SDSoC Environment User Guide

UG1027 (v2018.2) July 2, 2018

XILINX

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
07/02/2018 Version 2018.2	
Design Flow Overview	Updated Design Flow Overview diagram.
Throughout the document:	Minor text updates. Updated Figures.
Chapter 1: SDSoC Introduction and Overview	Updated content throughout chapter.
Chapter 2: The SDSoC Environment	Updated content throughout chapter. Updated the PS/PL Block Diagram. Updated Flow Diagram.
Chapter 3: The SDSoC Environment	Updates in all sections. Design Flow Overview diagram was updated to include Emulation.
Chapter 4: C-Callable IP Libraries	Updates throughout chapter.
06/06/2018 Version 2018.2	
Throughout the document:	Updated Figures.
SDSoC Introduction and Overview	Added Chapter.
The SDSoC Environment	Updated Chapter. • Added introductory information. • Modified the PS/PL C-Callable block diagram. • Added Pipelined Data Transfer and Compute figure. • Added information and links to topics in the Chapter.
	Design Flow Overview
	Working with the SDx GUI
	Using an SDx Workspace
Creating an SDSoC Application	Changed Chapter title. Made extensive changes to all sections.
	Using a Workspace: Consolidated information about environment variables and command shells.
	Creating an Application Project
	Working with Code: Updated all topics in this section.
Building the SDSoC Project	• Creating a Hardware Project: Separated topic. • Running Emulation: Separated figures for QEMU interface and Waveform window. • Guidelines for Invoking SDSCC/SDS++:Moved topic from Command Line Options Chapter; that chapter was deleted.

Section	Revision Summary
Targeting System Emulation	Updated topic.
Chapter 4: C-Callable IP Libraries	Updated C-Callable Libraries Chapter.
Debugging Techniques	Added Chapter.
Chapter 6: Profiling and Optimization	Updated Chapter.
Appendix A: Getting Started with Examples	Modified examples.
Managing Platforms and Repositories	Updated Appendix.
Compiling and Running Applications	Added Appendix.
04/04/2018 Version 2018.1	
The SDSoC Environment	Updated information to more correctly reflect the SDSoC Environment.
Hardware/Software System Runtime Environment	Updated code examples.
Design Flow Overview	Updated content to more correctly reflect the behavior of SDSoC.
Creating and Using C-Callable IP Libraries.	Minor edits to C-Callable IP contents.

Table of Contents

Chapter 1

SDSoC Introduction and Overview

The SDSoC™ environment provides a framework for developing and delivering hardware accelerated embedded processor applications using standard programming languages. It includes a familiar embedded processor development flow with an Eclipse-based integrated development environment (IDE), compilers for the embedded processor application and for hardware functions implemented on the programmable logic resources of the Xilinx® device. The `sdscc/sds++` (referred to as `sds++`) system compiler analyzes a program to determine the data flow between software and hardware functions, generating an application-specific system on-chip supporting bare metal, Linux, and FreeRTOS as the target operating system. The `sds++` system compiler generates hardware IP and software control code that automatically implements data transfers and synchronizes hardware accelerators and application software, pipelining communication and computation.

Using SoC devices from Xilinx, such as the Zynq-7000 SoC and the Zynq UltraScale+ MPSoC, you can implement elements of your application into hardware accelerators, running many times faster than optimized code running on a processor. Xilinx FPGAs and SoC devices offer many advantages over traditional CPU/GPU acceleration, including a custom architecture capable of implementing any function that can run on a processor, resulting in better performance at lower power dissipation. To realize the advantages of software acceleration on a Xilinx device, you should look to accelerate large compute intensive portions of your application in hardware. Implementing these functions in custom hardware lets you achieve an ideal balance between performance and power. The SDSoC environment provides tools and reports to profile the performance of your embedded processor application, and determine where the opportunities for acceleration are. The tools also provide automated run-time instrumentation of cache, memory and bus utilization to track real-time performance on the hardware.

Developers of hardware accelerated applications can make use of a familiar software-centric programming work flow to take advantage of FPGA acceleration with little or no prior FPGA or hardware design experience. As a software programmer, calling a hardware function is the same as calling a software function, letting the compiler implement the hardware/software partitioning. However, developers can also create predefined hardware accelerators for use in an embedded processor application, using a hardware-centric approach working through the Vivado® HLS compiler, or creating and packaging optimized RTL accelerators for distribution as a library of C-Callable IP.

The SDSoC environment provides predefined platforms for standard ZCU102, ZCU106, ZCU104, ZC706 and ZC702, Zynq®-based development boards. Third-party platforms are also available including: the Zedboard, Microzed, Zybo, Avnet Embedded Vision Kit, Video and Imaging Kit, SDR kit and more. You can also create a custom platform to meet your specific market requirements. An SDSoC platform consists of a hardware portion defining the embedded processor, the hardware function, and any peripherals supported by the platform; and a software portion defining the operating system boot images, drivers, and the application code. You can start your project using one of the standard SDSoC platforms to evaluate a design concept, to be later implemented on an SDSoC custom platform for production.

Software Acceleration with SDSoC

When compared with processor architectures, the structures that comprise the programmable logic (PL) fabric in a Xilinx® device enable a high degree of parallelism in application execution. The custom processing architecture generated by the `sdscc/sds++` (referred to as `sds++`) for a hardware function in an accelerator presents a different execution paradigm from CPU execution, and provides opportunity for significant performance gains. While you can retarget an existing embedded processor application for acceleration on programmable logic, writing your application to use libraries of existing hardware functions such as modifying your code to better use the device architecture or using the Xilinx xfOpenCV library yields significant performance gains and power reduction.

CPUs have fixed resources and offer limited opportunities for parallelization of tasks or operations. A processor, regardless of its type, executes a program as a sequence of instructions generated by processor compiler tools, which transform an algorithm expressed in C/C++ into assembly language constructs that are native to the target processor. Even a simple operation, like the addition of two values, results in multiple assembly instructions that must be executed across multiple clock cycles. This is why software engineers restructure their algorithms to increase the cache hit rate and decrease the processor cycles used per instruction.

An FPGA is an inherently parallel processing device capable of implementing any function that can run on a processor. Xilinx SoC devices have an abundance of resources that can be programmed and configured to implement any custom architecture and achieve virtually any level of parallelism. Unlike a processor, where all computations share the same ALU, the FPGA programming fabric acts as a blank canvas to define and implement your acceleration functions. The FPGA compiler creates a unique circuit optimized for each application or algorithm; for example, only implementing multiply and accumulate hardware for a neural net - not a whole ALU.

The `sds++` system compiler exercises the capabilities of the FPGA fabric through the automatic insertion of data movers and accelerator control IP, creation of accelerator pipelines, and invoking the Vivado®HLS tool, which employs processes of scheduling, pipelining, and dataflow:

- Scheduling: The process of identifying the data and control dependencies between different operations to determine when each will execute. The compiler analyzes dependencies between adjacent operations as well as across time and groups operations to execute in the same clock cycle when possible, or to overlap the function calls as permitted by the dataflow dependencies.

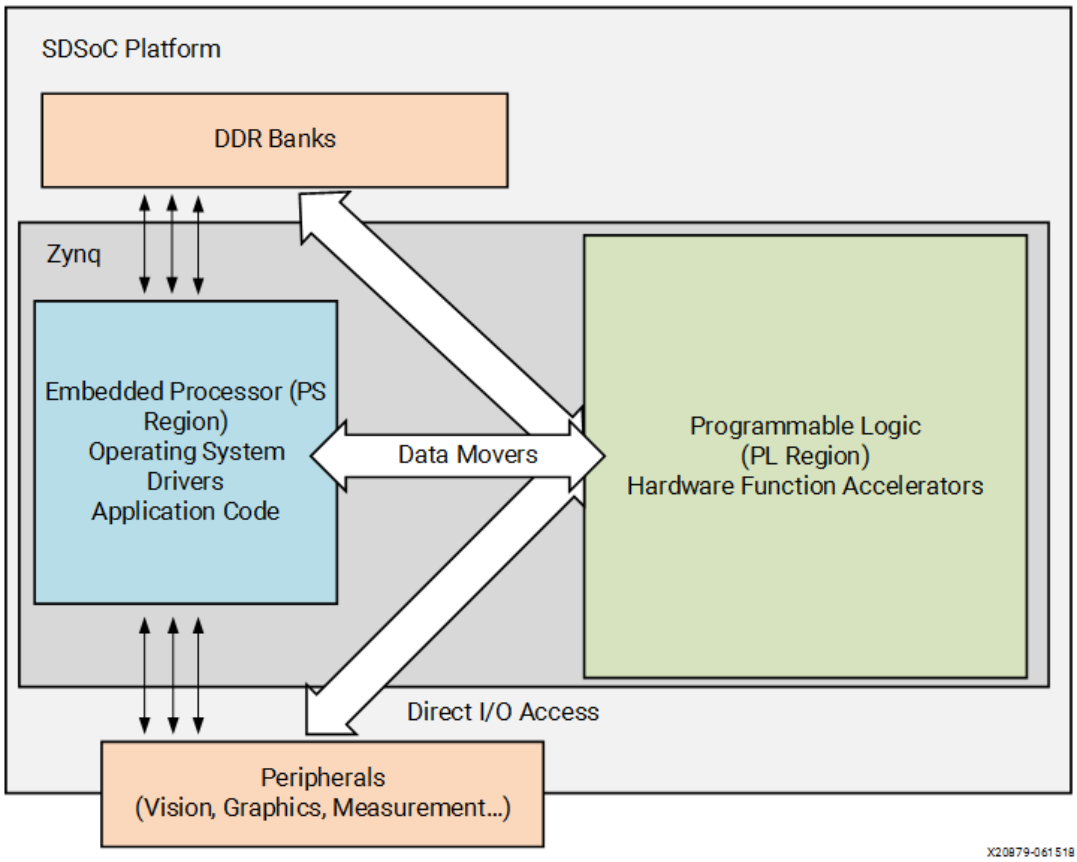
- Pipelining: A technique to increase instruction-level parallelism in the hardware implementation of an algorithm by overlapping independent stages of operations or functions. The data dependence in the original software implementation is preserved for functional equivalence, but the required circuit is divided into a chain of independent stages. All stages in the chain run in parallel on the same clock cycle. Pipelining is a fine-grain optimization that eliminates CPU restrictions requiring the current function call or operation to fully complete before the next can begin.
- Dataflow: Enables multiple functions implemented in the FPGA to execute in a parallel and pipelined manner instead of sequentially, implementing task-level parallelism. The compiler extracts this level of parallelism by evaluating the interactions between different functions of a program based on their inputs and outputs.

Execution Model of an SDSoC Application

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

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

Figure 1: **Architecture of an SDSoC System**



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

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

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

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

2. Within a program, every call to a hardware function is intercepted by a function call into a stub function with the same function signature (other than name) as the original function. Within the stub function, the following steps occur:
   a. A synchronous accelerator task control command is sent to the hardware.
   b. For each argument to the hardware function, an asynchronous data transfer request is sent to the appropriate data mover, with an associated `wait()` handle. A non-void return value is treated as an implicit output scalar argument.
   c. A barrier `wait()` is issued for each transfer request. If a data transfer between accelerators is implemented as a direct hardware stream, the barrier `wait()` for this transfer occurs in the stub function for the last in the chain of accelerator functions for this argument.
3. Cleanup of the `sds_lib` library occurs during the program's destructor upon exiting `main()`.

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

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

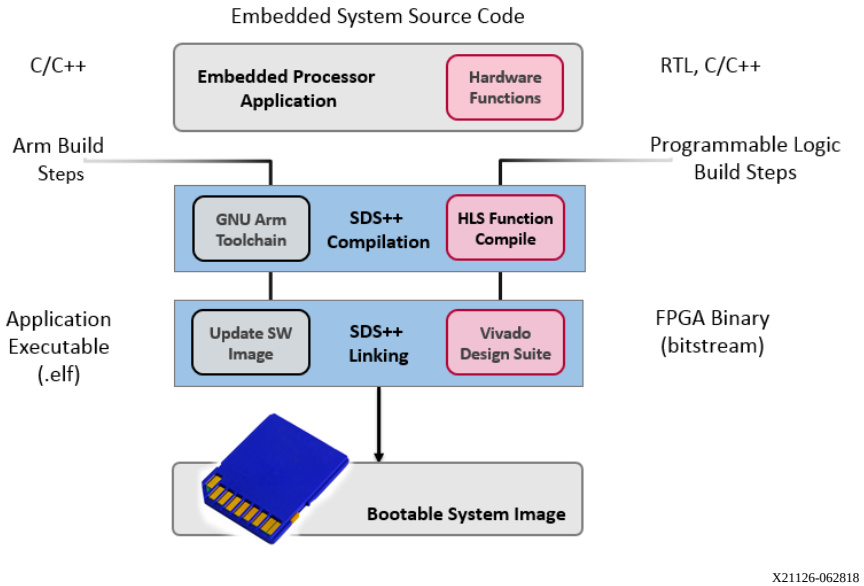
IMPORTANT!: *Every `async(ID)` pragma requires a matching `wait(ID)` pragma.*

SDSoC Build Process

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

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

Figure 2: **SDSoC Build Process**



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

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

Build Targets

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

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

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

SDSoC Development Methodologies

The SDSoC™ environment supports two primary use cases:

- Software-centric design: The development of an accelerated application written by software programmers using standard programming languages, accelerating compute intensive functions into programmable logic, or identifying application bottlenecks for acceleration by profiling the application.
- Hardware-centric design: The development of predefined accelerated functions for use in embedded processor applications like a library of intrinsic functions. This design methodology can be driven from a top-down approach of writing the hardware function in a standard programming language like C or C++, and then synthesized into RTL for implementation into programmable logic; or by using standard RTL design techniques to create and optimize the accelerated function.

The two use-cases are often combined, letting teams of software and hardware developers define hardware accelerators and developing embedded processor applications to use them. This combined methodology involves different components of the application, developed by different people, and potentially from different companies. You can use predefined hardware functions from libraries available for use in your accelerated application, such as the Xilinx® xfOpenCV libraries, or develop all the accelerators within your own team.

Software-Centric Design

The software-centric approach to accelerated application development, or accelerator development, begins with the use of the C or C++ programming language. The code is written as a standard software program, with some attention to the specific architecture of the code. The software-centric development flow typically uses the following steps:

Table 1: **Software-Centric Design Flow**

Task	Steps
Profile the embedded processor application.	• Baseline the performance, identify bottlenecks and functions to accelerate. • Assess acceleration potential, plan budgets and requirements.
Code the desired accelerators.	• Convert the desired functions to define the hardware function code without optimization.

