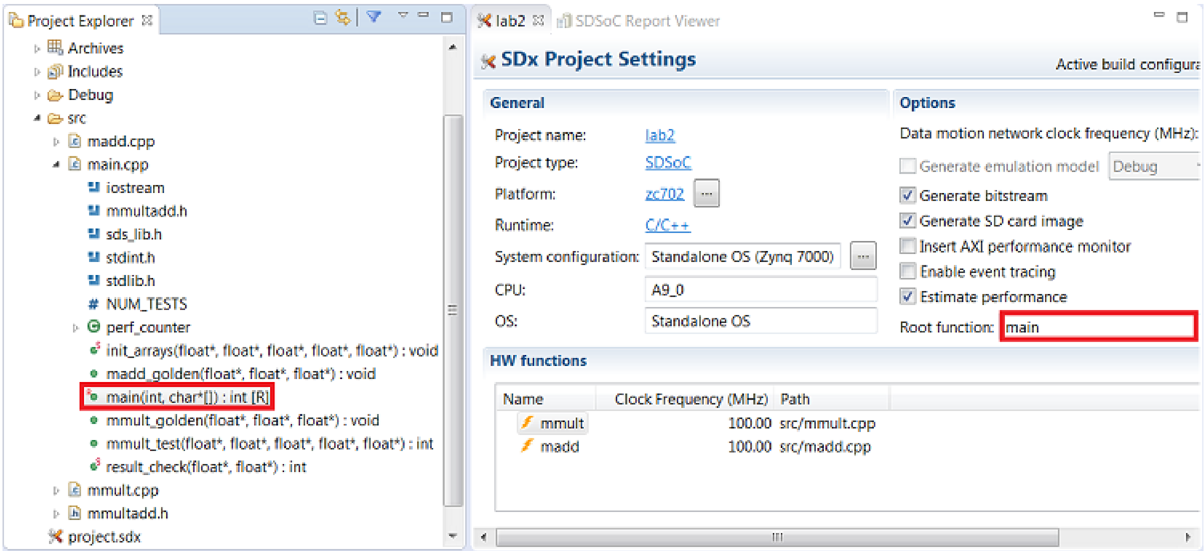
Resource	Used	Total	% Utilization
DSP	162	220	73.64
BRAM	32	140	22.86
LUT	23781	53200	44.7
FF	13773	106400	12.94

Changing Scope of Overall Speedup Comparison

In the Performance, speedup and resource estimation report, the Summary section 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. In the SDx Project Settings window, click the browse button on the **Root function** field to change the root for the 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 **SDx Project Settings**, click on **Estimate performance** 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 SDx™ IDE (lab2_linux) for the **ZC702** platform and System Configuration set to **Linux SMP (Zynq 7000)** using the design template for **Matrix Multiplication and Addition**.
2. Click on the tab labeled **lab2_linux** (if the tab is not visible, in the **Project Explorer** tab under the **lab2_linux** project double click on `project.sdx`). In the **HW Functions** panel, observe that the `madd` and `mmult` functions already appear in the list of functions marked for hardware – template projects in the SDx environment include information for automating the process of marking hardware functions.
3. If the **HW Functions** panel did not list any functions, you would click on the **Add HW Functions** 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.
4. In the **SDx Project Settings** in the **Options** panel, check the **Estimate performance** box. This enables the performance estimation flow for the current build configuration.

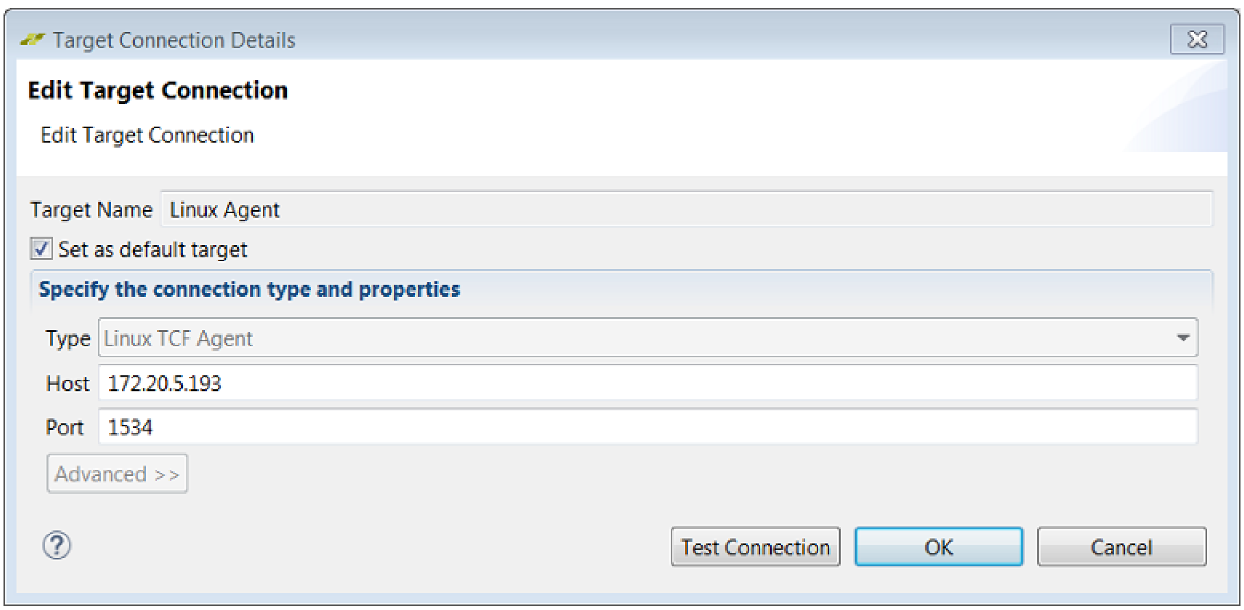
5. The **Build** icon provides a drop-down menu for selecting the build configuration and building the project. Clicking on the **Build** icon builds the project and with the **Estimate performance** option checked, the performance estimation flow runs. Click **Build**.

   The SDx IDE builds the project. A dialog box displaying the status of the build process appears.

6. For this lab, you will also need an Ethernet cable to connect to the board. Ensure that the board is connected to an Ethernet router using the Ethernet cable. First, copy the contents of the `sd_card` folder under the build configuration to an sd card and boot up the board. Then make sure that a serial terminal is also connected.
7. Note 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` *in the terminal window at the prompt.*

8. Back in the SDx IDE 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 SDx IDE runs a software-only version of the application. It then collects performance data and uses it to display the performance estimation report.

Chapter 4

Application Code Optimization

Lab 3: Optimize the Application Code

This tutorial demonstrates how you can modify your code to optimize the hardware-software system generated by the SDx environment. You will 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.*

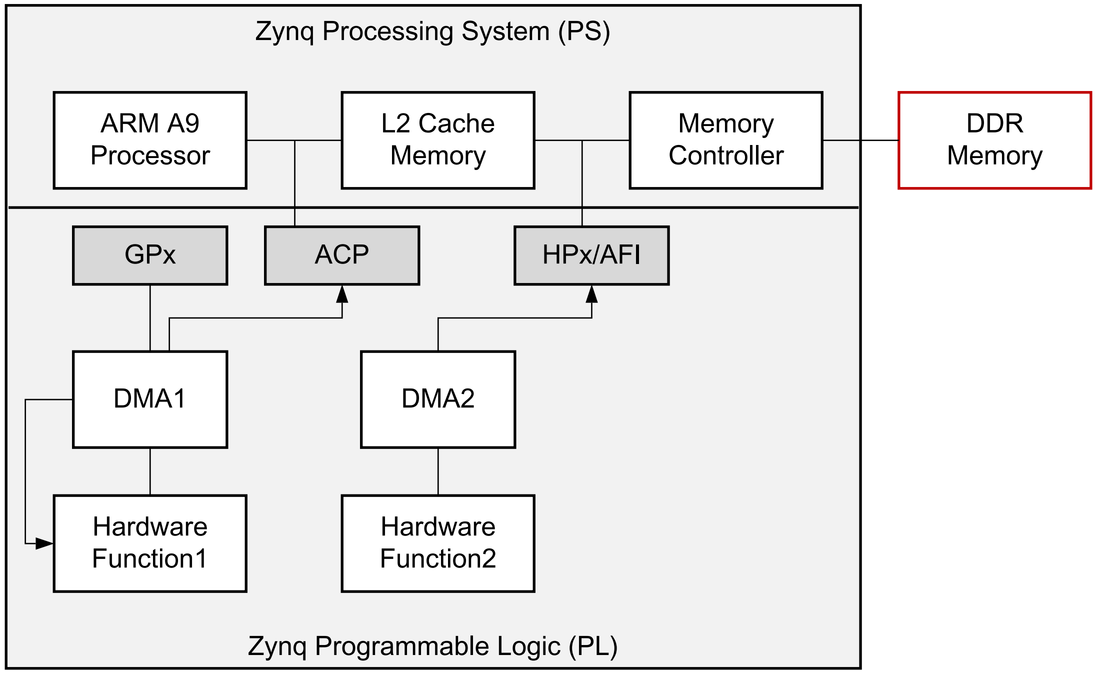
NOTE: *You can complete this tutorial even if you do not have a ZC702 board. When creating the SDSoC environment project, select your board and one of the available applications if the suggested template* ***Matrix Multiplication and Addition*** *is not found. For example, boards such as the MicroZed 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 SDx IDE 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 1: Simplified Zynq + DDR Diagram Showing Memory Access Ports and Memories



When the software running on the ARM A9 processor "calls" a hardware function, it actually invokes an `sds++` generated stub function that in turn calls underlying drivers to send data from the processor memory 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 `sds++` compiler 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	Hardware functions have cache coherent access to DDR via the PS L2 cache.
AFI (HP)	Hardware functions have fast non-cache coherent access to DDR via the PS memory controller.
GP	Processor directly writes/reads data to/from hardware function. Inefficient for large data transfers.
MIG	Hardware functions access DDR from PL via a MIG IP memory controller.