Table 1: **Software-Centric Design Flow** *(cont'd)*

Task	Steps
Verify functionality, iterate as needed.	• Run system emulation to generate application and accelerator profiling data including: ◦ Estimated FPGA resource usage. ◦ Overall application performance. ◦ Visual timeline showing application calls and accelerator start/stop times. • Address design recommendations provided by tool guidance.
Optimize for performance, iterate as needed.	• Analyze the profile summary, and application timeline. • Optimize data movement throughout system: ◦ Application to DDR, DDR to accelerator, hardware function interface to local buffers (bursting) ◦ Maximize DDR bandwidth usage with efficient transfer sizes ◦ Overlapping of transfers ◦ Prefetching • Optimize the accelerator code for performance: ◦ Task-level parallelism (dataflow) ◦ Instruction-level parallelism (pipelining, and loop unrolling) ◦ Match datapath size to interface bandwidth (arbitrary bit-width)

Hardware-Centric Design

A hardware-centric flow first focuses on developing and optimizing the accelerators and typically leverages advanced FPGA design techniques to create a library of C-Callable IP. This begins with the definition of the hardware function in C or C++ for use in Vivado® HLS, or the use of an RTL language, or an existing IP design or block design in the Vivado® Design Suite. The hardware function is defined in RTL code, and synthesized and implemented into the programmable logic of the target device. A software function signature is needed to use the C-Callable IP in the accelerator application, or a compiled library of functions is created for use across multiple applications. The hardware-centric development flow typically uses the following steps:

Table 2: **Hardware-Centric Design Methodology**

Task	Steps
Study the SDSoC™ platform specification, and the Zynq® device specification and programming model.	• Hardware platform, software platform, data movers, AXI interface, DDR.
Identify cycle budgets and performance requirements.	
Define the accelerator architecture and interfaces.	
Develop the accelerator.	• Use Vivado® HLS for C or C++ hardware functions. • Use traditional RTL design techniques in the Vivado® Design Suite.
Verify functionality and performance, iterate as needed.	• Run hardware/software co-simulation in Vivado HLS. • Run logic simulation in the Vivado simulator.
Optimize the quality of results to reduce resource utilization and increase frequency, iterate as needed.	• For HLS, make sure the design is DRC clean. • Run the Vivado implementation flow, using the techniques specified in the *UltraFast Design Methodology Guide for the Vivado Design Suite* (UG949) • Use best practices for out-of-context synthesis and estimation.
Import the C-Callable IP into the SDSoC environment	• For the HLS flow import the C or C++ code into your SDSoC project. • For RTL flow use the C-Callable IP wizard. • See C-Callable Libraries for more information.
Develop sample application code to test the hardware function.	• Test sample applications with a dummy function having the same interfaces as the C-Callable IP. See C-Callable Libraries for more information.
Verify the hardware function works properly with application, iterate as needed.	• Use system emulation for debug. • Use the Hardware debug methodology for complex internal debug problems.
Optimize host code for performance, iterate as needed:	• Use the Profile Summary report, the Activity Timeline, and event timers in the host application to measure performance. • Make sure the design rules check (DRC) is clean. • Work to achieve an Activity Timeline that matches the desired performance. • Techniques: Overlapping transactions, out-of-order (OOO) synthesis queues, sub-devices.

Table 2: **Hardware-Centric Design Methodology** *(cont'd)*

Task	Steps
Finalize the Software Acceleration Layer deliverable (API, share lib, plug-in...)	

Best Practices for Acceleration with SDSoC

Some specific things to keep in mind when developing your application code and hardware function in the SDSoC™ environment:

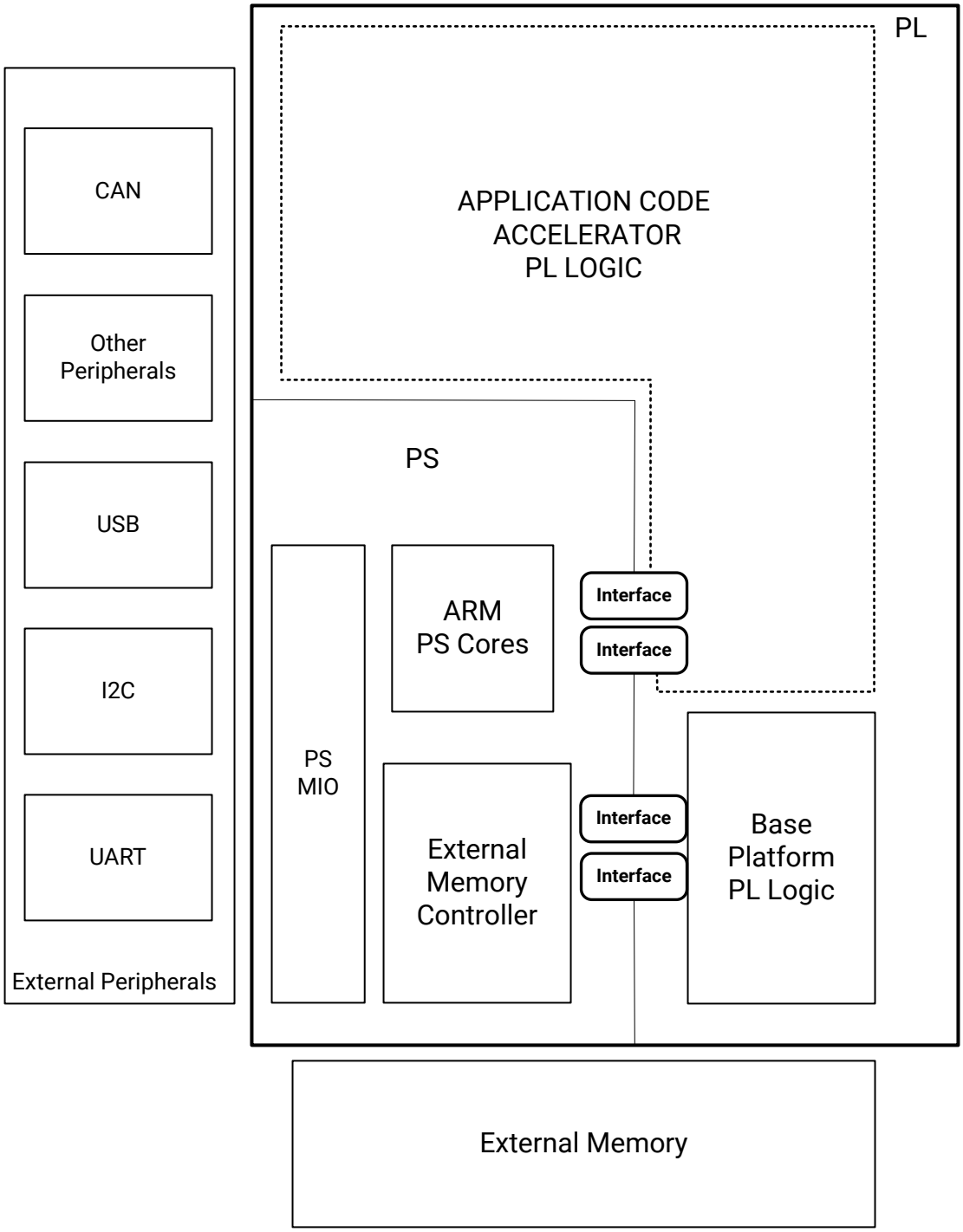
- General guidelines:
  - Reduce resource utilization and improve parallelism by streaming data instead of copying data into the PL region. For example, in an image processing application, stream rows of pixels that make up a frame instead of copying the image frame in one long data transfer.
  - Reuse the data local to the PL region rather than transferring it back and forth to limit DMA.
  - Look to accelerate functions that have:
    - A high compute time to data transfer time ratio.
    - Predictable communication streams.
    - Self contained control structure – not needing control logic outside the accelerator.
  - Look for opportunities to increase task-level parallelization by launching multiple accelerators concurrently, or multiple instances of an accelerator.
- For a software-centric approach:
  - Use good memory management techniques, such as having known array sizes, and using `sds_alloc()`/`sds_free()` to allocate/de-allocate physically contiguous memory, thereby reducing the device footprint and increasing baseline performance.
  - Use system emulation to validate your code frequently to make sure it is functionally correct.
  - Write/migrate hardware functions to separate C/C++ files as to not recompile the entire design for incremental changes. .
- For a hardware-centric approach using C-Callable IP:
  - Keep track of the AXI4 Interface offsets for an IP, or accelerator, and what function definition parameters require what data type. The interfaces need to be byte aligned.
  - Maintain the original Vivado® IP project so that modifications to it can be quickly implemented.
  - Keep the static library (`.a`) file and corresponding header file together.

Chapter 2

The SDSoC Environment

The software-defined system-on-chip (SDSoC™) environment provides the tools necessary for implementing heterogeneous embedded systems using theXilinx® Zynq®devices. The concept of a platform is integral to the SDSoC environment and it defines the hardware, software, and meta-data components on which SDSoC applications are built. Platforms that support Zynq devices combine an Arm®-based processor system (PS) with high performance, user-programmable logic (PL) to develop products that partition the system design across hardware and software.

Figure 3: **Processor System (PS) and Programmable Logic (PL) Block Diagram**

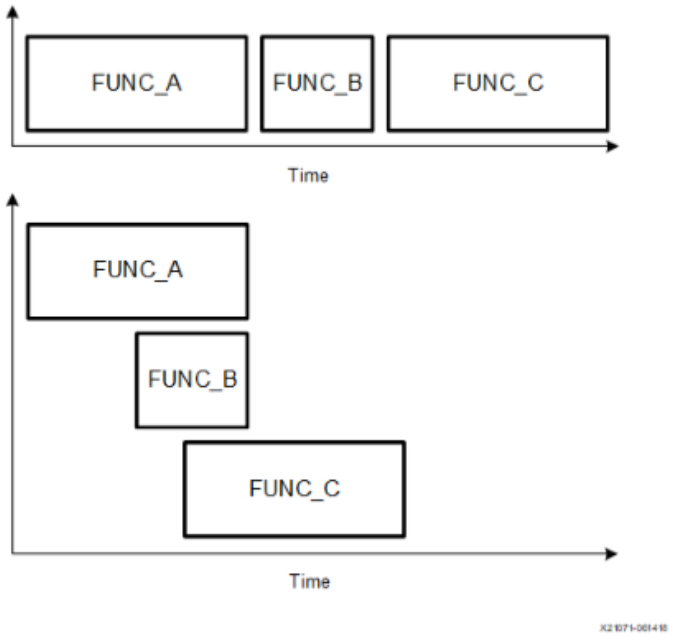


The task of designing, exploring, and selecting a hardware/software partitioning solution is accomplished by working in a software development environment using an Eclipse-based integrated development environment (IDE). The SDx™ IDE uses the `sds++` system compiler to convert C/C++ code into high-performance hardware accelerators. Declarations within the platform meta-data identify interface ports, clocks, interrupts, and reset blocks that are available

for the system compiler to use as it attaches hardware accelerators to the base platform.

The system compiler analyzes a program to determine the data flow between software and hardware functions and generates an application-specific system-on-chip. The `sds++`-generated hardware IP and software control code automatically implements data transfers and synchronizes the hardware accelerators with application software. Performance is achieved by pipelining communication and computation, thereby producing hardware functions that can run with maximum parallelism as illustrated in the following figure.

Figure 4: **Pipelined Data Transfer and Compute**



The `sds++` system compiler invokes the Vivado® High Level Synthesis (HLS) tool to transform software functions into a bitstream that defines and configures the programmable logic (PL) portion of the SoC. In addition, stub functions are generated so application software compiled and linked using the standard GNU toolchain transparently uses the implemented hardware functions. All necessary drivers and libraries are automatically included in this system compilation process.

The final output of system compilation is the generated `sd_card` directory, which at minimum is populated with a Zynq bootable `BOOT.BIN` file, the executable and linkable format (ELF) file application code, and a `README.txt` boot instructions file. The `BOOT.BIN` file contains any necessary bootloaders, bitstreams, and application code to boot the generated system on a target board. For systems that run Linux on the target board, the `sd_card` directory also contains the Linux image file used during the boot process.

The SDSoC system compilers generate complete applications and let users iterate over design and architectural features by refactoring at the program level, reducing the time necessary to achieve working applications on target platforms. To achieve high performance, each hardware function runs independently; the system compilers generate hardware and software components that ensure synchronization between the hardware functions and the application software while enabling pipelined computation and communication. Application code can involve many hardware functions, multiple instances of a specific hardware function, and calls to a hardware function from different parts of the program.

For SDSoC, the environment reflects the resources and performance available within the Zynq-7000 SoC or the Zynq® UltraScale+™ MPSoC device family. When creating applications that require specific real-time behavior, it is important to be aware of the execution environment.

The Zynq® family includes a processor system (PS) with dedicated Arm® processing cores, on-

chip memories, embedded peripherals, interconnect blocks, a double data rate (DDR) memory controller, and a programmable logic (PL) fabric used by the SDSoC-generated accelerators.

Ideal processor, memory, and AXI interface performance are shown in the following table using switching characteristics from the Zynq family datasheets.

Table 3: **Processor, Memory, and AXI Interface Performance**

Clock or Interface	Zynq UltraScale+ MPSoC	Zynq-7000 SoC
Max APU clock frequency	Arm Cortex-A53 64-bit Quad-Core: 1500 MHz	ARMArm Cortex-A9 32-bit Dual-Core: 1000 MHz
Max RPU clock frequency	ARM ArmCortex-R5 32-bit Dual-Core: 600 MHz	N/A
DDR type and bit width	DDR4: x32, x64	DDR3: x16, x32
DDR Max performance	2400 Mb/s	1333 Mb/s
DDR Max Ideal Throughput	153.6 Gb/s	42.6 Gb/s
AXI Interface width	128-bi[1] 64-bit, 32-bit	64-bit, 32-bit
AXI Interface Max Frequency	333 MHz	250 MHz
AXI Interface Max Ideal Throughput	42.6 Gb/s	16 Gb/s
Number of AXI Interface Ports	12	6
Total AXI Throughput	511.2 Gb/s	96.0 Gb/s

Notes:

1. Current SDSoC™ DMA supports up to 64-bit wide interfaces only.

This document provides descriptions of the following:

- An overview of the system capabilities. See Design Flow Overview.
- How to use the system with the IDE and with command line entries. See Creating an SDSoC Project.
- How to work with code, import source files, select hardware accelerators and clock frequencies. See Working with Code.
- How to build a project and emulate the system. See Creating an Application Project and Running System Emulation.
- How to use C-Callable IP libraries. See C-Callable IP Libraries.
- Debugging, profiling, and optimization techniques. See Debug Techniques and Profiling and Optimization.
- Examples to aid in understanding how to use the system. See Getting Started with Examples.
- How to manage files within the environment. See Managing Platforms and Repositories
- Compiling and running applications for Arm® and MicroBlaze™ processors. See Compiling and Running Applications

Getting Started

Download and install the SDSoC™ Tool Suite according to the directions provided in the *SDSoC Environments Release Notes, Installation, and Licensing Guide* (UG1294).

After installing the SDx tools, you can find detailed instructions and hands-on tutorials to introduce the primary work flows for project creation, specifying functions to run in programmable logic, system compilation, debugging, and performance estimation in the *SDSoC Environment Getting Started Tutorial* (UG1028). Working through the tutorial and its labs is the best way to get an overview of the SDx environment, and should be considered a prerequisite to application development.

***Note*:** The SDx tool suite includes the entire tool stack to create a bitstream, object code, and executables. If you have installed the Xilinx® Vivado® Design Suite and the Software Development Kit (SDK) tools independently, you should not attempt to combine these installations with the SDx tools. Ensure that your tools are derived from an SDx installation (which includes the Vivado Design Suite and SDK tools).

RECOMMENDED: *Although SDSoC supports Linux application development on Windows hosts, a Linux host is strongly recommended for SDSoC platform development, and required for creating a platform supporting a target Linux OS.*

Elements of SDSoC

The SDSoC environment includes the compiler, `sds++`, to generate complete hardware/software systems, an Eclipse-based user interface to create and manage projects and workflows, and a system performance estimation capability to explore different "what if" scenarios for the hardware/software interface. Elements of the SDx™ tools include:

- Eclipse-base IDE
- The `sds++` system compiler
- High Level Synthesis (HLS)
- Vivado® Design Suite
- IP integrator and IP libraries
- Vivado-generated SDx Platforms
- SDx-generated hardware accelerators and associated control software
- SDx-generated data movers and associated control software
- The Target Communication Framework (TCF)
- GNU software development tools

The SDSoC environment includes the GNU toolchains and standard libraries (for example, `glibc`), a performance analysis perspective within the Eclipse C/C++ Development Tooling (CDT)-based GUI, and command-line tools.

The SDSoC system compiler employs underlying tools from the Vivado Design SuiteHLS Editions including Vivado® HLS, IP integrator, and IP libraries for data movement and interconnect, and the RTL synthesis, implementation, and bitstream generation tools.

The principle of design reuse underlies workflows you employ with the SDSoC environment, using established, platform-based, design methodologies. The SDSoC system compiler generates an application-specific system-on-chip by customizing a target platform.

The SDSoC environment includes a number of built-in platforms for application development, and others can be provided by Xilinx partners, or custom-developed by FPGA design teams. The *SDSoC Environment Platform Development Guide* (UG1146) describes how to create a design using the Vivado Design Suite, specify platform properties to define and configure platform interfaces, and define the corresponding software run-time environment to build a platform for use in the SDSoC environment.