Learning Objectives

After you complete the tutorial (lab3), 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

1. Create a new project in the SDx™ IDE (lab3) for the **ZC702 platform** and **Linux SMP (Zynq 7000)** System configuration using the design template for **Matrix Multiplication and Addition**.
2. Click on the tab labeled **lab3** to view the **SDx Project Settings**. If the tab is not visible, in the **Project Explorer** double click on the `project.sdx` file under the **lab3** project.
3. In the **HW Functions** panel, observe that the `madd` and `mmult` functions already appear in the list of functions marked for hardware acceleration.
4. To get the best runtime performance, switch to use the **Release** configuration by clicking on the Active Build Configuration option and then selecting Release. You could also select **Release** from the Build icon, or by right-clicking the project and selecting **Build Configurations→Set Active→Release.** The Release build configuration uses a higher compiler optimization setting than the Debug build configurations.

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 SDx system compiler, so set project linker options to prevent generating the bit stream, boot image and build.
   a. Click on the **lab3** tab to select the SDx Project Settings.
   b. Deselect the **Generate bitstream** and **Generate SD card image** check boxes.
2. Right-click on the top level folder for the project in Project Explorer and select **Build Project**.
3. When the build completes, in the **Reports** panel, double-click **Data Motion Network Report** 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.

lab3 | Data Motion Network Report

file:///C:/tutorials/2016.3_sdx/UG1028/lab3/Release/_sds/reports/data_motion.html

Partition 0

Data Motion Network

Accelerator	Argument	IP Port	Direction	Declared Size(bytes)	Pragmas	Connection
madd_1	A	A	IN	1024*4		mmult_1:C
	B	B	IN	1024*4		ps7_S_AXI_ACP:AXIDMA_SIMPLE
	C	C	OUT	1024*4		ps7_S_AXI_ACP:AXIDMA_SIMPLE
mmult_1	A	A	IN	1024*4		ps7_S_AXI_ACP:AXIDMA_SIMPLE
	B	B	IN	1024*4		ps7_S_AXI_ACP:AXIDMA_SIMPLE
	C	C	OUT	1024*4		madd_1:A

Accelerator Callsites

Accelerator	Callsite	IP Port	Transfer Size(bytes)	Paged or Contiguous	Datamover Setup Time(CPU cycles)	Transfer Time (CPU cycles)
madd_1	main.cpp:128:11	A	4096	paged		
		B	4096	contiguous	1558	5888
		C	4096	contiguous	1558	5888
mmult_1	main.cpp:127:11	A	4096	contiguous	1558	5888
		B	4096	contiguous	1558	5888
		C	4096	paged		

4. Add `sys_port` pragma.

    a. Double-click `mmultadd.h` file in the Project Explorer view, under the `src` folder, 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.

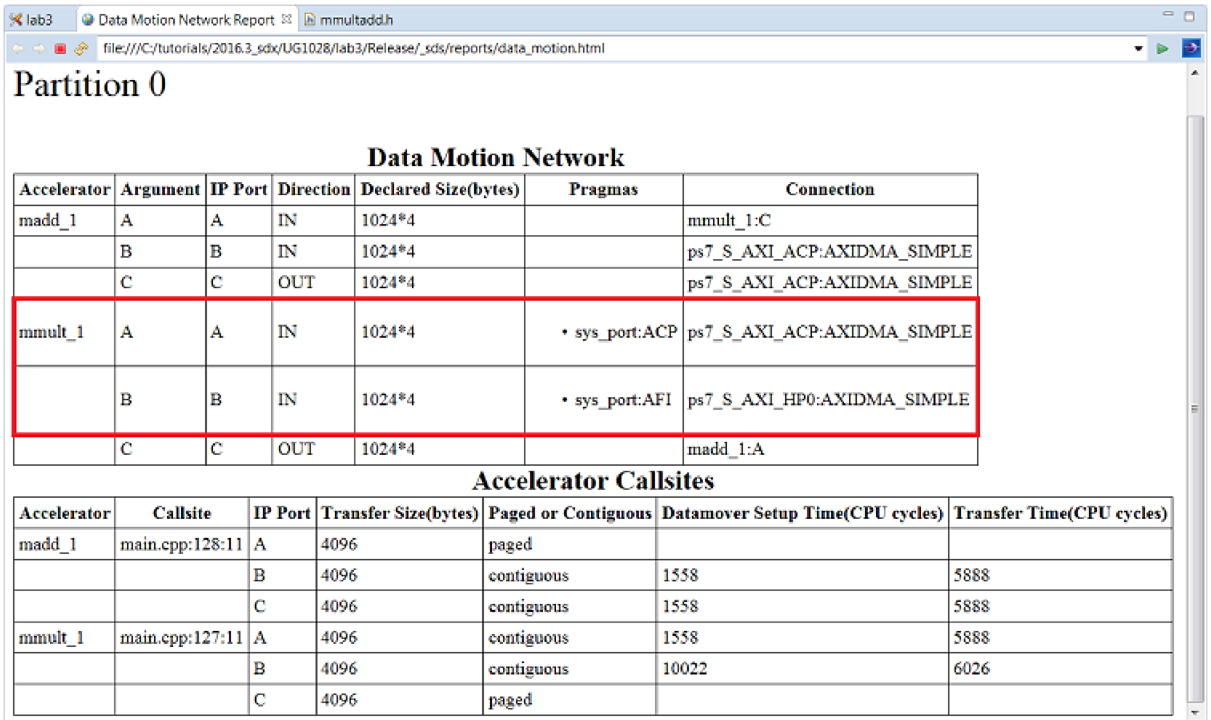
```
#pragma SDS data sys_port(A:ACP, B:AFI)
```

```
/**
 * Design principles to achieve best performance
 *
 * 1. Declare secquential access to stream data into accelerators via a hardware FIFO
 *    interface.  Otherwise, the default RAM interface requires all data to arrive
 *    before starting HLS accelerator
 */
#pragma SDS data access_pattern(A:SEQUENTIAL, B:SEQUENTIAL, C:SEQUENTIAL)
#pragma SDS data sys_port(A:ACP, B:AFI)
void mmult (float A[N*N], float B[N*N], float C[N*N]);

#pragma SDS data access_pattern(A:SEQUENTIAL, B:SEQUENTIAL, C:SEQUENTIAL)
void madd(float A[N*N], float B[N*N], float C[N*N]);

#endif /* _MMULTADD_H_ */
```

    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.
7. When the build completes, click on the tab showing the Data Motion Network Report (`data_motion.html` file).
8. Click anywhere in the Data Motion Network Report pane and select **Refresh** from the context menu.

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



Partition 0

Data Motion Network

Accelerator	Argument	IP Port	Direction	Declared Size(bytes)	Pragmas	Connection
madd_1	A	A	IN	1024*4		mmult_1:C
	B	B	IN	1024*4		ps7_S_AXI_ACP:AXIDMA_SIMPLE
	C	C	OUT	1024*4		ps7_S_AXI_ACP:AXIDMA_SIMPLE
mmult_1	A	A	IN	1024*4	• sys_port:ACP	ps7_S_AXI_ACP:AXIDMA_SIMPLE
	B	B	IN	1024*4	• sys_port:AFI	ps7_S_AXI_HP0:AXIDMA_SIMPLE
	C	C	OUT	1024*4		madd_1:A

Accelerator Callsites

Accelerator	Callsite	IP Port	Transfer Size(bytes)	Paged or Contiguous	Datamover Setup Time(CPU cycles)	Transfer Time(CPU cycles)
madd_1	main.cpp:128:11	A	4096	paged		
		B	4096	contiguous	1558	5888
		C	4096	contiguous	1558	5888
mmult_1	main.cpp:127:11	A	4096	contiguous	1558	5888
		B	4096	contiguous	10022	6026
		C	4096	paged		

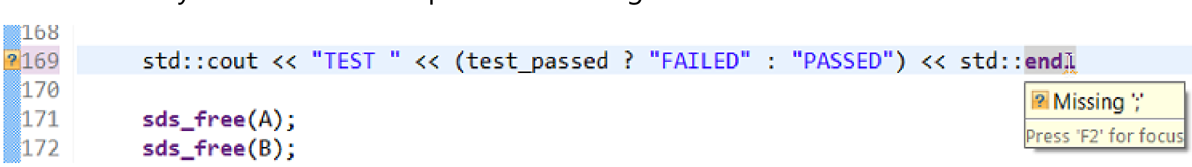
9. Delete the pragma `#pragma SDS data sys_port(A:ACP, B:AFI)` and save the file.

Error Reporting

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

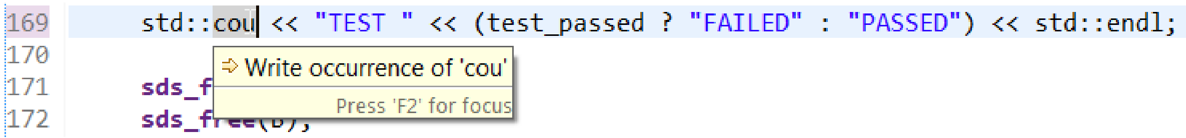
1. Open the source file `main.cpp` from the `src` folder and remove the semicolon at the end of the `std::cout` statement near the bottom of the file.

   Notice that a yellow box shows up on the left edge of the line.

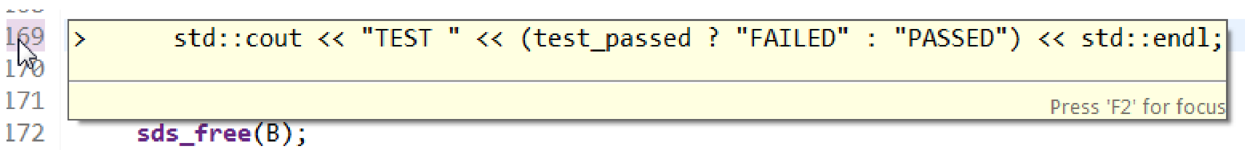


2. Move your cursor over the yellow box and notice that it tells you that you have a missing semicolon.
3. Insert the semicolon at the right place and notice how the yellow box disappears.

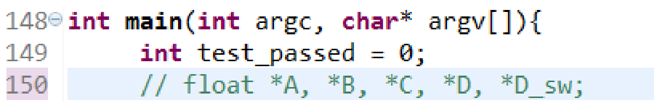
4. Now change `std::cout` to `std::cou` and notice how a pink box shows up on the left edge of the line.



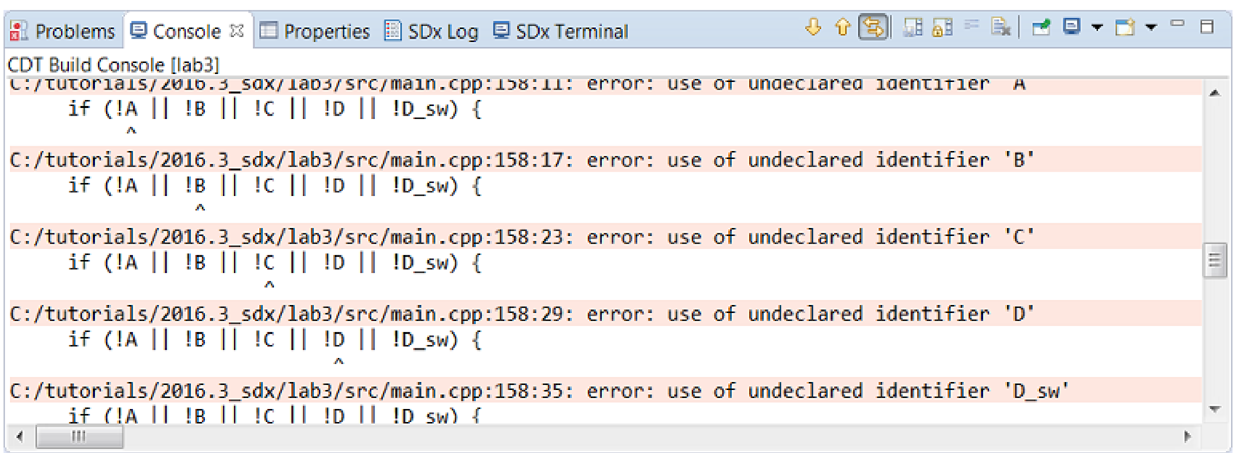
5. Move the cursor over the pink box to see a popup displaying the "corrected" version of the line with `std::cout` instead of `std::cou`.



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 `Release/_sds/reports/sds_main.log` and `Release/_sds/reports/sds_mmult.log` files to see the detailed error reports.



10. Uncomment the line where the variables are declared.

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.

Flow Overview used the `mult_add` template 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 lab3 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 `mmultadd.h` in the folder view under `lab3/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(A:AXIDMA_SG, B:AXIDMA_SIMPLE, 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 **Data Motion Report** from the Reports tab.

   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`.*

Data Motion Network

Accelerator	Argument	IP Port	Direction	Declared Size(bytes)	Pragmas	Connection
madd_1	A	A	IN	1024*4		ps7_S_AXI_ACP:AXIDMA_SG
	B	B	IN	1024*4		ps7_S_AXI_ACP:AXIDMA_SIMPLE
	C	C	OUT	1024*4		ps7_S_AXI_ACP:AXIDMA_SIMPLE
mmult_1	A	A	IN	1024*4	• data_mover:AXIDMA_SG	ps7_S_AXI_ACP:AXIDMA_SG
	B	B	IN	1024*4	• data_mover:AXIDMA_SIMPLE	ps7_S_AXI_ACP:AXIDMA_SIMPLE
	C	C	OUT	1024*4	• data_mover:AXIFIFO	ps7_M_AXI_GP0:AXIFIFO

6. Remove the pragma `#pragma SDS data data_mover(A:AXIDMA_SG, B:AXIDMA_SIMPLE, C:AXIFIFO)` that you entered in step 2 and save the file.

Using malloc() instead of sds_alloc()

For this exercise you start with the source used in lab3, modify the source to use `malloc()` instead of `sds_alloc()`, and observe how the data mover changes from Simple DMA to Scatter-Gather DMA.

1. Double-click the `main.cpp` in the Project Explorer view, under src folder, 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 `Release/_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.

Data Motion Network

Accelerator	Argument	IP Port	Direction	Declared Size(bytes)	Pragmas	Connection
madd_1	A	A	IN	1024*4		ps7_S_AXI_ACP:AXIDMA_SG
	B	B	IN	1024*4		ps7_S_AXI_ACP:AXIDMA_SIMPLE
	C	C	OUT	1024*4		ps7_S_AXI_ACP:AXIDMA_SIMPLE
mmult_1	A	A	IN	1024*4	• data_mover:AXIDMA_SG	ps7_S_AXI_ACP:AXIDMA_SG
	B	B	IN	1024*4	• data_mover:AXIDMA_SIMPLE	ps7_S_AXI_ACP:AXIDMA_SIMPLE
	C	C	OUT	1024*4	• data_mover:AXIFIFO	ps7_M_AXI_GP0:AXIFIFO

8. Undo all the changes made in step 2 and save the file.

Adding Pragmas to Control the Amount of Data Transferred

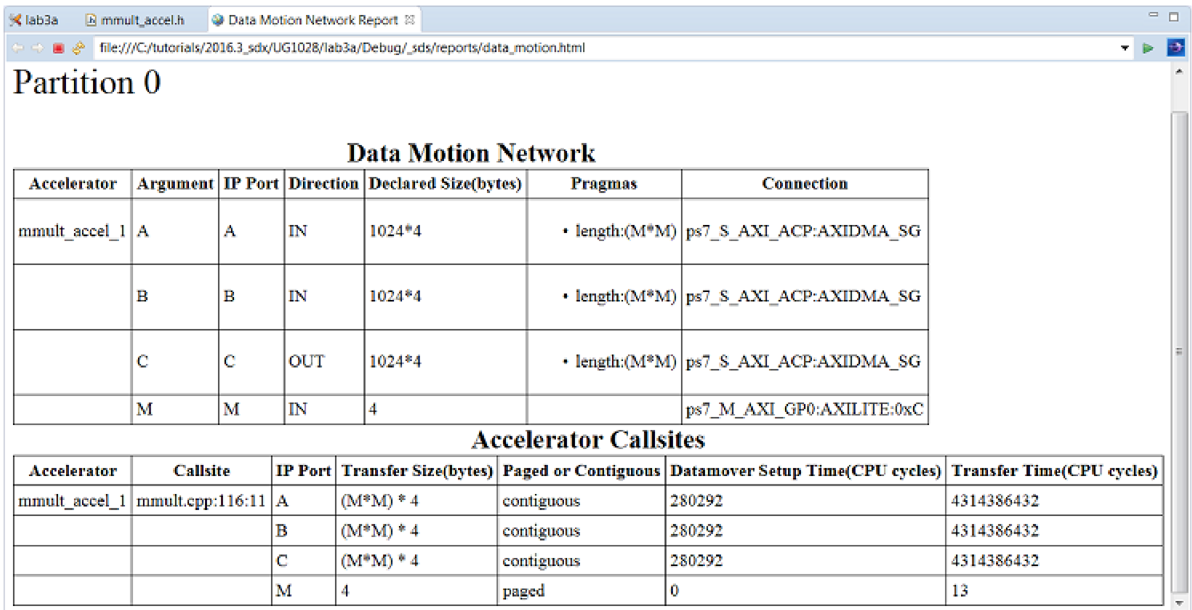
For this step, you use a different design template to show the use of the copy pragma. In this template an extra parameter called **M** is passed to the matrix multiply function. This parameter allows the matrix multiplier function to multiply two square matrices of any size **M***M** up to a maximum of 32*32. The top level allocation for the matrices creates matrices of the maximum size 32x32. The **M** 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 SDx 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→Xilinx SDx Project.**
    b. In the new project dialog box, type in a name for the project (for example `lab3a`)
    c. Select **zc702** and **Linux SMP (Zynq 7000)**.
    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 by deselecting the **Generate bitstream** and **Generate SD Card Image** checkboxes in the Options panel.
3. Add data copy pragmas by double-clicking **mmult_accel.h** in the Project Explorer view (under the `src` folder) 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, **M** is used to specify the size.

```
#pragma SDS data copy(A[0:M*M], B[0:M*M], C[0:M*M])
#
void mmult_accel (float A[N*N],
                  float B[N*N],
                  float C[N*N],
                  int M);
```

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.
6. When the build completes, in the Project Explorer view, double-click to open **Data Motion Network Report** in the Reports tab.
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_1	A	A	IN	1024*4	• length:(M*M)	ps7_S_AXI_ACP:AXIDMA_SG
	B	B	IN	1024*4	• length:(M*M)	ps7_S_AXI_ACP:AXIDMA_SG
	C	C	OUT	1024*4	• length:(M*M)	ps7_S_AXI_ACP:AXIDMA_SG
	M	M	IN	4		ps7_M_AXI_GP0:AXILITE:0xC

Accelerator Callsites

Accelerator	Callsite	IP Port	Transfer Size(bytes)	Paged or Contiguous	Datamover Setup Time(CPU cycles)	Transfer Time(CPU cycles)
mmult_accel_1	mmult.cpp:116:11	A	(M*M) * 4	contiguous	280292	4314386432
		B	(M*M) * 4	contiguous	280292	4314386432
		C	(M*M) * 4	contiguous	280292	4314386432
		M	4	paged	0	13

Chapter 5

Accelerator Optimization

Lab 4: Optimize the Accelerator Using Directives

In this exercise, you modify the source file in the project to observe the effects of Vivado HLS pragmas on the performance of generated hardware. See SDSoC Environment Optimization Guide (UG1235) for more information on this topic.