An SDSoC platform defines a base hardware and software architecture and application context, which includes the following:

- Processing system
- External memory interfaces
- Custom input/output
- Software run time including: Operating system (for example, Linux, FreeRTOS, or Standalone), boot loaders, drivers for platform peripherals, and the root file system

Every project you create within the SDSoC environment targets a specific platform. Using the SDx IDE to build on the base platform foundation with application-specific hardware accelerators and data motion networks and connecting accelerators to the platform, you can create customized, application-specific system-on-chip designs for different base platforms, and use base platforms for many different applications.

You are provided the option to use either a platform, that includes a device support archive (DSA), or a hardware definition file (HDF); either of which you can export from Vivado. You can select a pre-defined hardware specification or browse to previously exported DSA/HDF.

See the *SDSoC Environments Release Notes, Installation, and Licensing Guide* (UG1294) for the most up-to-date list of supported devices and required software.

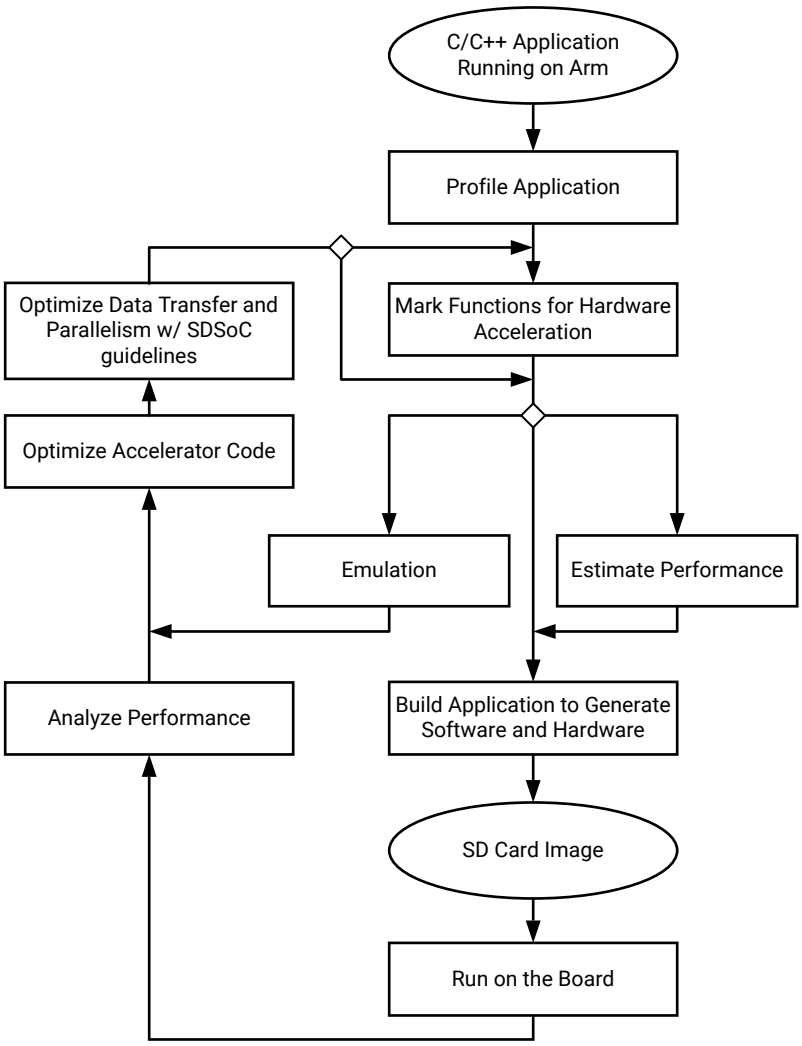
Design Flow Overview

The SDSoC™ environment is a tool suite for building efficient system-on-chip (SoC) applications, starting from a platform that provides the base hardware and target software architecture. A boot image and the executable application code are generated by the SDSoC tools.

The following figure shows a representative top-level design flow that shows key components of the tool suite. For the purposes of exposition, the design flow proceeds linearly from one step to the next, but in practice you are free to choose other work flows with different entry and exit points.

Starting with a software-only version of the application that has been compiled for CPUs, the primary goal is to identify portions of the program to move into programmable logic and to implement the application in hardware and software built upon a base platform.

Figure 5: **User Design Flow**



X14740-071018

The steps are:

1. Select a development platform, compile the application, and ensure it runs properly on the platform.
2. Identify compute-intensive hot spots to migrate into programmable logic to improve system performance, and isolate them into functions that can be compiled into hardware. See Selecting Functions for Hardware Acceleration.
3. Invoke the SDSoC system compiler to generate a complete system-on-chip and SD card image for your application. See Working with Code.

You can instrument your code to analyze performance, and if necessary, optimize your system and hardware functions using a set of directives and tools within the environment. *SDSoC Environment Profiling and Optimization Guide* (UG1235) for profiling and optimization best practices.

The `sds++` compilers orchestrate the system generation process either through the IDE or in the terminal shell using command lines and makefiles. You select functions to run in hardware, specify accelerator and system clocks, and set properties on data transfers. You can insert pragmas into application source code to control the system mapping and generation flows, providing directives to the system compiler for implementing the accelerators and data motion networks.

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

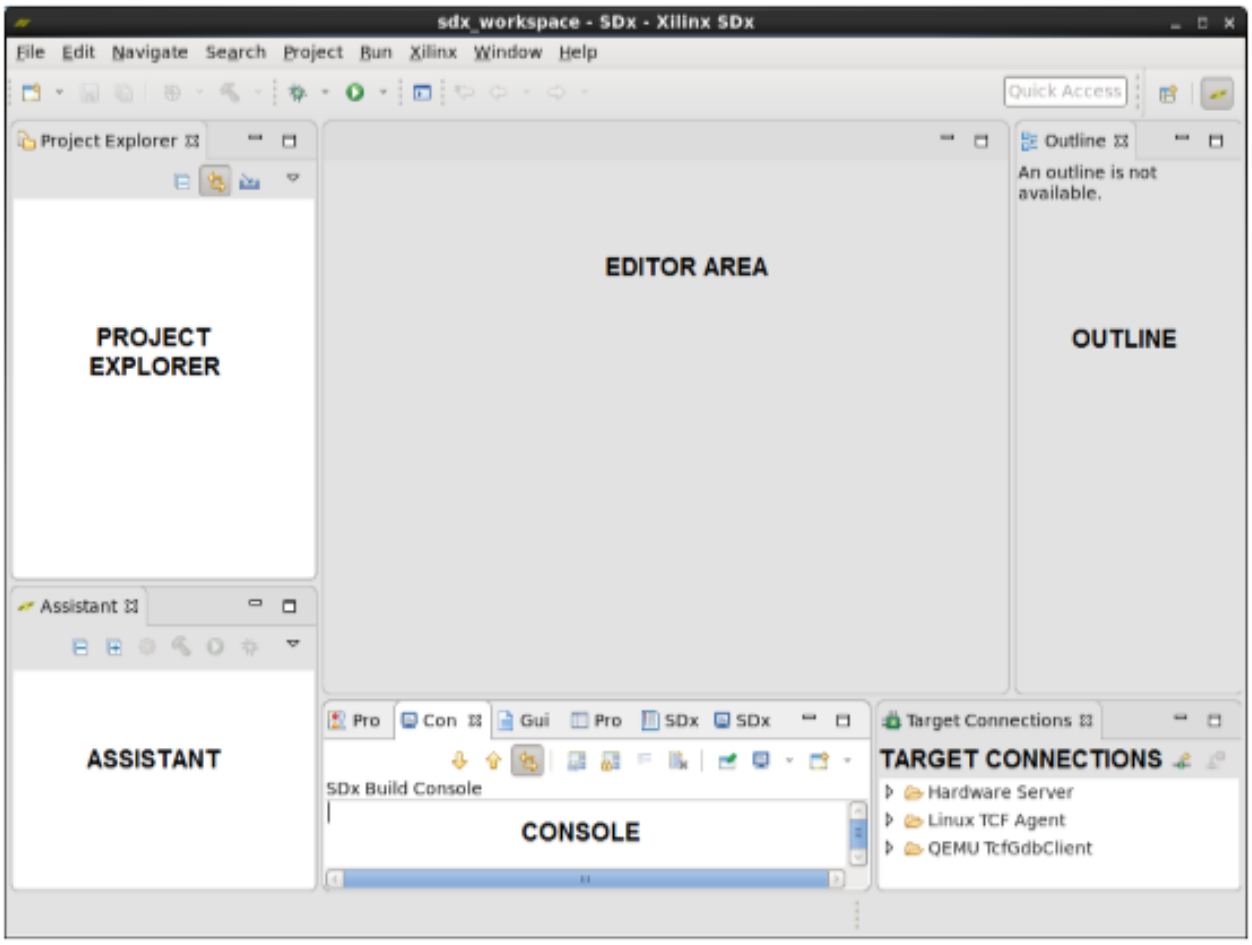
The overall design process involves iterating the steps until the generated system achieves your performance and cost objectives.

To run through the introductory tutorial and become familiar with creating a project, selecting hardware functions, and compiling and running a generated application on the target platform, see *SDSoC Environment Getting Started Tutorial* (UG1028).

Working with the SDx GUI

When you open a project in the SDx™ IDE, the workspace is arranged in a series of different views and editors, also known as a *perspective* in the IDE. The tool opens with the SDx (default) perspective shown in the following figure.

Figure 6: **SDSoC- Default Perspective**



Some key views/editors in the default perspective are:

- **Project Explorer:** Displays a file-oriented tree view of the project folders and their associated source files, plus the build files, and reports generated by the tool.
- **Assistant:** Provides a central location to view/edit settings, build and run your SDSoC application, launch profiling and debug sessions, and open reports.
- **Editor Area:** Displays project settings, build configurations, and provides access to many commands for working with the project.
- **Console Area:** Presents multiple views including the command console, design guidance, project properties, logs and terminal views.
- **Outline:** Displays an outline of the current source file opened in the Editor Area.
- **Target Connections:** Provides status for different targets connected to the SDx tool, such as the Vivado® hardware server, Target Communication Framework (TCF), and quick emulator (QEMU) networking.

To close a view, click the **Close** button (x) on the tab of the view. To open a view, select **Window** → **Show View** and select a view. You can arrange views to suit your needs by dragging and dropping them into new locations in the IDE.

To save the arrangement of views as a perspective, select **Window** → **Perspective** → **Save Perspective As**. This lets you define different perspectives for initial project editing, report analysis, and debug for example. Any changes made without saving as a perspective are stored with the workspace. To restore the default arrangement of views, select **Window** → **Perspective** → **Reset Perspective**. SDx (default) perspective.

To open different perspectives, select **Window** → **Perspective** → **Open Perspective**.

To restore the SDx (default) perspective, click the SDx button  on the right side of the main toolbar.

Creating an SDSoC Application

Using an SDx Workspace

IMPORTANT!: *Linux host is strongly recommended for SDSoC™ platform development, and required for creating a platform supporting a target Linux OS.*

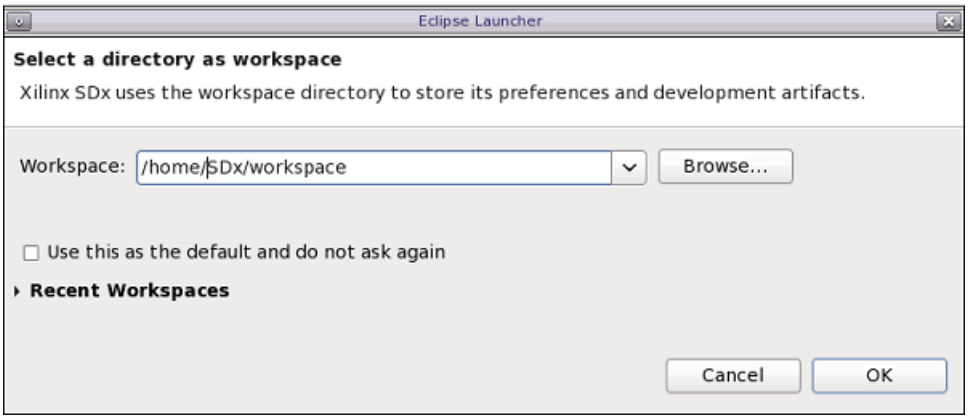
You can launch the SDx IDE directly from the desktop icon or from the command line by one of the following methods:

- Using the following command from the command prompt: `sdx -workspace<workspace_name>`.
- Double-clicking the SDx icon to start the program.
- Launching it from the **Start** menu in the Windows operating system.

IMPORTANT!: *When opening a new shell to enter an SDx™ command, ensure that you first source either the `settings64.sh` file or `settings64.csh` to set up a path to the tools. This sets up the `$PATH` and `$LD_LIBRARY_PATH` environment variables and other required settings. On Windows, run the `settings64.bat` file from the command shell. See the SDSoC Environments Release Notes, Installation, and Licensing Guide (UG1294) for more information.*

The SDx IDE opens, and prompts you to select a workspace, as shown in the following figure.

Figure 7: **Specify the SDx Workspace**



The SDx workspace is the folder that stores your projects, source files, and results while working in the tool. You can define separate workspaces for each project or have workspaces for different types of projects. The following instructions show you how to define a workspace for an SDSoC project.

1. Click the **Browse** button to navigate to, and specify, the workspace, or type the appropriate path in the **Workspace** field.
2. Select the **Use this as the default and do not ask again** check box to set the specified workspace as your default choice and eliminate this dialog box in subsequent uses of SDx.
3. Click **OK**.

TIP: *You can change the current workspace from within the SDx IDE by selecting* ***File*** → ***Switch Workspace****.*

You have now created an SDx workspace.

Creating an Application Project

TIP: *Example designs are provided with the SDSoC™ tool installation, and also on the Xilinx® GitHub repository. See Appendix A: Getting Started with Examples for more information.*

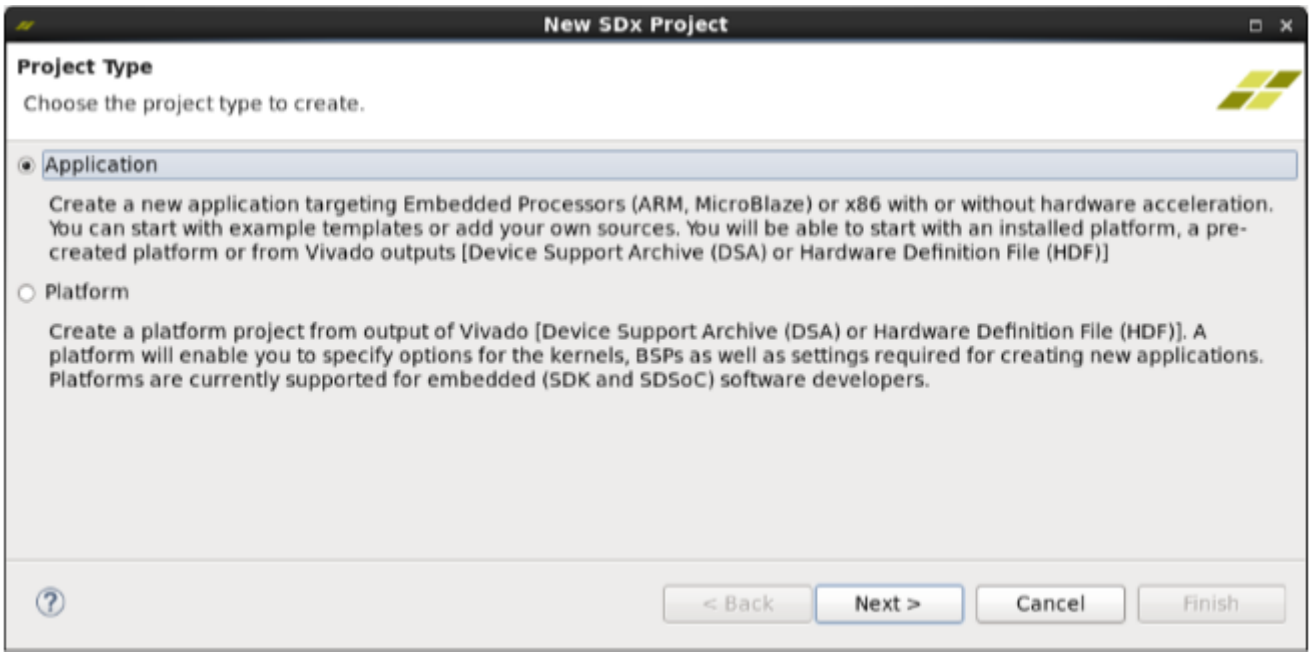
After launching the SDx™ IDE you can create a new SDx Project. Select **File** → **New** → **SDx Project**. If this is the first time the SDx IDE has been launched, the SDx Welcome screen opens to let you specify an action. Select **Create SDx Project**.