1. Create a new project in the SDx™ environment (lab4) for the **ZC702** Platform and **Linux SMP (Zynq 7000)** System Configuration using the design template for **Matrix Multiplication and Addition**.
2. Click on the tab labeled **lab4** to view the **SDx Project Settings**. If the tab is not visible, in the **Project Explorer** double click on the `project.sdx` file under the **lab4** project.
3. In the **HW Functions** panel, observe that the `madd` and `mmult` functions already appear in the list of functions marked for hardware acceleration.
4. To get the best runtime performance, switch to use the **Release** configuration by clicking on the Active Build Configuration option and then selecting Release. You could also select **Release** from the Build icon, or by right-clicking the project and selecting **Build Configuration→Set Active→Release**. The Release build configuration uses a higher compiler optimization setting than the Debug build configurations.
5. Double click the `mmult.cpp` in the Project Explorer view to bring up the source editor view.
6. Find the lines where the pragmas `HLS pipeline` and `HLS array_partition` are located.

7. Remove these pragmas by commenting out the lines.

```
void mmult (float A[N*N], float B[N*N], float C[N*N])
{
     float Abuf[N][N], Bbuf[N][N];
// #pragma HLS array_partition variable=Abuf block factor=16 dim=2
// #pragma HLS array_partition variable=Bbuf block factor=16 dim=1

     for(int i=0; i<N; i++) {
          for(int j=0; j<N; j++) {
// #pragma HLS PIPELINE
               Abuf[i][j] = A[i * N + j];
               Bbuf[i][j] = B[i * N + j];
          }
     }

     for (int i = 0; i < N; i++) {
          for (int j = 0; j < N; j++) {
// #pragma HLS PIPELINE
               float result = 0;
               for (int k = 0; k < N; k++) {
                    float term = Abuf[i][k] * Bbuf[k][j];
                    result += term;
               }
               C[i * N + j] = result;
          }
     }
}
```

8. Save your file.
9. Right click the top-level folder for the project and click **Build Project** in the menu.
10. After the build completes, copy the `lab4/Release/sd_card` folder to an SD card.
11. Insert the SD card into the ZC702 board and power on the board.
12. Connect to the board from a serial terminal in the SDx Terminal tab of the SDx IDE. Click the **+** icon to open the settings.
13. After the board boots up, you can execute the application at the Linux prompt. Type `/mnt/lab4.elf`.

    Observe the performance and compare it with the performance that was seen with the commented out pragmas present (compare it with the results of lab1). 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.

```
Connected to: Serial ( COM3, 115200, 0, 8 )
sh-4.3# /mnt/lab4.elf
Testing 1024 iterations of 32x32 floating point mmultadd...
Average number of CPU cycles running mmultadd in software: 184056
Average number of CPU cycles running mmultadd in hardware: 2252733
Speed up: 0.0817034
TEST PASSED
sh-4.3#
```

Lab 5: Task-Level Pipelining

This lab demonstrates how to modify your code to optimize the hardware-software system generated by the SDx IDE 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 even 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`).*

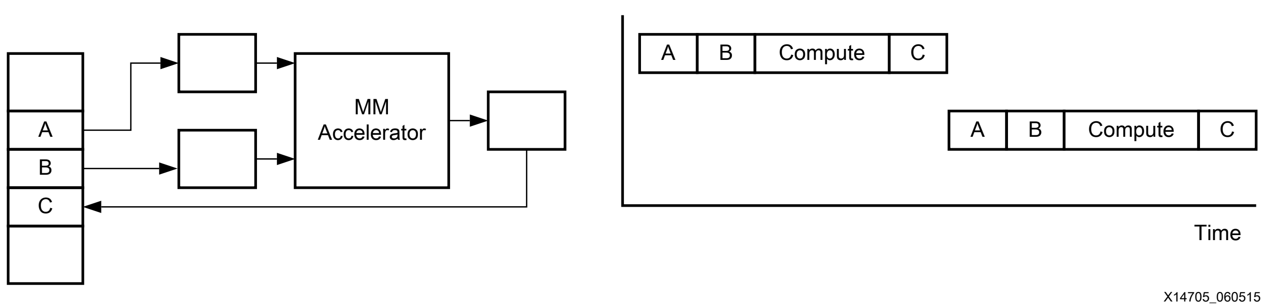
Task Pipelining

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.

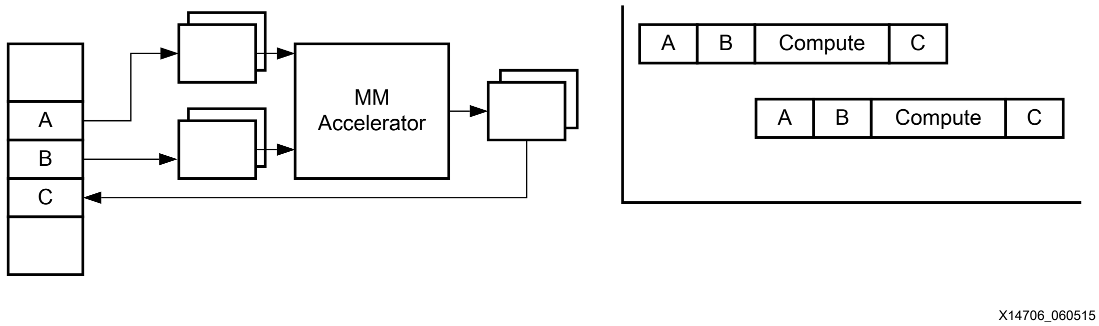
The following figure illustrates the matrix multiply design on the left side and on the right side a time-chart of these events for two successive calls that are executing sequentially.

Figure 2: Sequential Execution of Matrix Multiply Calls



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 3: Pipelined Execution of Matrix Multiply Calls



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:

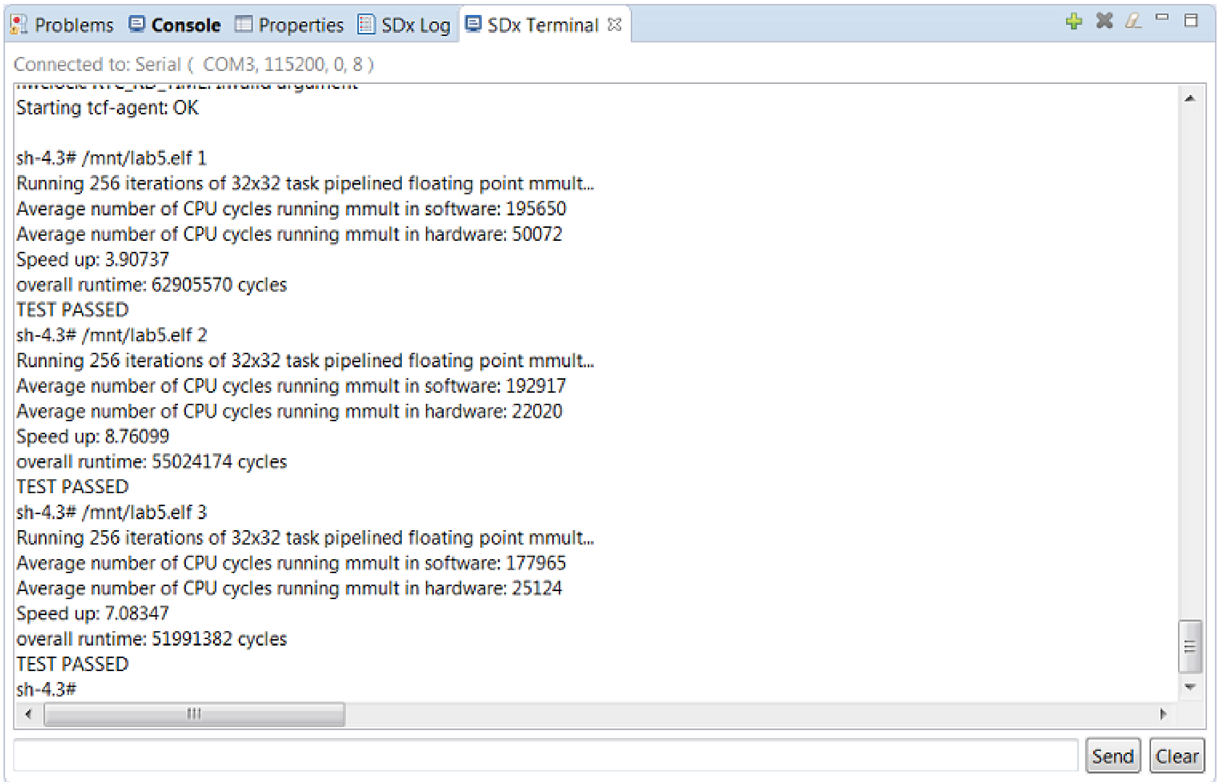
- Use the SDx IDE to optimize your application to reduce runtime by performing task-level pipelining.
- Observe the impact on performance of pipeline calls to an accelerator when overlapping accelerator computation with input and output communication.

Task Pipelining in the Matrix Multiply Example

The SDx IDE 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 SDx project (lab5) by selecting **File→New→Xilinx SDx Project**. Enter the project name `lab5`, select the **ZC702** Platform and **Linux SMP (Zynq-7000)** System Configuration, 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>/SDx/2016.3/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 (for example `./lab5/src`).
4. Click on **lab5** in SDx and from the context menu select **Refresh**. This adds all the copied sources in the previous step to the project.
5. Change the build configuration to **Release**.
6. Mark the function `mmult_accel` in the file `mmult_accel.cpp` for hardware using the **Add HW Functions..** icon in the **SDx Project Settings** or **Toggle HW/SW** in the **Project Explorer**.
7. Build the project.

8. 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.



Chapter 6

Debugging

Lab 6: Debug

This tutorial demonstrates how to use the interactive debugger in the SDx IDE.

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

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 even if you do not have a ZC702 board. When creating the SDx 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.*

Learning Objectives

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

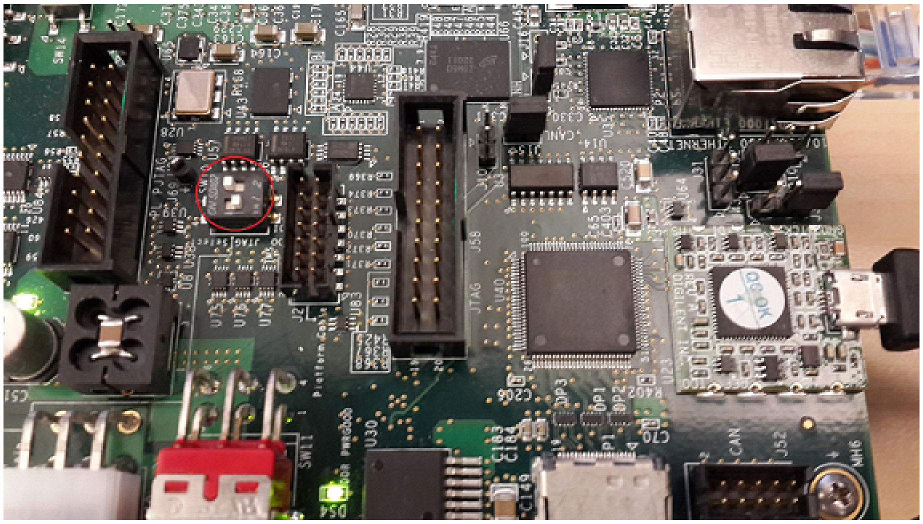
- Use the SDx IDE to download and run your standalone application.
- Optionally step through your source code in the SDx IDE (debug mode) and observe various registers and memories. Note that this is limited to code running on the ARM A9, and does not apply to code that has been converted into hardware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

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.
4. Power on the board.

   Ensure that you allow Windows to install the USB-UART driver and the Digilent driver to enable the SDx IDE 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 SDx™ project (lab6) for the ZC702 platform and Standalone OS using the design template for Matrix Multiplication and Addition.

To create a standalone project in the SDx IDE:

1. Launch the SDx IDE.
2. Select **File→New→Xilinx SDx Project.**
3. Specify the name of the project (for example, `lab6`) in the **Project name** field. Click **Next**.
4. From the **Platform** list, select **zc702**. Click **Next**.
5. From the **System Configuration** drop-down list, select **Standalone OS (Zynq 7000)**. Click **Next**.
6. From the list of application templates, select **Matrix Multiplication and Addition** and click **Finish**.
7. Click on the tab labeled **lab6** to select the **SDx Project Settings** (if the tab is not visible, double click the `project.sdx` file in the **Project Explorer**) and in the **HW functions** panel, observe that the `mmult` and `madd` functions were marked as hardware functions when the project was created.

8. If hardware functions were removed or not marked, you would click on the **Add HW Functions** 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.
9. 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.

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 Debug folder in the **lab6** 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 SDx IDE. Note that the debugger resets the system, programs and initializes the device, then breaks at the `main` function. The source code is shown in the center panel, local variables in the top right corner panel and the SDx log at the bottom right panel shows the Debug configuration log.

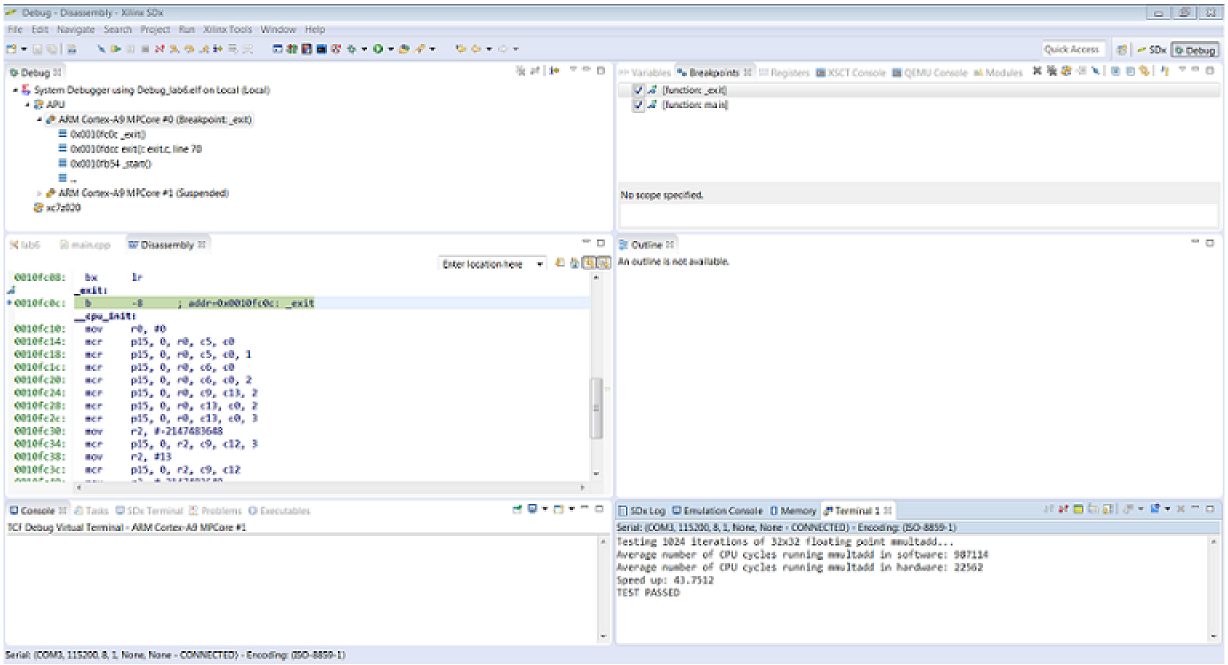
3. Before you start running your application you need to connect a serial terminal to the board so you can see the output from your program. In this example, we are using the SDSoC environment Terminal view invoked by **Window→Show View→Other** and selecting **Terminal→Terminal.** Click the **Terminal** tab near the bottom of the **Debug** perspective and then click the **Connect** icon to connect the terminal to the board (which should be powered up already). Use the following settings: (**Connection Type**: Serial, **Port**: COM<n>, **Baud Rate**: 115200 baud).

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 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. Continuing in lab6, right-click debug hierarchy in the Debug view (System Debugger using `Debug_lab6.elf`), 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.

Debugging Linux Applications

To debug a Linux application in the SDSoC environment:

1. Create a project, for example lab6_linux, targeted to the **Platform** ZC702 and the **System Configuration Linux SMP (Zynq 7000)**. From the list of application templates, select **Matrix Multiplication and Addition**.

   For details, see Creating a New Project.

2. Observe that the functions `mmult` and `madd` are marked for hardware implementation in the **HW functions** table of the **SDx Project Settings**.

   For details, see Marking Functions for Hardware Implementation.

3. Build a project and generate executable, bitstream, and SD card boot image. For the Active build configuration, use **Debug**.

   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* `lab6_linux.zip` *file provided in the project files folder (*`<path to install>/SDx/2016.3/docs/labs/lab6_linux.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 **lab6_linux** project and double-click* `project.sdx` *to display the **SDx Project Settings**. In the **Options** panel, uncheck the **Generate bitstream** box and leave the **Generate SD card image** box checked. Clean the project (right click on **lab6_linux** and select **Clean Project**) and rebuild it (right click on **lab6_linux** 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.

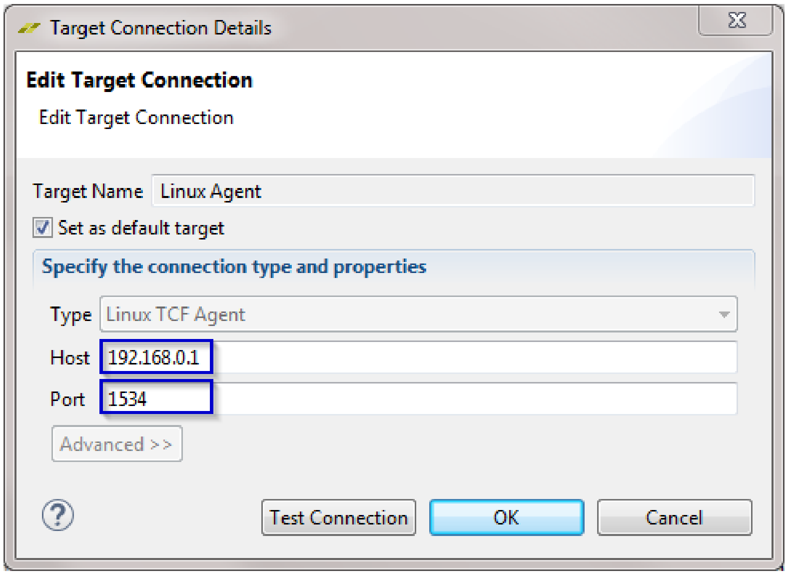
5. Copy the contents of the generated `sd_card` directory to an SD card, and plug the SD card into the ZC702 board.

6. Ensure that the board is connected to your computer via 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. When you see the terminal prompt, set the IP address by entering `ifconfig eth0 192.168.0.1`. Your computer must be configured so the Ethernet adapter is on the same subnetwork as the ZC702 board. On a Windows host system, open **Control Panel\Network and Internet\Network Connections**, and double-click to open the **Local Connection** for the Ethernet Adapter. In the **Networking** tab, select **Internet Protocol Version 4 (TCP/IPv4)**, and click on the **Properties** button. On the **General** tab, select **Use the Following IP Address** and enter `192.168.0.11`. Click **OK**.

   If your subnetwork already has a device at `192.168.0.11`, you can choose another address, as long as it begins with `192.168.0.x`.