The New SDx Project wizard opens. In the Project Type page, you can choose to create either an Application or a Platform project:

- **Application Project**: A software application with portions of the application optionally accelerated onto in the programmable logic (PL) fabric.
- **Platform Project**: Defines the base hardware and software components that make up the SDSoC platform.

Select **Application** and click **Next**.

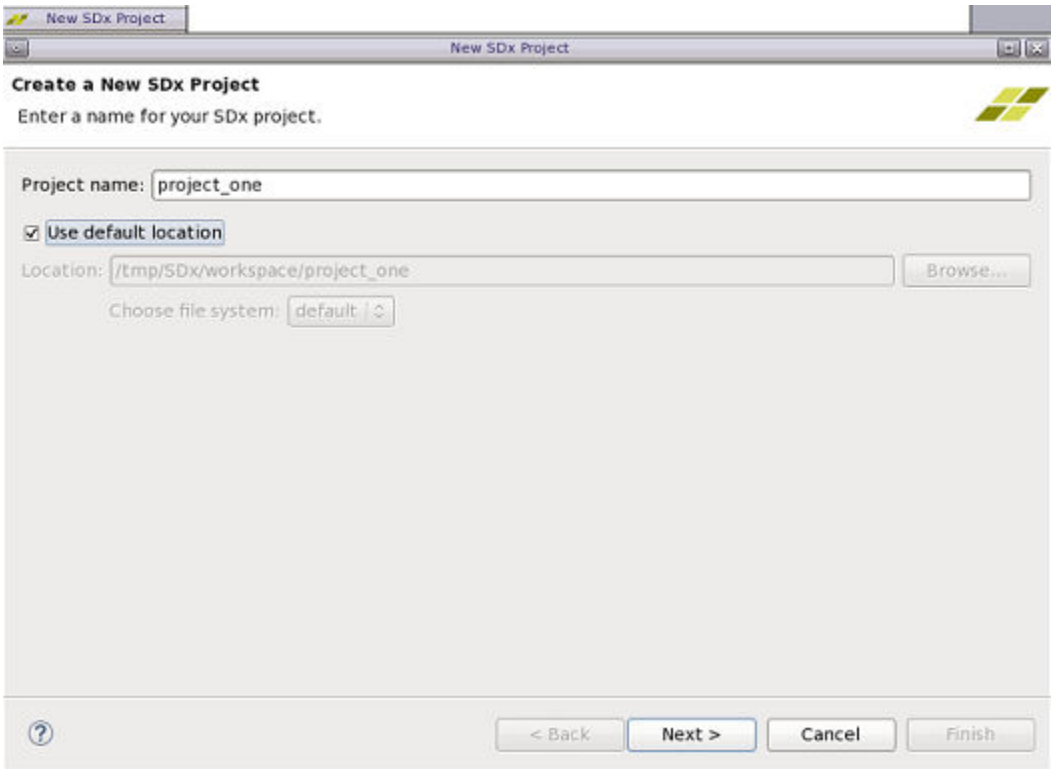
Figure 8: **Project Type**



In the Create a New SDx Project page, you can name your project.

Specify the **Project name**.

Figure 9: **Create New SDx Project**



The **Use default location** is selected by default to locate your project in a folder in the SDx workspace. You can uncheck this check box to specify that the platform project is created in a **Location** of your choice.

If you specify the location, you can use **Choose file system** to select the **default** file system, or enable the Eclipse Remote File System Explorer (**RSE**).

IMPORTANT!: *The project location cannot be a parent folder of an SDx workspace.*

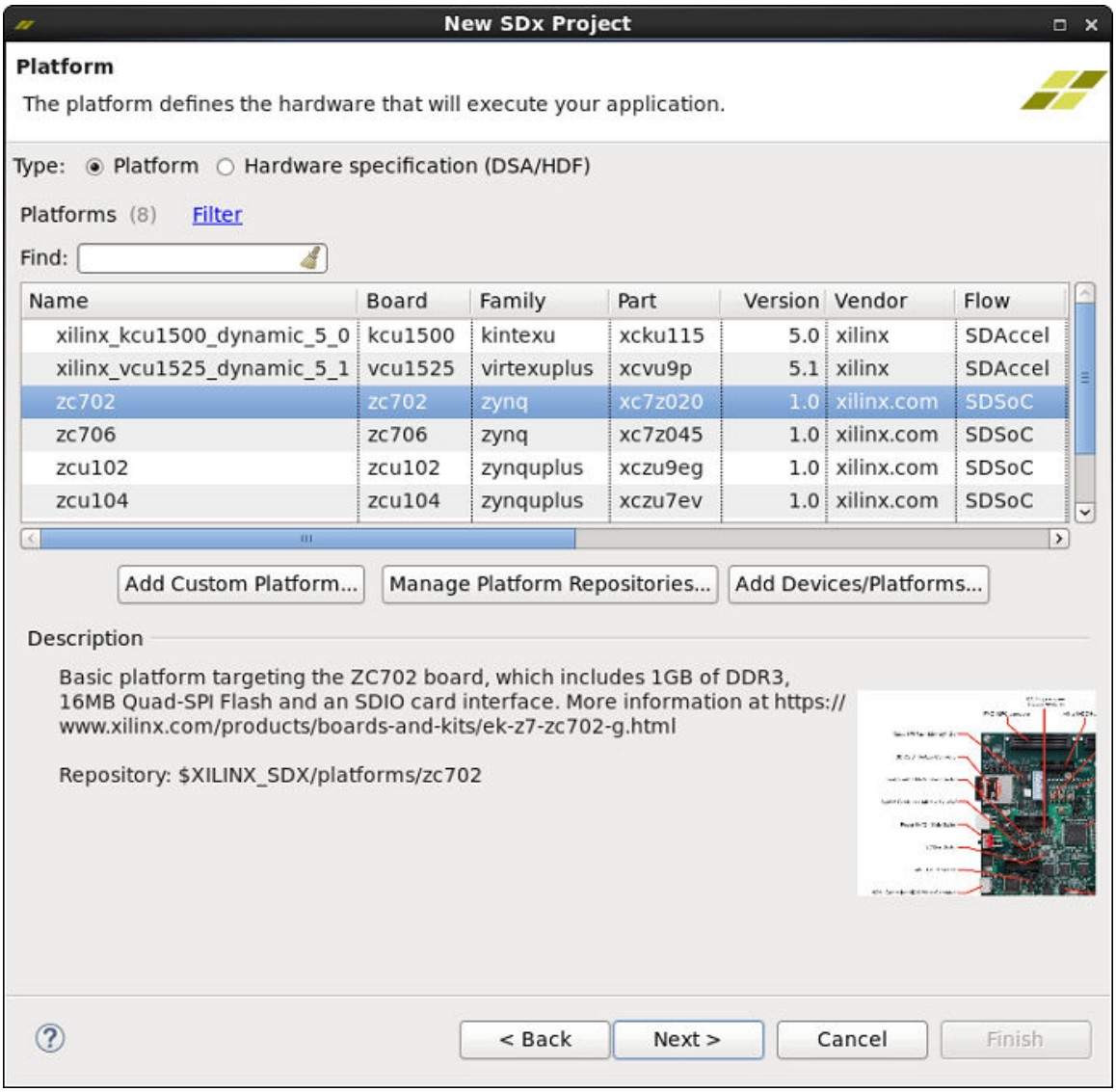
Click **Next**.

The Platform page, similar to the one in the following figure, displays two selection options for an application: a Platform or an Application that uses a device support archive (DSA) or hardware definition file (DSA/HDF.)

You can add platforms into a repository, or even develop a platform to add to projects. See Appendix B: Managing Platforms and Repositories for more information.

Select **zc702** to use the pre-defined ZC702 platform and click **Next**.

Figure 10: **Specify SDSoC Platform**



A platform is composed of a DSA, which describes the base hardware design, the metadata used in attaching accelerators to declared interfaces, and the software environment, which can include operating system images (for example, Linux), as well as boot-up and run-time files.

SDSoC offers pre-defined platforms for specific boards based on the following:

- Zynq®-7000
  - zc702
  - zc706
- Zynq® UltraScale+™ MPSoC

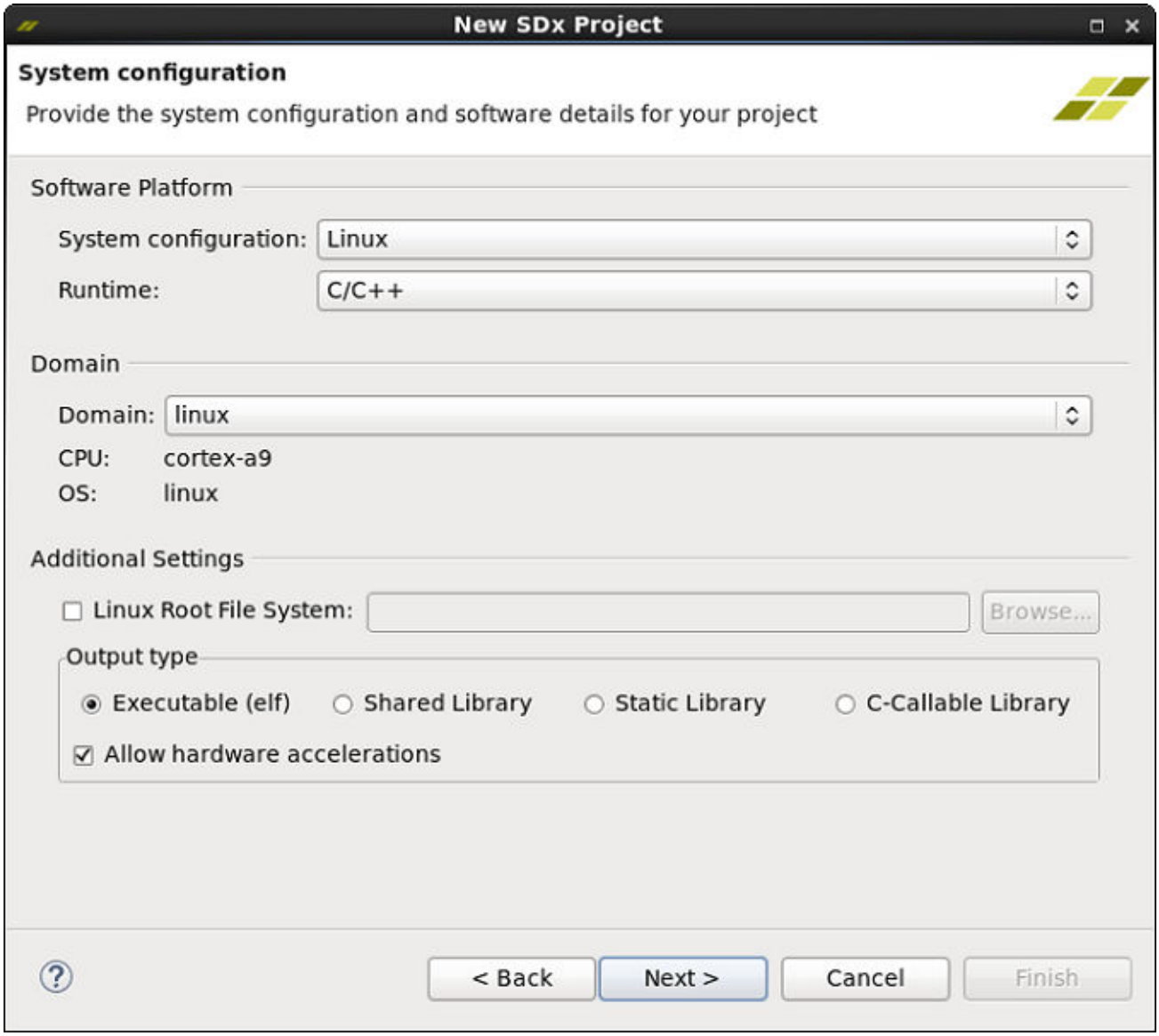
- zcu102
- zcu104
- zcu106

You can select the target platform for your project from the listed platforms. You can also filter the listed platform by clicking the **Filter** link, which opens the Filter dialog box so you can set filters to specify the Flow, Family, or Vendor for your target platform. By default, **Latest Revision** is checked. This option filters out older revisions of the listed platforms.

IMPORTANT!: *Your selection of a platform determines if you are working in an SDAccel project or an SDSoC project. Be sure to select the right platform for your project, as subsequent processes are driven by this choice. The* ***Flow*** *column indicates an SDAccel or SDSoC project.*

After selecting the target platform and clicking **Next**, the System Configuration page opens, as shown in the following figure. It lets you select a **System configuration** and select **Runtime** from a list of those defined for the selected platform. The System Configuration defines the software environment that runs on the hardware platform. It specifies the operating system and the available run-time settings for the processors in the hardware platform.

Figure 11: **Specify System Configuration**



When setting system configuration, you can also check the **Domain**, specifying the processor and **OS**. The Additional Settings contain options for **Linux Root File System**, and **Output type**.

The **Output type** affects the type of SDx project that is created. The default output type is an executable linking file (ELF).

You can specify an executable ELF file, a Shared Library, a Static Library, or a C-Callable IP Library.

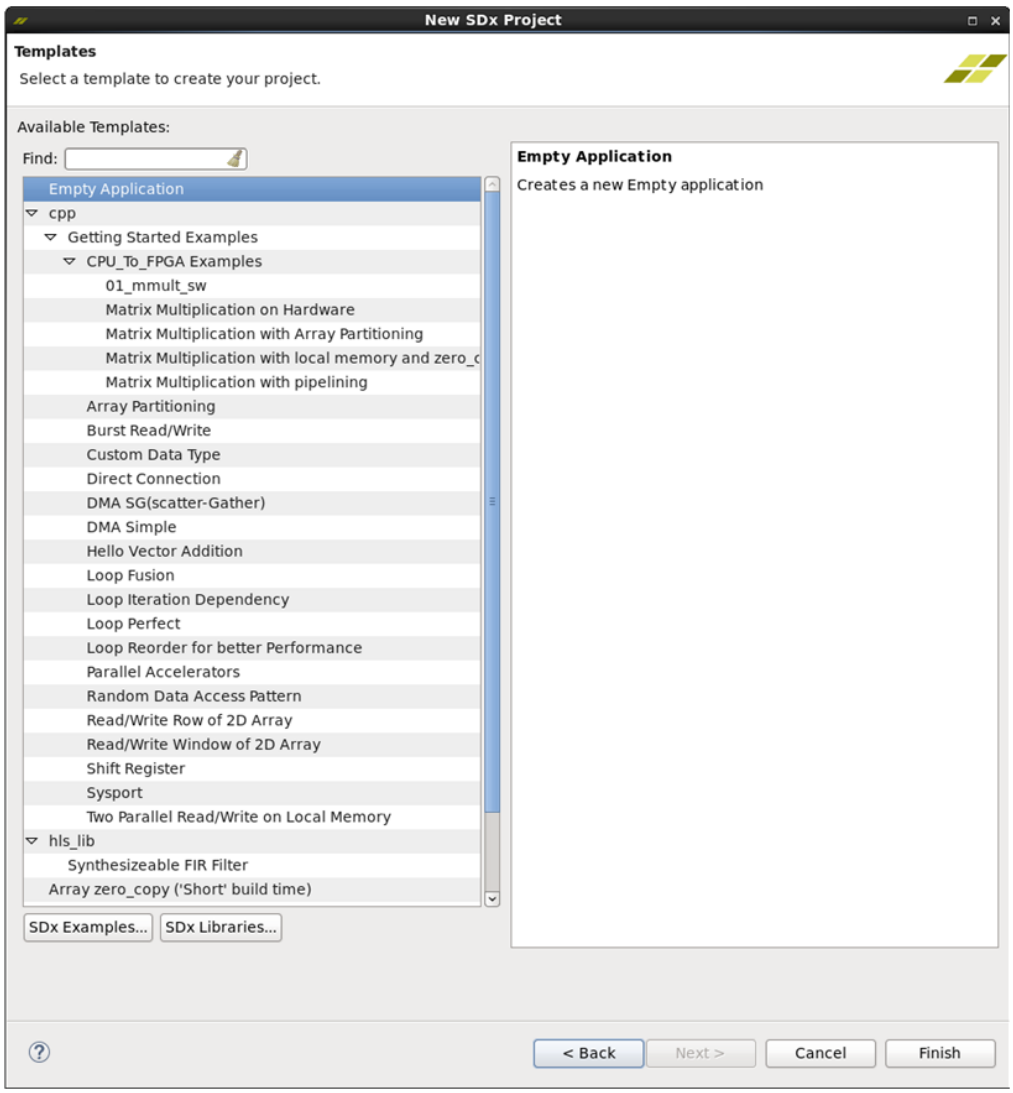
Selecting **C-Callable Library** creates a project that is used to interface to a Vivado® IP block from C code. Unchecking **Allow hardware accelerations** creates an SDK project.