7. Back in the SDSoC environment in the **Target Connections** panel, 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.*

Lab 7: Hardware Debug

This lab 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 SDx IDE 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 even if you do not have a ZC702 board. When creating the SDx environment project, select your board and one of the available templates, 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.*

Tracing a Standalone or Bare-Metal Project

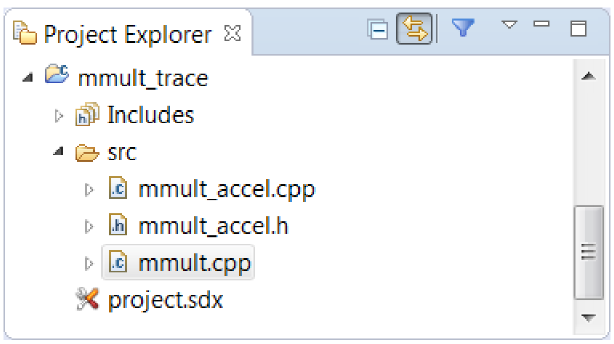
You can learn how to create a new project, configure the project to enable the SDSoC trace feature, build the project, and run the application on the board.

Creating a New Project

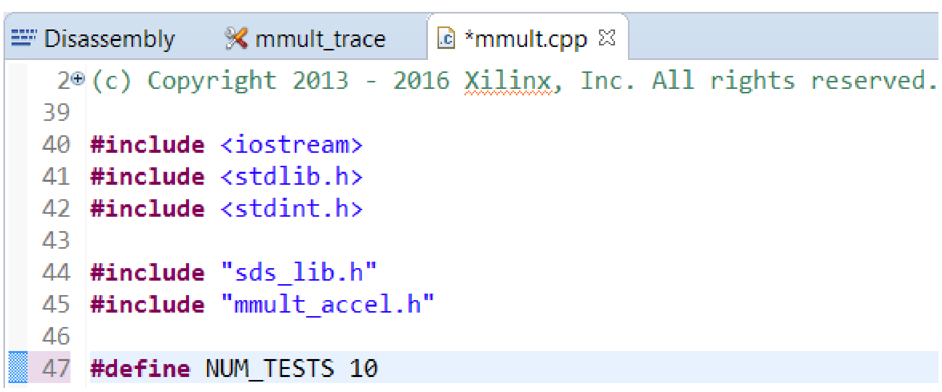
1. Select **File→New→Xilinx SDx Project.**
2. In the New Project wizard, name the project `mmult_trace` and click **Next**.
3. In the Choose Hardware Platform page, select **zc702** and click **Next**.

   NOTE: *Select the appropriate platform if you are using something other than the ZC702 board.*

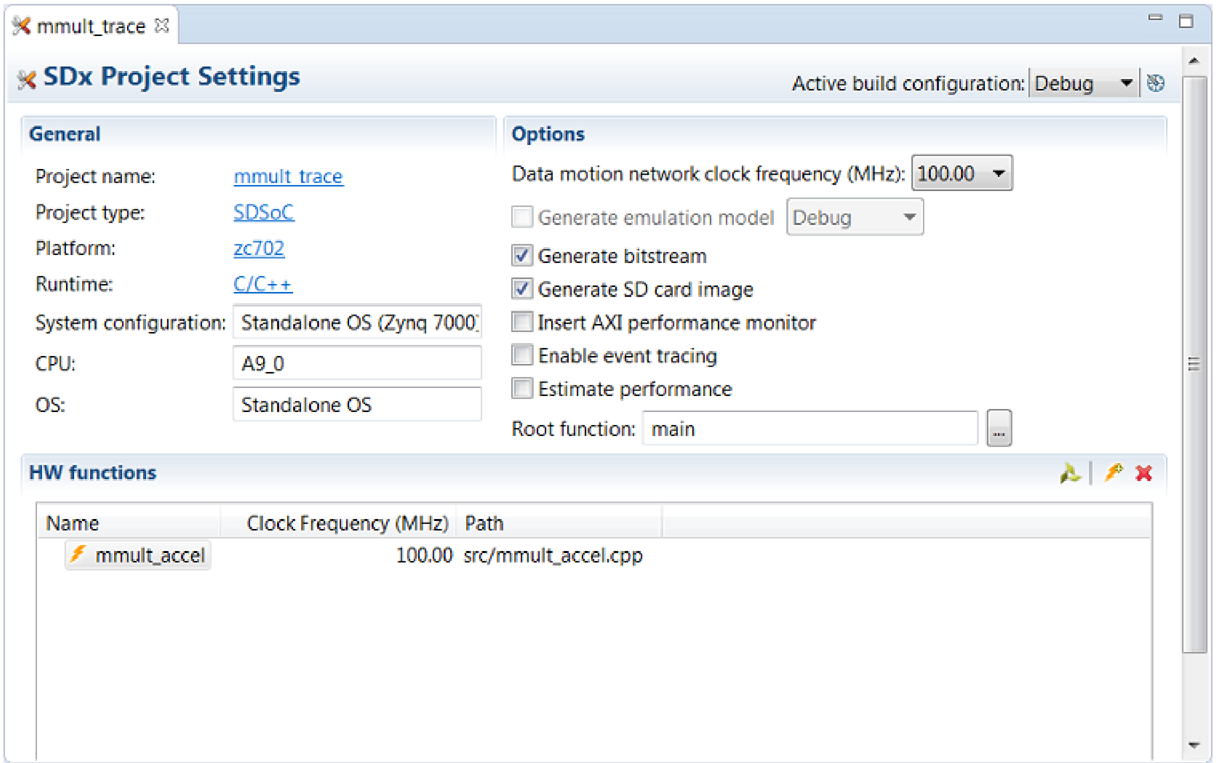
4. Select **Standalone OS (Zynq 7000)** as the System Configuration.
5. Select **Matrix Multiplication** as the template for this project and click **Finish**.
6. In the Project Explorer, expand the various folders by clicking on the triangle ▷ , then open the `mmult.cpp` file.



7. Change the number of tests symbol **NUM_TESTS** from 1024 to **10**, then save and close the file.

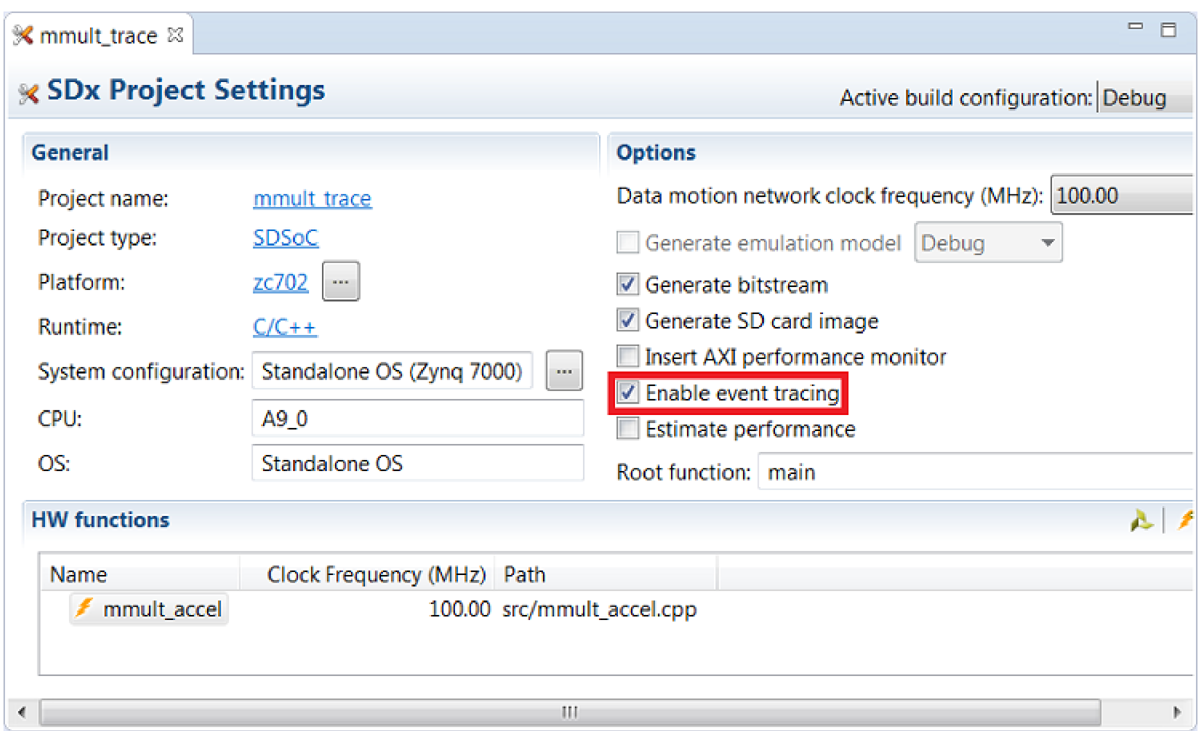


8. In the SDx Project Settings (in the **mmult_trace** tab), notice that **mmult_accel** in the HW Functions section of the project overview is already marked for implementation in hardware.