After selecting the **System Configuration** and clicking **Next**, the Templates page displays, as shown in the following figure. It lets you specify an application template for your new project. The `samples` directory within the SDx tools installation contains multiple source code example templates.

Initially, you have the option in the Template dialog box of an **Empty Application** or one of the provided application templates.

The latter provides code to introduce you to application development in SDx. You can download additional SDx examples or update the installed examples by clicking the **SDx Examples** button, which retrieves content from the Xilinx® GitHub as discussed in Appendix A: Getting Started with Examples.

Figure 12: **Application Templates**



You can use the template projects as examples to learn about the SDx tool and acceleration kernels or as a foundation for your new project. Note that you must select a template. You can select **Empty Application** to create a blank project into which you can import files and build your project from scratch.

Click **Finish** to close the New SDx Project wizard and open the project.

Working with Code

The SDx™ environment provides a GUI-based IDE as well as command line control to invoke the `sds++` system compiler with user-specified command options from a shell prompt.

Application code generally consists of C/C++ source files, C/C++ header files, and libraries created for shared or static use. The SDx™ tools let you identify and convert C/C++ source code functions into hardware accelerators. By analyzing function arguments, argument types, and any applied directives or pragmas, the SDx™ tools generate data movers and pipelined data flows to feed data into and out of an accelerator. Typical data sources and data sinks are memories and I/O streams.

The SDSoC-generated accelerators reside in the PL and need to fit in the PL resources. See Execution Model of an SDSoC Application. The interfaces between the PS and PL are user-configurable. For instance, SDx tools automatically generate data movers for crossing the PS-PL boundary with interface guidance as set by your C/C++ source code.

Source files to get started creating an SDx application with example SDx pragmas are provided within the IDE; select **Xilinx** → **SDx Examples**. Additional examples are available for download from the GitHub.

Importing Sources

Importing C/C++ Sources

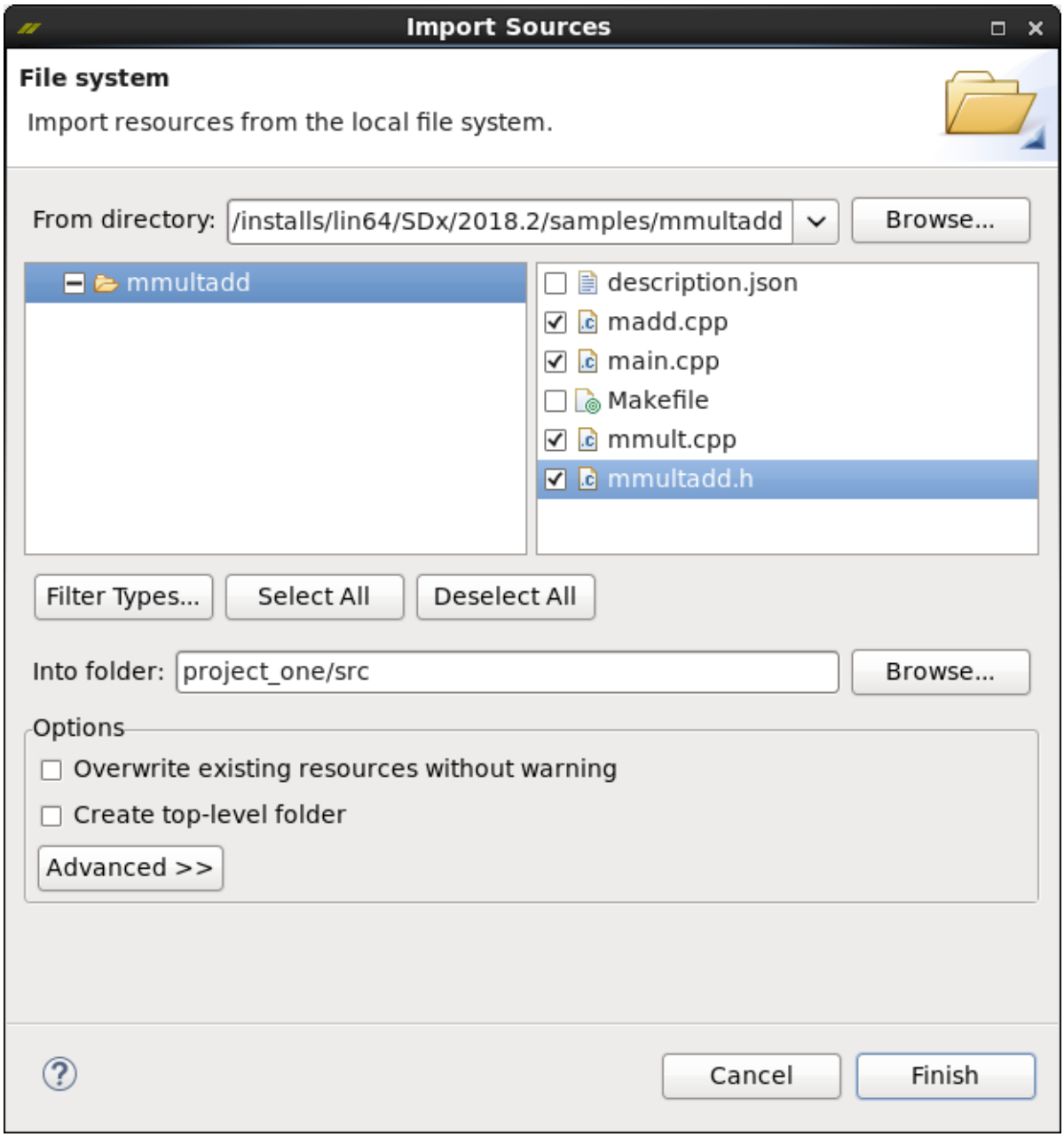
With the project open in the SDx™ IDE, you can import source files to add them to the project. To add files, in the Project Explorer, right-click the `src` folder and select the **Import Sources** command.

IMPORTANT!: *When you import source files into a workspace, it copies the file into the workspace. Any changes to the files are lost if you delete the workspace.*

This displays the Import dialog box that lets you specify the source of the files from which you are importing. The different sources include importing from archives, from existing projects, from the file system, and from a Git repository. Select the source of files to import and click **Next**.

The dialog box that displays depends on the source of files you selected in the prior step. In the following figure, you can see the File system dialog box that displays as a result of choosing to import sources from the file system.

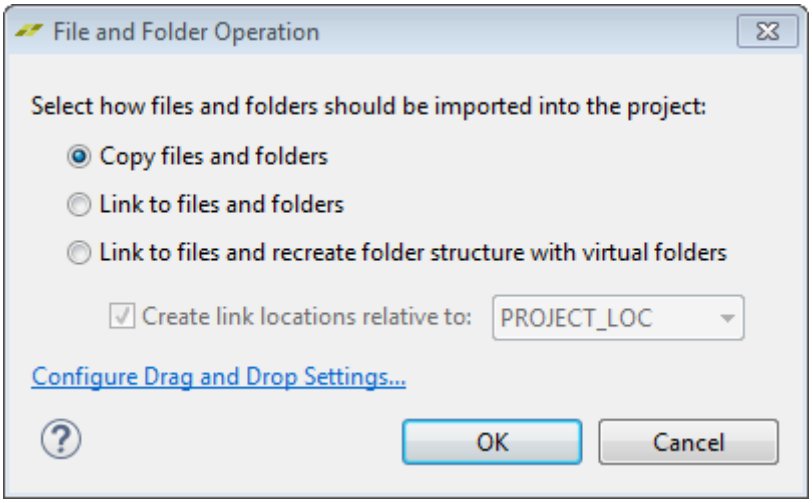
Figure 13: **Import File System Sources**



The File System dialog box lets you navigate to a folder in the system and select files to import into your project. You can specify files from multiple folders, and specify the folder to import files into. You can select the **Overwrite existing resource without warning** check box to overwrite any existing files, and select the **Create top-level folder** check box to have the files imported into a directory structure that matches the source file structure. If this check box is not enabled, which is the default, then the files are imported into the folder listed in the **Into folder** option.

On the Windows operating system, you can add files to your project by dragging and dropping them from the Windows Explorer. Select files or folders in the Explorer and drop them into the `src` folder, or another appropriate folder in the SDx IDE Project Explorer. When you do this, the tool prompts you to specify how to add the files to your project, as shown in the following figure.

Figure 14: **File and Folder Operation**



You can copy files and folders into your project, add links to the files, or link to the files in virtual folders to preserve the original file structure. There is also a link to the **Configure Drag and Drop Settings** option, which lets you specify how the tool should handle these types of drag and drop operations by default. You can also access these settings through the **Windows → Preferences** menu command.

After adding source files to your project, you are ready to begin configuring, compiling, and running the application.

RECOMMENDED: *When you make code changes, including changes to hardware functions, it is valuable to rerun a software-only compile to verify that your changes did not adversely change your program. A software-only compile is much faster than a full-system compile.*

Importing C-Callable IP Libraries

In addition to C/C++ source files, you can incorporate pre-existing hardware functions in your design with use of a C-Callable IP library, that is published as an `arm.a` static library file. Code examples that use C-Callable IP are available in the SDSoC™ installation tree under the `samples/rtl` directory. You create and add your own C-Callable IP libraries through the SDx IDE.

***Note*:** The static library (`.a` files) might need to be rebuilt for the appropriate Arm® processor (Cortex™-A9 or Cortex-A53) before adding it to the project. Add the associated header file (`.h`) for the C-Callable IP function to the project, as follows:

1. In the Project Explorer, right-click the project.
2. Select **SDS++ Linker** → **Libraries** → **C/C++ Build Settings**.
3. Add *`libname_without_leading_lib_text_characters`* to the `-l`Libraries list.
4. Add the directory that contains the `libname.a` source file to the `-L` Library search path.
5. Click **Apply**.
6. Click **OK**.
7. Ensure that a C-Callable IP header (`.h`) file is present in project source tree.
8. Build the project.

See Chapter 4: C-Callable IP Libraries for more information on creating and using C-Callable IP libraries.

Selecting Functions for Hardware Acceleration

The first task is to identify portions of application code that are suitable for implementation in hardware, and that significantly improve the overall application performance when run in hardware.

Before marking any functions for acceleration you should profile the software. Self-contained compute intensive functions with limited external control logic are good starting points, especially when its possible to stream data between hardware, the CPU, and memory to overlap the computation with the communication. You can build and run your application on one of the platforms provided in the SDSoC install to identify compute intensive functions on the Arm® processor.

Every platform included in the SDSoC environment includes a pre-built SD card image from which you can boot and run your application code if you do not have any functions selected for acceleration on the hardware platform. Running the application this way lets you profile the original application code to identify candidates for acceleration.

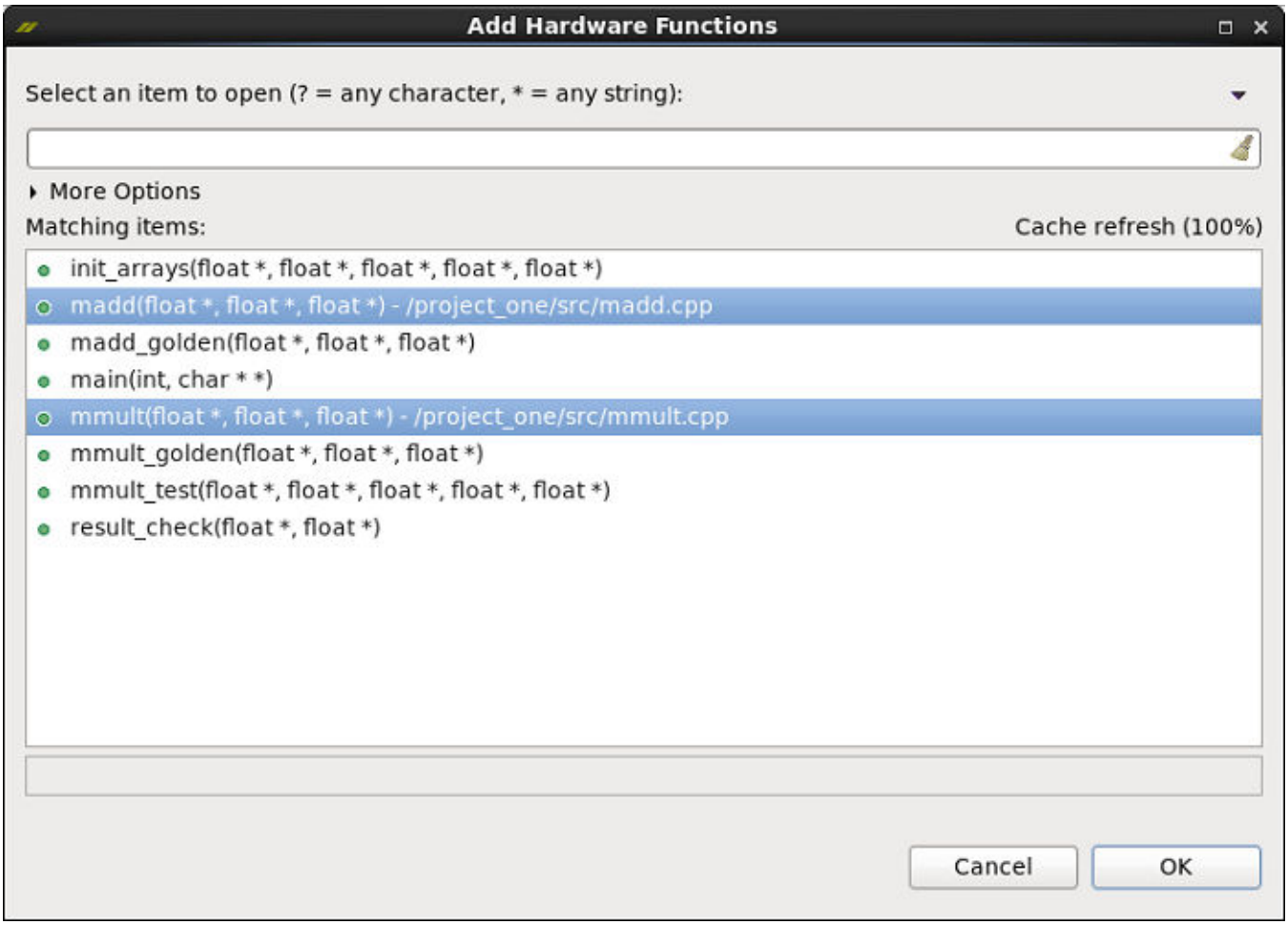
See Chapter 6: Profiling and Optimization for more information on profiling your application. Also, the *SDSoC Environment Profiling and Optimization Guide* (UG1235) provides more extensive detail.

After determining the function or functions to move into hardware, with a project, you can select the function from the Add Hardware Functions portion of the window.