Configuring the Project to Enable the Trace Feature in the Options Section

1. In the Project Settings window, click the checkbox for **Enable event tracing**.

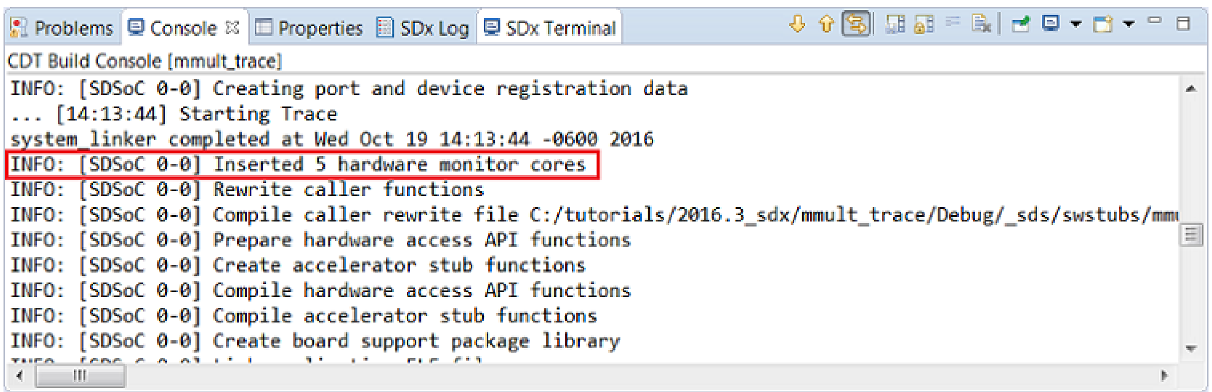


Building the Project

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

IMPORTANT: *The build process might take approximately 15 to 20 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 SDx 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>/SDx/2016.3/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.



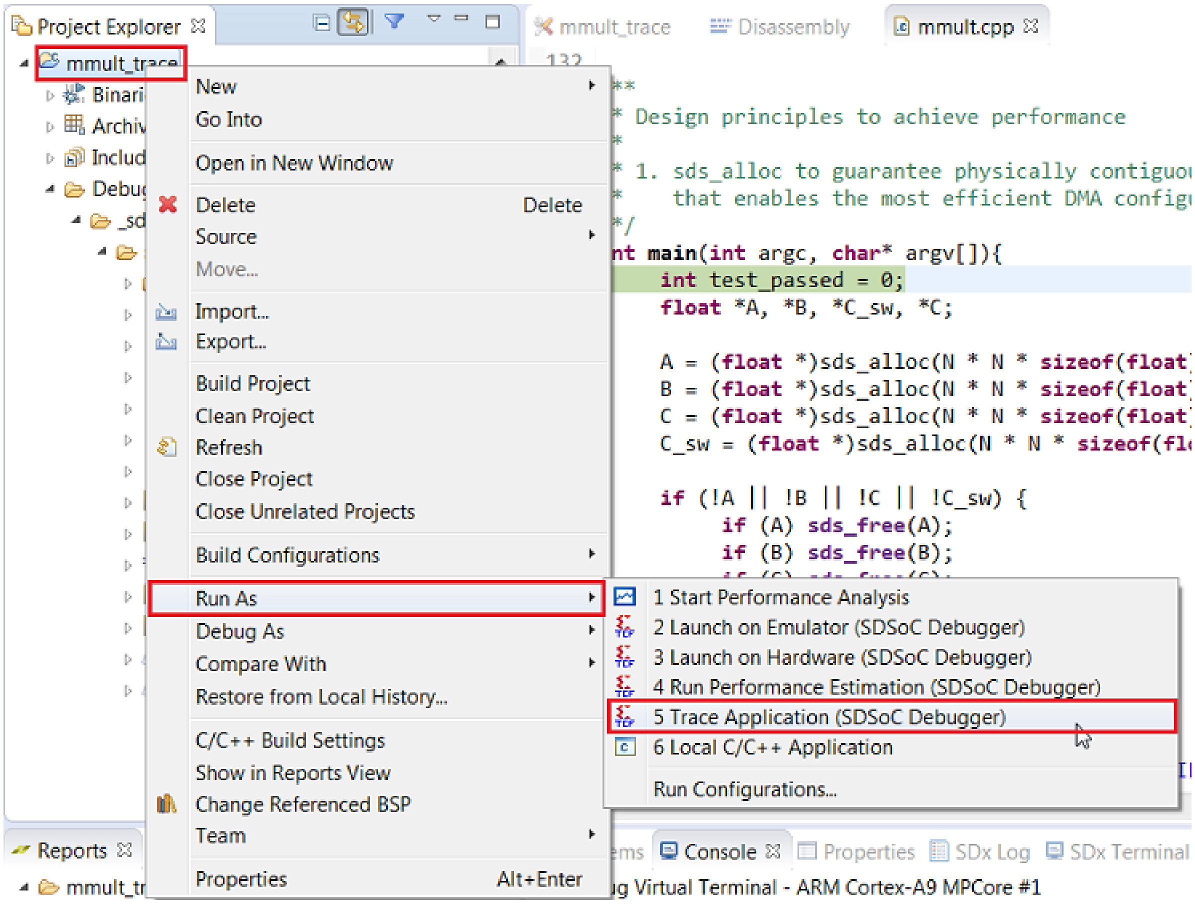
Running the Application on the Board

1. When the build is finished, right-click on the project in the Project Explorer and select **Run As→Trace Application (SDSoC Debugger).**

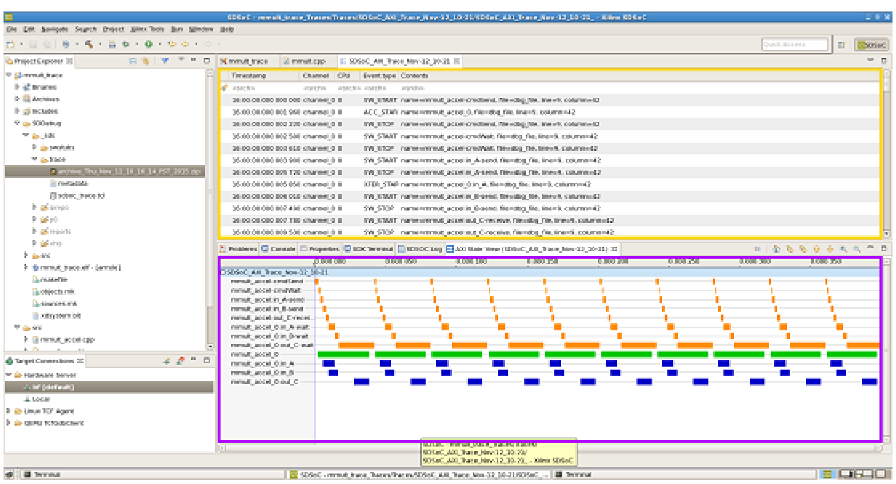
   NOTE: *Be sure not to select Debug As because it will enable breakpoints. If your program breakpoints during execution, the timing will not be accurate (because the software will stop, the hardware will continue running, and the trace timer used for timestamping will continue to run).*

   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.*

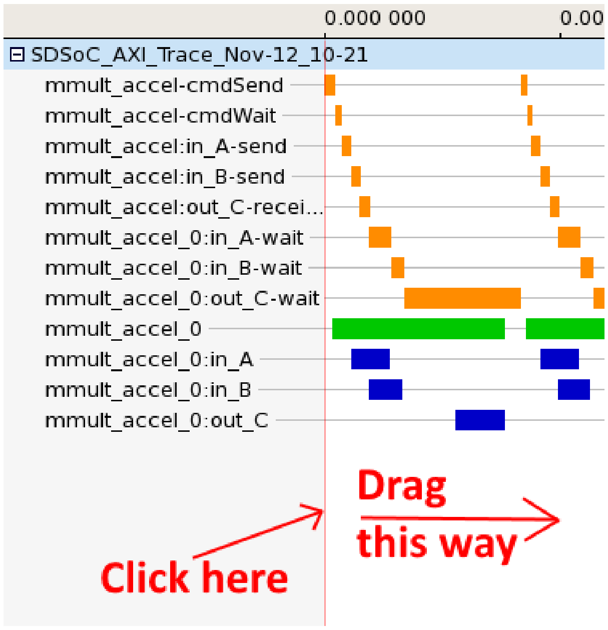


2. After the application exits, and all trace data is collected and displayed, you will see two main areas in the trace visualization: the event textual listing on top (yellow highlighted border), and the event timeline on the bottom (purple highlighted border). Both areas display the same information. 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).

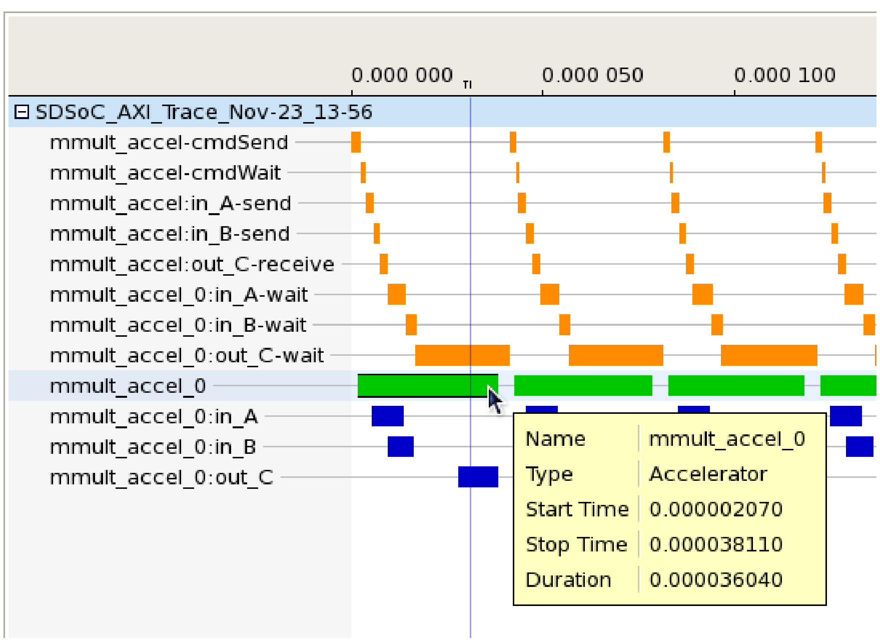


The first thing you should notice is that the 10 iterations of the application are clearly visible as repeated groups of events. Orange events are software events, green events are accelerator events, and blue events are data transfer events.

3. 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.