TIP: *If the Editor Area window is not open, you can double-click the* `<project>.sdx` *file in the* ***Project Explorer*** *to open it, or you can use the* ***Add Hardware Function*** *context menu from the Assistant that is available when you right-click the project.*

Click the **Hardware Functions** button of the Project Editor window to display the list of candidate functions within your program. This displays the Add Hardware Functions dialog box, that lets you select one or more functions from a list of functions in the call graph of the `main` function by default.

Figure 15: **Add Hardware Functions Dialog Box**

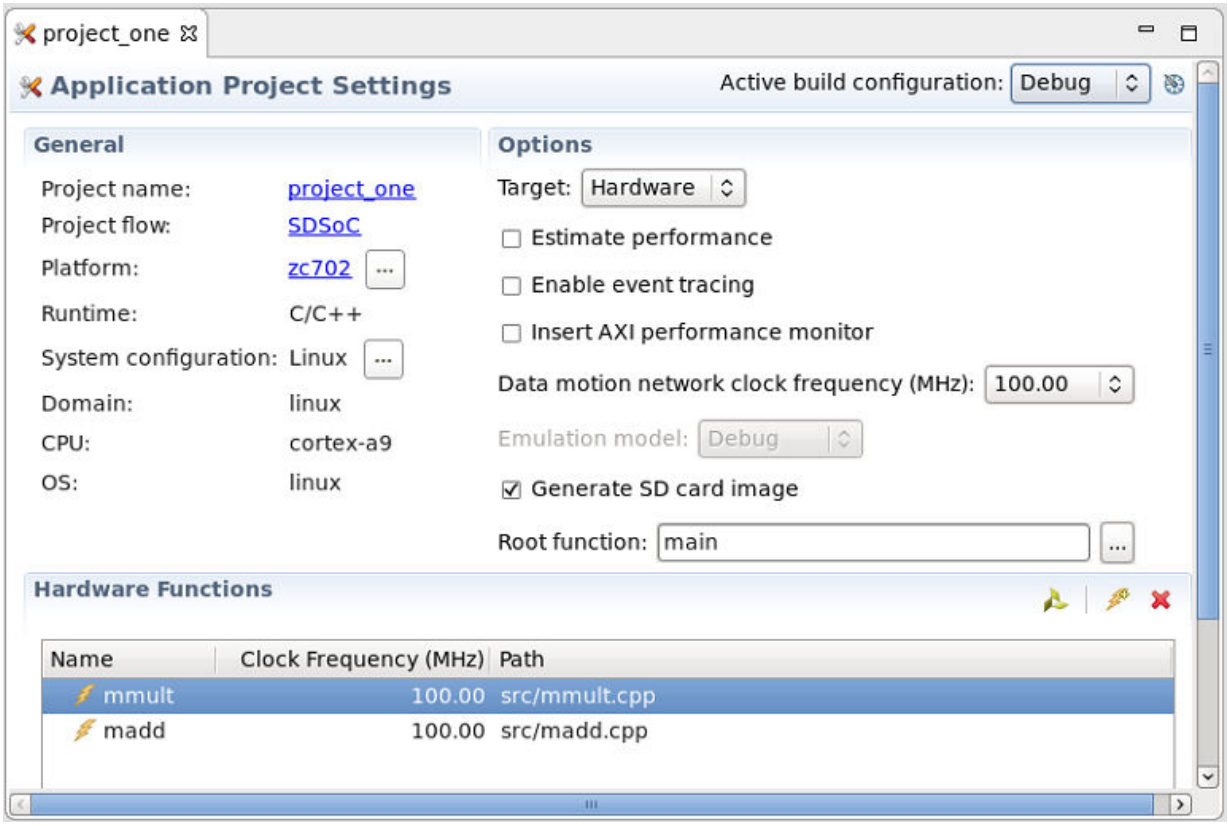


From within the Add Hardware Functions dialog box, select one or more functions for hardware acceleration and click **OK**.

The list of functions starts at the **Root function** as defined in the **Options** panel of the Project window, and is set to `main` by default. You can change the **Root function** by clicking the **Browse** (...) command and selecting an alternate root.

The functions display in the Hardware Functions panel of the Application Project Setttings window as shown in the following figure.

Figure 16: **Hardware Function Panel**



TIP: *If you do not see a function you expect in the Add Hardware Function dialog box, navigate to its source file in the* ***Project Explorer*** *window, expand the outline of the source, right-click the function and select* ***Toggle HW/SW****.*

When moving a function optimized for CPU execution into programmable logic, you can usually revise the code to improve the overall performance. See the "Programming Hardware Functions" section in the *SDSoC Environment Programmers Guide* (UG1278).

For accelerators using the `xfOpenCV` library, right-click and select **Toggle Hardware** from the associated header files in the project included in Project Explorer. See the *Xilinx OpenCV User Guide* (UG1233) for more information.

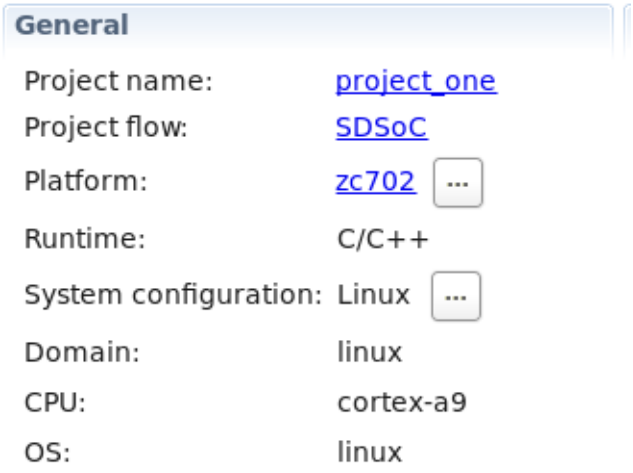
Selecting Clock Frequencies

After selecting hardware functions, it could be necessary to select the clock frequency for running the function or the data motion network clock.

Every platform supports one or more clock sources, which is defined by the platform developer as described in *SDSoC Environment Platform Development Guide* (UG1146). Setting the clock frequency of the hardware function determines the clock setting for the hardware function's logic as well as its associated data mover. Setting the clock frequency of the Data Motion network determines the clock used for the axi_lite control buses. By default, the Data Motion network shares its clock with the hardware accelerator generated during system generation. You can select the Data Motion network clock and the hardware function clock from the SDx™ IDE or the command line.

You can view the available platform clocks by selecting the Platform link in the **General** option of the **Application Project Settings** window. This displays details of the platform, including the available clock frequencies.

Figure 17: **SDx™ IDE - General**

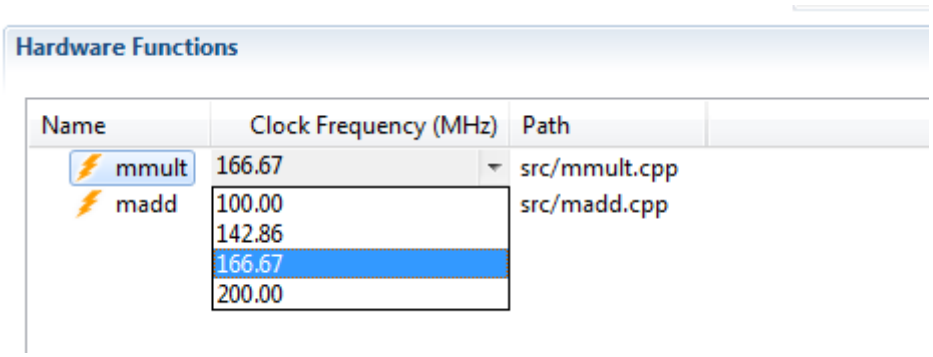


IMPORTANT!: *Be aware that it might not be possible to implement the hardware system with some faster clock selections. If this occurs, reduce the clock frequency.*

The function clock displays in the SDx™ Project window, in the **Hardware Functions** panel.

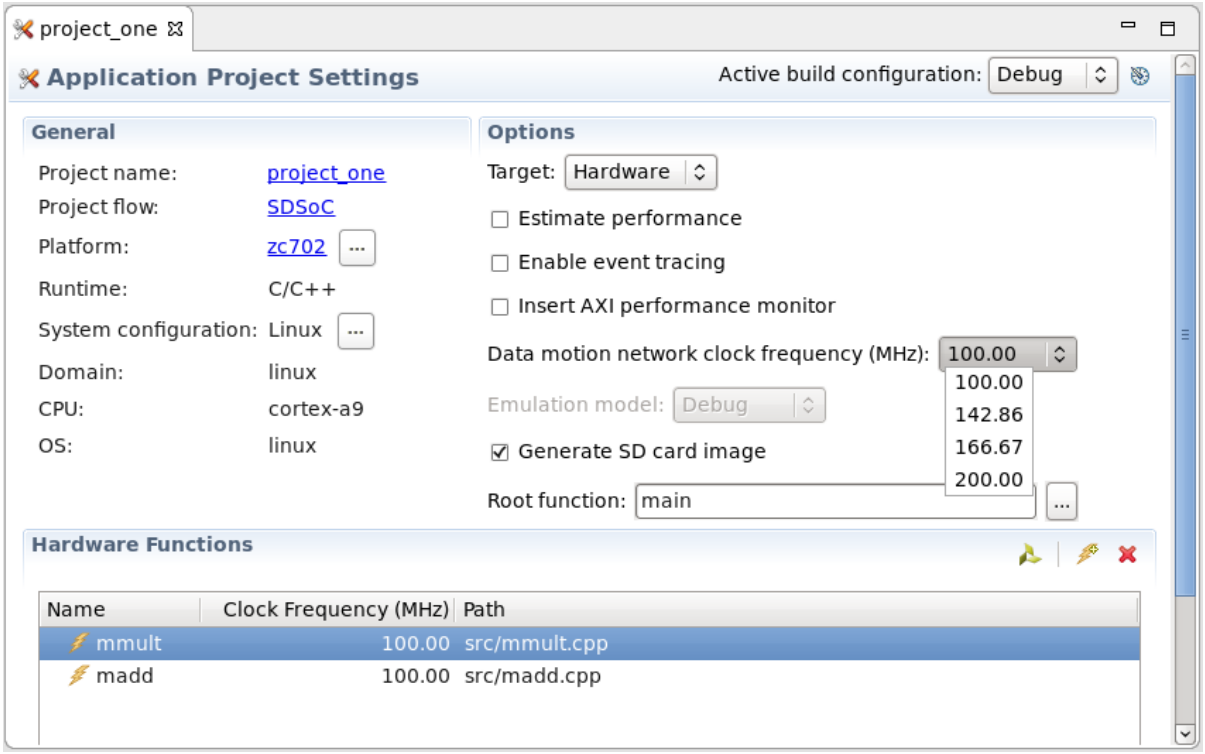
Select a function from the list, like `mmult` in the figure below, and click in the **Clock Frequency** column to access the pull-down menu to specify the clock frequency for the function.

Figure 18: **Select Function Clock Frequency**



To specify the Data Motion clock frequency, select the **Data motion network clock frequency** pull-down menu in the **Options** panel of the SDx Project Editor window. The Data Motion network clock frequency menu is populated by the available clocks on the platform.

Figure 19: **Data Motion Network Clock Frequency**



Command Line Options

You can set clock frequencies for either the Hardware Accelerator or the Data Motion network at the command line, as shown in the following examples.

Set the clock frequency for a function from the command line, by specifying the Clock ID (`clkid`).

```
$ sds++ -c mmult.cpp -o mmult.o -sds-pf zc702 -sds-hw mmult mmult.cpp -
clkid 1 -sds-end
```

Select a Data Motion clock frequency from the command line with the `-dmclkid` option. For example:

```
$ sds++ -sds-pf zc702 -dmclkid 1
```

You can use the following command to see the available clocks for a platform, and determine the clock ID:

```
sds++ -sds-pf-info <platfom_name>
```

Building the SDSoC Project

After you have added source code, identified the functions to accelerate, and selected the accelerator and data mover clock frequencies, you can build for a Hardware target or an Emulation target.

For Hardware targets, see Targeting Hardware

For Emulation targets, see Running System Emulation.

The following section describes the SDx™ project to target a Hardware Project build:

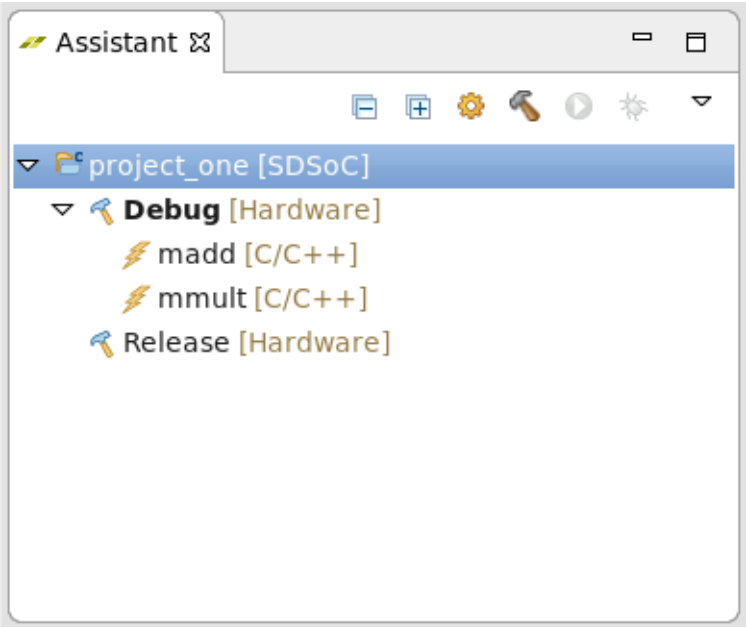
Targeting Hardware

Select the **Build** button () or right-click the project name and select **Build Project** in the Project Explorer window.

Initiate a build through the Assistant window by right-clicking either the **Debug[Hardware]** or **Release[Hardware]** selections.

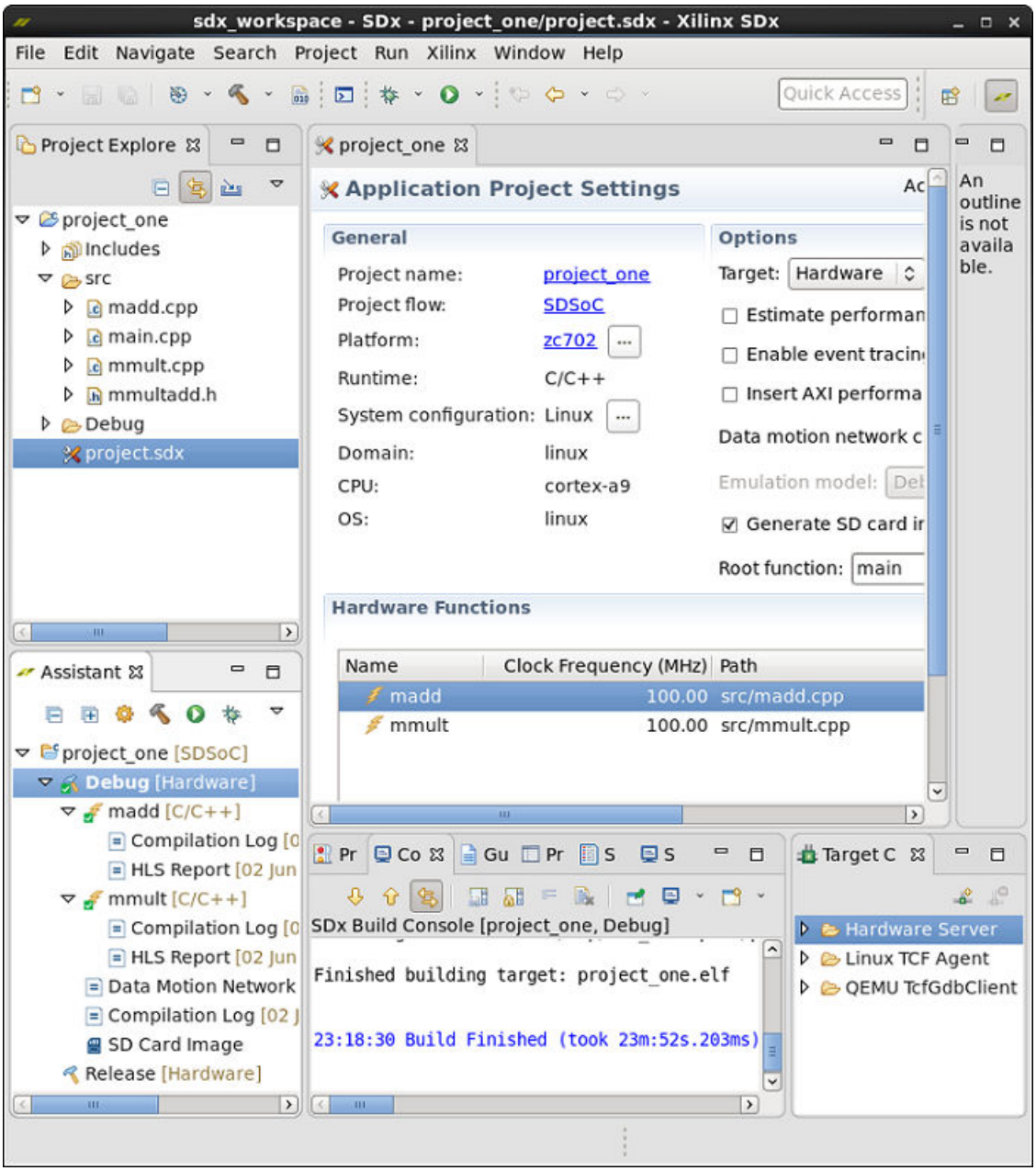
Right-click **Debug [Hardware]** and, from the context menu, select **Build**.

Figure 20: **Assistant Window**



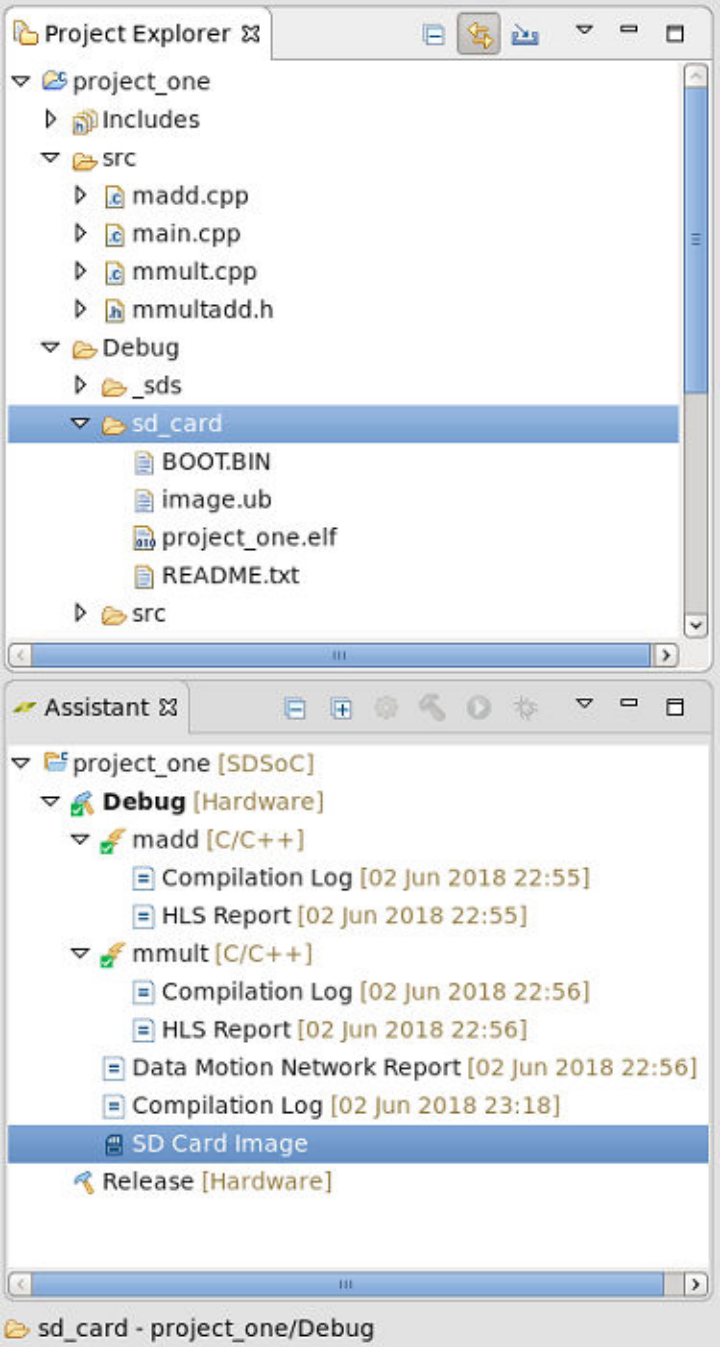
This generates a bitstream and bootable SD card image, based on the check-marked project options. The build process also produces a **Compilation Log** file and a **Data Motion Network Report** that you can access through the Assistant window.

Figure 21: **Build Invocation**



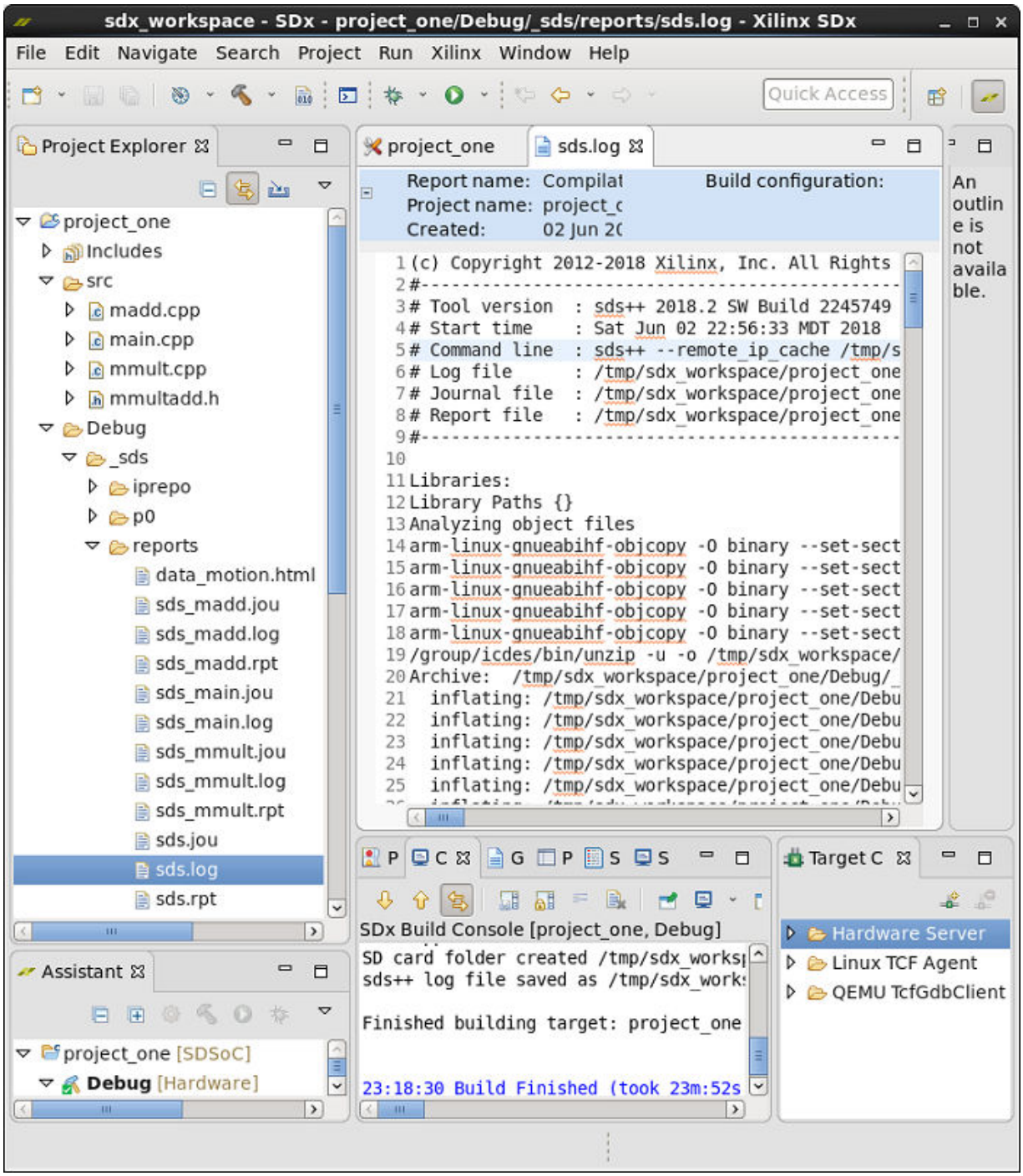
The results of the build process produce files to populate an SD card for booting and running on a target board. Using the Assistant window the set of generated files can be viewed by right-clicking and selecting **Open** → **Open in Project Explorer**.

Figure 22: **Build Results**



SDx creates detailed reports of the compilation process and saves those files in the `Debug/_sds/reports` directory. Access these files through the Project Explorer or the Assistant window. The following figure shows a Project Explorer expansion of the `reports` directory and a view into the `sds.log` compilation log file.

Figure 23: **Compilation Log**



An excerpt of a log file showing the equivalent system compiler commands run by the IDE for creating the accelerators added as part of the matrix multiply and matrix add example, shown in the following code snippet. The `-sds-pf` option identifies the platform, and each accelerated function is identified by placing the function name and its defining source file between their own set of `-sds-hw` and `-sds-end` hardware function options. The accelerator clock ID is also chosen within the hardware function options by setting the `-clkid` option.

Compilation of function `madd()`:

```
sds++ -Wall -O0 -g -I"../src" -c -fmessage-length=0 -MT"src/madd.o"\
 -MMD -MP -MF"src/madd.d" -MT"src/madd.o" -o "src/madd.o" "../src/
madd.cpp"\
 -sds-hw mmult mmult.cpp  -clkid 2 -sds-end\
 -sds-hw madd madd.cpp  -clkid 2 -sds-end\
 -sds-sys-config linux -sds-proc linux -sds-pf "zc702"
```

Compilation of function `main()`:

```
sds++ -Wall -O0 -g -I "../src" -c -fmessage-length=0 -MT"src/main.o"\
 -MMD -MP -MF"src/main.d" -MT"src/main.o" -o "src/main.o" "../src/
main.cpp"\
 -sds-hw mmult mmult.cpp  -clkid 2 -sds-end\
 -sds-hw madd madd.cpp  -clkid 2 -sds-end\
 -sds-sys-config linux -sds-proc linux -sds-pf "zc702"
```

Compilation of function `mmult()`:

```
sds++ -Wall -O0 -g -I"../src" -c -fmessage-length=0 -MT"src/mmult.o"\
 -MMD -MP -MF"src/mmult.d" -MT"src/mmult.o" -o "src/mmult.o" "../src/
mmult.cpp"\
 -sds-hw mmult mmult.cpp  -clkid 2 -sds-end\
 -sds-hw madd madd.cpp  -clkid 2 -sds-end\
 -sds-sys-config linux -sds-proc linux -sds-pf "zc702"
```

The compiled object files are then linked by the SDx tools to produce the single executable file for the matrix multiply and matrix add example containing the application code as well as the code to invoke the accelerated functions. This executable is targeted for Linux as specified with the `-sds-sys-config` and `-sds-proc` options.

Linking object files to produce an executable file (ELF) and boot files for SD card:

```
sds++ --remote_ip_cache /sdx_wksp_ex_mmult_madd/ip_cache -o
"ex_mmult_madd.elf"\
 ./src/madd.o ./src/main.o ./src/mmult.o\
 -dmclkid 2 -sds-sys-config linux\
 -sds-proc linux -sds-pf "zc702"
```

XP Option: Advanced Feature for Controlling a Vivado Build

While the `sds++` system compiler automatically invokes the Vivado® design tools to implement the hardware system, users who are familiar with Vivado tool options have the ability to further customize the flow by passing arguments on the command line. The `-xp` option of the `sds++` system compiler can be used to pass a `parameter-value` or `property-value` pairs into the Vivado tools for guiding accelerator implementation.

The following is an example of specifying a Vivado synthesis option:

```
sds++ <command_options> -xp
"vivado_prop:run.synth_1.STEPS.SYNTH_DESIGN.TCL.POST=<full path to
postsynth.tcl>"
```

This example `-xp` option specifies a post-synthesis execution of a tool Tcl file. Multiple `-xp` options can be specified on a single system compiler invocation.

However, for synthesis and implementation strategies, the `sds++` command directly takes the `-impl-strategy` and `-synth-strategy` options, and does not use the `-xp` option.

For example:

```
sds++ <command_options> -synth-strategy <strategy_name>
```

```
sds++ <command_options> -impl-strategy <strategy_name>
```

```
sds++ <command_options> -synth-strategy <strategy_name> -impl-strategy
<strategy_name>
```

When using the IDE, the `-xp` options are accessed by right-clicking the project within the Project Explorer window and selecting **C/C++ Build Settings** → **SDS++ Linker** → **Miscellaneous** tool settings are then used to set the linker flags.

See the following resources for more information:

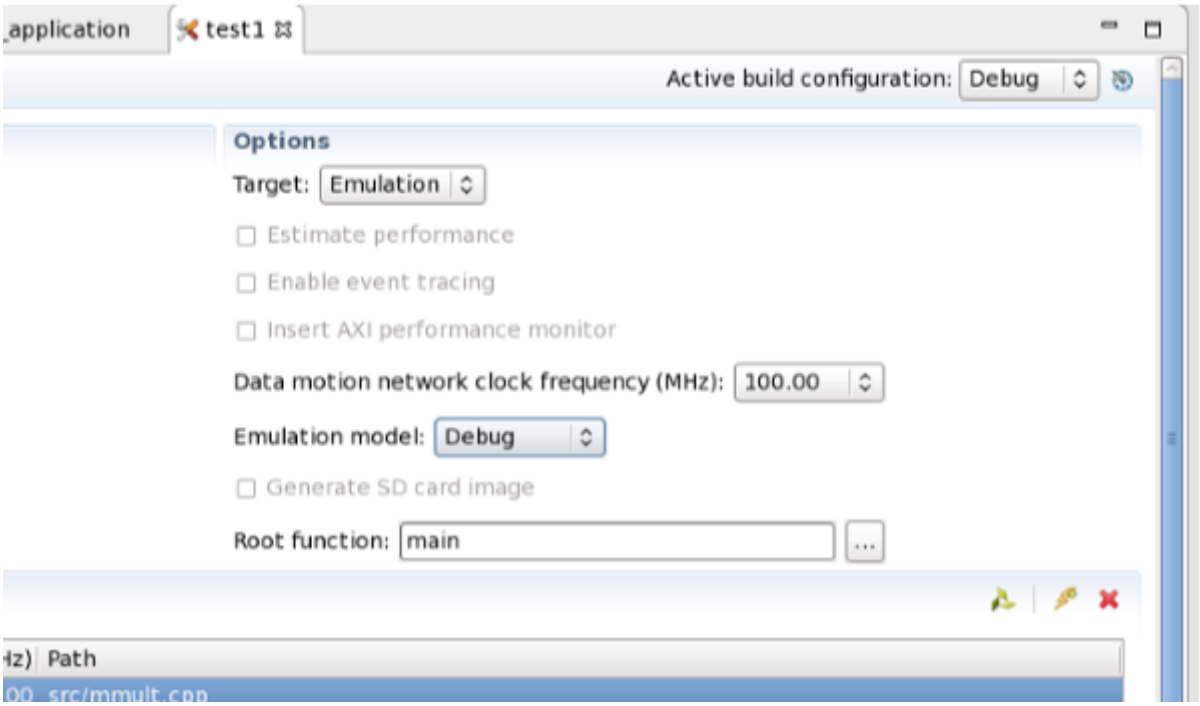
- *Vivado Design Suite Tcl Command Reference Guide* (UG835))
- *Vivado Design Suite User Guide: Synthesis* (UG901
- *Vivado Design Suite User Guide: Implementation* (UG904)
- *SDSoC Environment Profiling and Optimization Guide* (UG1235)
- *SDx Command and Utility Reference Guide* (UG1279)

Targeting System Emulation

After the hardware functions are identified, the logic can be compiled, and the entire system (PS and PL) verified using SDSoC™ emulation on Xilinx® base platforms (such as zc702, zc706, zcu102, zcu104, zcu106). System emulation lets you verify and debug the system with the same level of accuracy as a full bitstream compilation, without requiring a bitstream. This can significantly reduce design iteration time, and allow faster iteratation through debug cycles more.