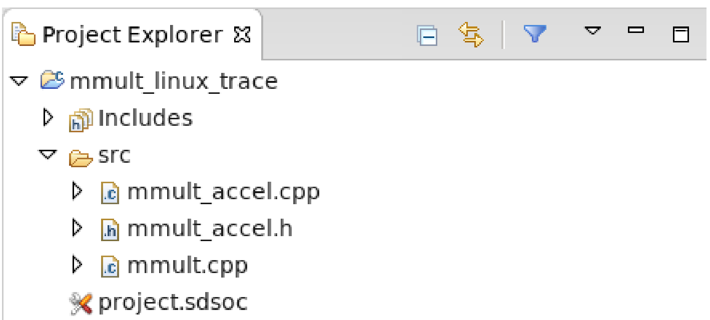
4. If you hover the cursor over one of the events, you will see a detailed tool-tip appear displaying the detailed information about each trace. The example below shows the first accelerator event, which corresponds to the start/stop of the `mmult_accel` function that we chose to implement in hardware (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).



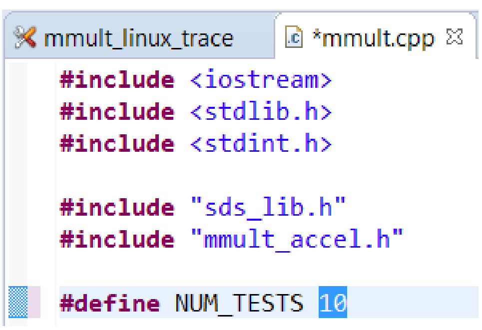
Tracing a Linux Project

You can learn how to create a new project, configure the project to enable the SDx trace feature, build the project, run the application on the board, and view the trace data.

1. Create a new project.
   a. Select **File→New→Xilinx SDx Project.**
   b. In the New Project wizard, name the project `mmult_linux_trace` and click **Next**.
   c. Select **zc702** as the Hardware Platform. Click **Next**.
   d. For System configuration select**Linux SMP (Zynq 7000)**.
   e. Click **Next**.
   f. Select **Matrix Multiplication** as the template for this project and click **Finish**.
   g. In the Project Explorer, expand the various folders by clicking on the triangle , then open the `mmult.cpp` file under the `src` folder.



   h. Change the number of tests symbol **NUM_TESTS** from 1024 to **10**, then save and close the file.



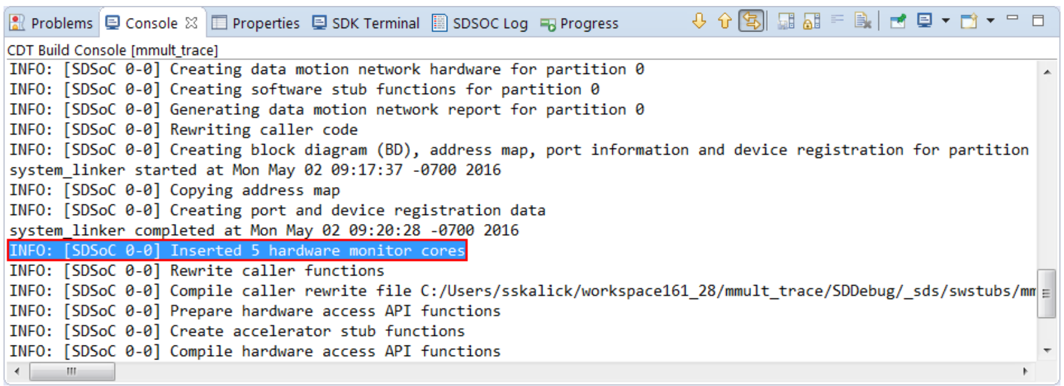
   i. In the SDx Project Overview (in the **mmult_linux_trace** tab), notice that the **mmult_accel** in the HW Functions section of the project overview is already marked for implementation in hardware.
2. Configure the project to enable the Trace feature in the SDx IDE.
   a. In the Project Overview window, click the checkbox for **Enable Event Tracing** under the Options section.

3. Build the project.

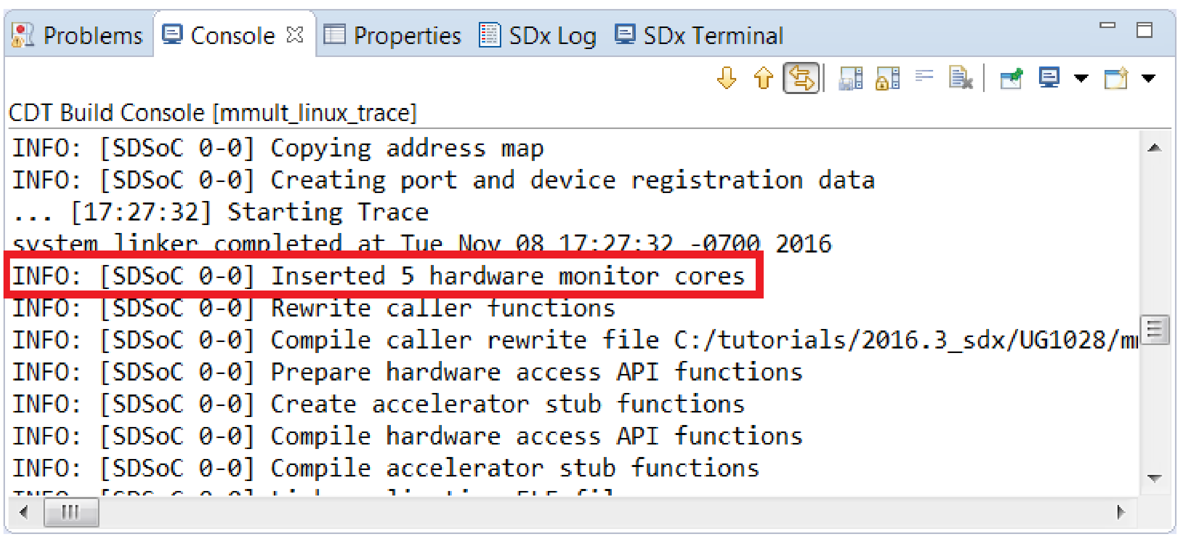
    a. Click the **Build** button to start building the project. (This will take a while.)

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 SDx 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/2016.3/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.



X16927-050316



4. Run the application on the board.

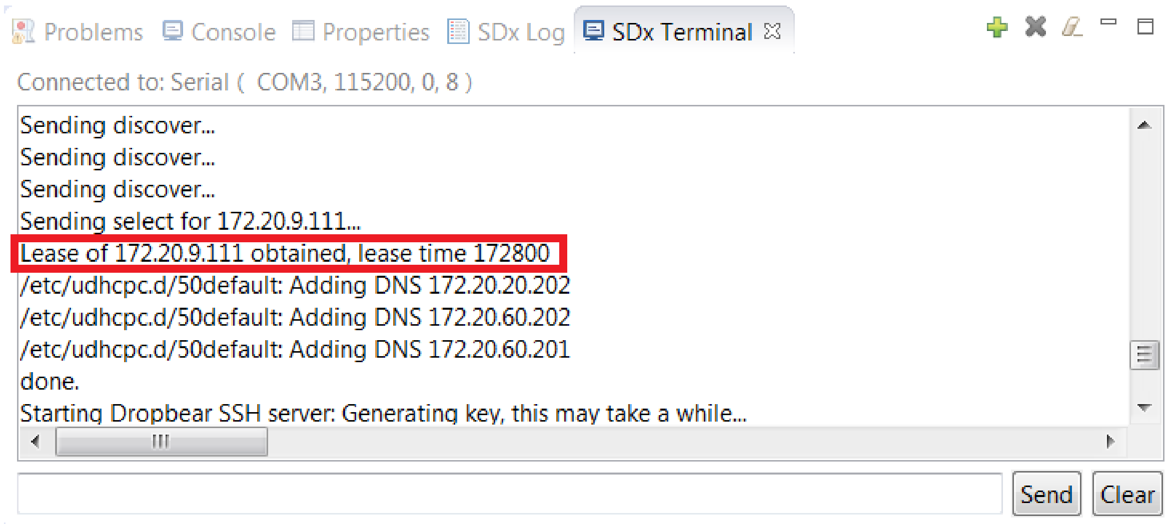
    a. When the build is finished, copy the files in the `sd_card` directory onto an SD card and insert into the SD card socket on the board.

    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 by clicking the + button on the SDx Terminal tab.

d. Connect the USB/JTAG port to the PC and boot Linux on the board.

e. Check the IP address of the zc702 board by looking at the SDx Terminal log.



f. From the **Target Connections** view, set up the **Linux TCF Agent** in the same manner as in Using the Performance Estimation Flow With Linux.

g. 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.*

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.

Appendix A

Additional Resources and Legal Notices

Xilinx Resources

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

Solution Centers

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

References

These documents provide supplemental material useful with this guide:

1. *SDx Environments Release Notes, Installation, and Licensing Guide* (UG1238)
2. *SDSoC Environment User Guide* (UG1027)
3. *SDSoC Environment Optimization Guide* (UG1235)
4. *SDSoC Environment Tutorial: Introduction* (UG1028)
5. *SDSoC Environment Platform Development Guide* (UG1146)
6. *SDSoC Environment Tutorial: Creating a Platform from a Reference Design* (UG1236)
7. SDSoC Development Environment web page
8. *UltraFast Embedded Design Methodology Guide* (UG1046)
9. *ZC702 Evaluation Board for the Zynq-7000 XC7Z020 All Programmable SoC User Guide* (UG850)
10. *Vivado Design Suite User Guide: High-Level Synthesis* (UG902)
11. *PetaLinux Tools Documentation: Workflow Tutorial* (UG1156)
12. Vivado® Design Suite Documentation
13. *Vivado Design Suite User Guide: Creating and Packaging Custom IP* (UG1118)

Please Read: Important Legal Notices

AUTOMOTIVE APPLICATIONS DISCLAIMER