To enable system emulation, from the Editor Area, click the **Target** field, and select **Emulation**.

Figure 24: **Emulation Target**



SDSoC emulation offers **Debug** and **Optimized** modes, which can be specified by clicking in the **Emulation Model** field.

- **Debug**: Builds the system through RTL generation, and the IP integrator block design containing the hardware function, elaborates the hardware design, and runs behavioral simulation on the design, with a waveform viewer to help you analyze the results. If there is a functional issue with the code, use this option.
- **Optimized**: Runs the behavioral simulation in batch mode, returning the results without the waveform data. While Optimized can be faster, it returns less information than Debug mode.

To capture waveform data from the PL hardware emulation for viewing and debugging, select the Debug pull-down menu option. For faster emulation without capturing this hardware debug information, select Optimized.

After specifying the emulation options, click the **Build** ( ) command to compile the active build configuration. The Build command invokes the system compilers to build your application project. There are two build configurations available:

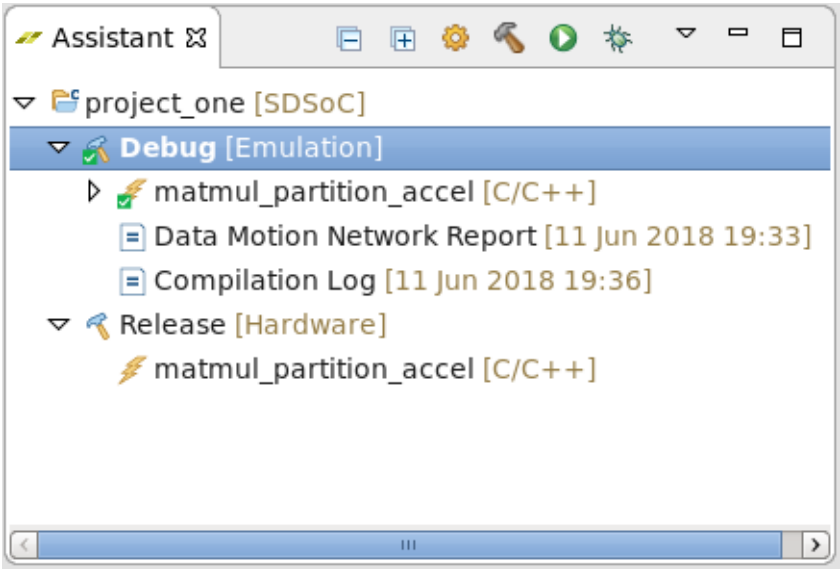
- **Debug**: This compiles the project use in software debug. The compiler produces extra information for use by a debugger to facilitate debug and let you step through the code.
- **Release**: The compiler tries to reduce code size and execution time of the application. This strips out the debug code so that you really cannot do debug with this version.

TIP: *Debug and Release modes describe how the software code is compiled; it does not affect the compilation and implementation of the hardware functions.*

You can launch the build from the Assistant window also, by selecting **Debug[Emulation]** → **Build**.

After SDSoC emulation has completed, the Assistant lists the **Data Motion Network Report**, and the **Compilation Log**, as shown in the following figure:

Figure 25: **Assistant Window Display**

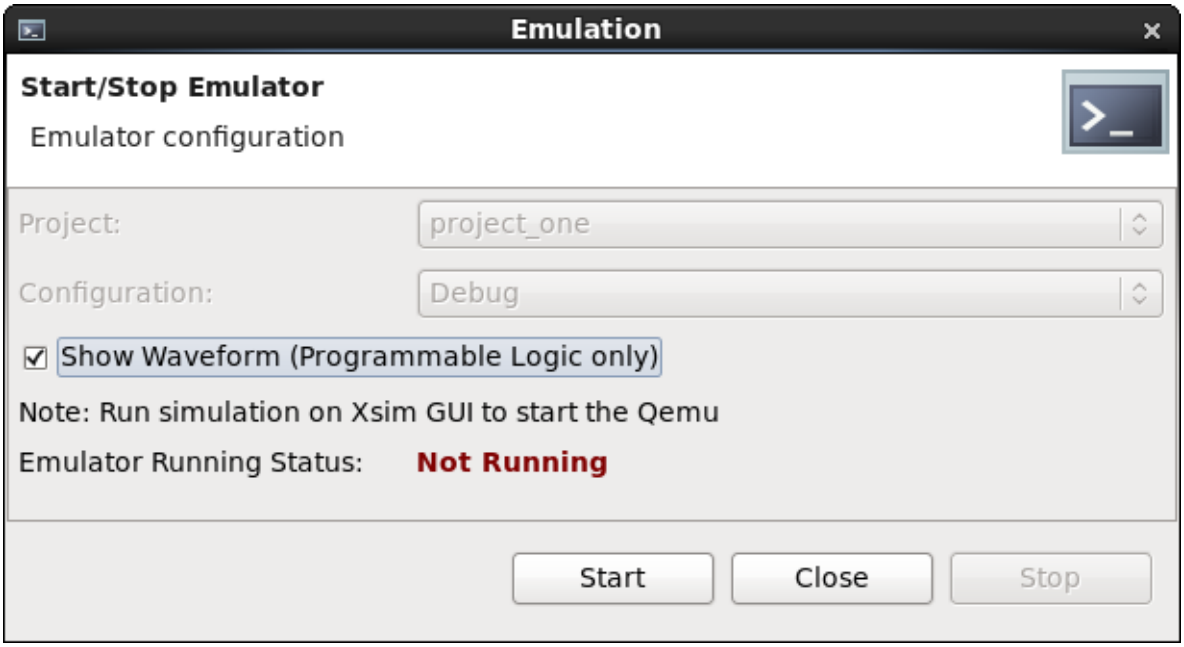


The build process can take some time, depending on your application code, the size of your hardware functions, and the various options you have selected. To compile the hardware functions, the tool stack includes SDx, Vivado HLS, and the Vivado simulator, that is invoked when a waveform is required..

After the system is compiled for emulation, you can invoke the system emulator using the **Xilinx** → **Start/Stop Emulator** menu command, or using `sdsoc_emulator` from the command line.

When the Start/Stop Emulator dialog box opens, if the specified emulation mode is **Debug**, you can choose to run emulation with, or without waveforms. If the emulation mode is **Optimized**, the **Show waveforms** check-box is disabled, and cannot be changed.

Figure 26: **Start/Stop Emulator**



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

Enabling the **Show Waveform** option provides the same functionality in the console window, plus the behavioral simulation of the PL-resident IP, with a waveform window. The waveform window allows you to see the value of any signal in the hardware functions over time. When using **Show Waveform**, you must manually add signals to the waveform window before starting the emulation. Use the Scopes pane to navigate the design hierarchy, then select the signals to monitor in the Object pane, and right-click to add the signals to the waveform pane. Select the **Run → Run All** option to start updates to the waveform window. For more information on working with the Vivado® simulator waveform window, see the *Vivado Design Suite User Guide: Logic Simulation* (UG900).

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

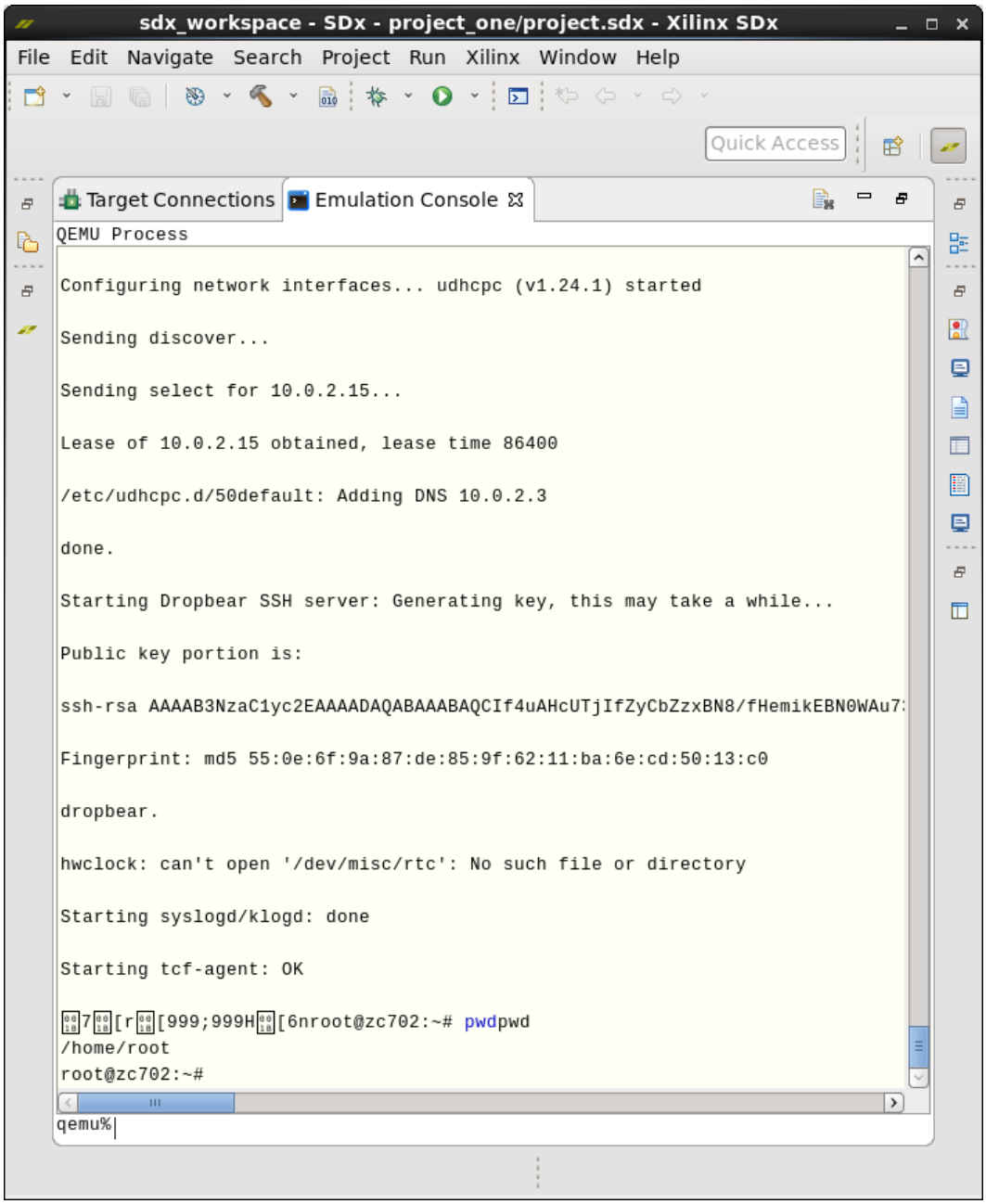
The system emulation can also be started by selecting the active project in the Project Explorer view and right-clicking to select **Run As** → **Launch on Emulator** menu command, or the **Debug As** → **Launch on Emulator** menu command. Launching the emulator from the **Debug As** menu prompts you to change to the debug perspective to arrange the windows and views to facilitate debugging the project. See Working with the SDx GUI for more information on changing perspectives.

You see the program output in the console tab, and if the **Show Waveform** option was selected, you also see any appropriate response in the hardware functions in the RTL-PL waveform. During any pause in the execution of the code, the RTL-PL waveform window continues to execute and update, just like an FPGA running on the board.

The emulation can be stopped at any time using the menu option **Xilinx** → **Start/Stop Emulator** and selecting **Stop**.

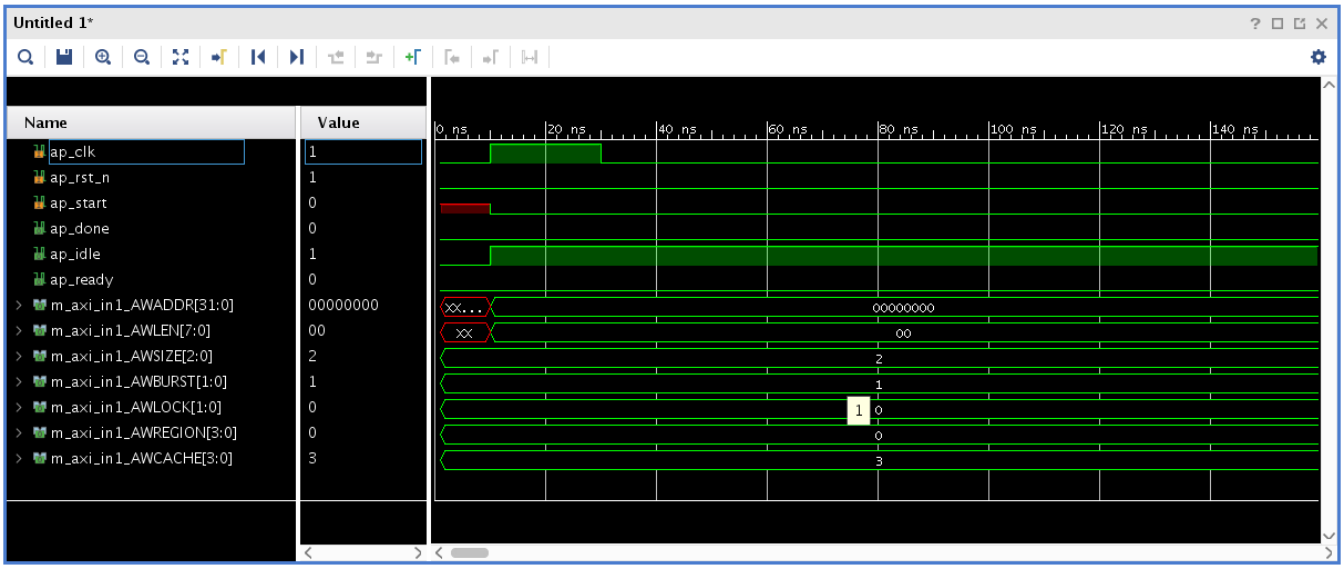
A system emulation session with the quick emulation (QEMU) console, is shown in the following figure.

Figure 27: **QEMU Console**



As shown in the following figure, the PL waveform displays if you selected the waveform option and the **Run All** option:

Figure 28: **Waveform Window**



The Assistant window lets you right-click **Debug → Run → Launch on Emulator (SDx Application Debugger)**.

You can find more information about emulation in the *SDSoC Environment Debugging Guide* (UG1282).

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

Guidelines for Invoking SDSCC/SDS++

The `sds++` system compiler provides a command line interface that is easily integrated into standard make files. The example designs provided in `<sdx_root>/samples` each contain a Makefile that demonstrates command line invocation. In addition, the SDSoC™ IDE automatically generates makefiles that invoke `sds++` for all C++ files and `sdscc` for all C files, but the only source files that must be compiled with `sds++` are those containing code that:

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

A large software project can include many files and libraries that are unrelated to the hardware accelerator and data motion networks generated by `sds++`. All other source files can safely be compiled with the Arm® GNU toolchain.

If the `sds++` compiler issues errors on source files unrelated to the generated hardware system (for example, from an xfOpenCV library), you can compile these files through `gcc/g++` instead of `sds++`.

The `sdscc/sds++ -help` option provides a listing of available system compiler options and a brief description.

```
sdscc -help
sds++ -help
```

You can see the list of available platforms with their names, description, and system configurations by using the `-sds-pf-list` option as shown in the following code:

```
sds++ -sds-pf-list
```

Getting platform details such as the specific Xilinx device, available clock frequencies, and clock identifiers are shown when you use the following command:

```
-sds-pf-info
```

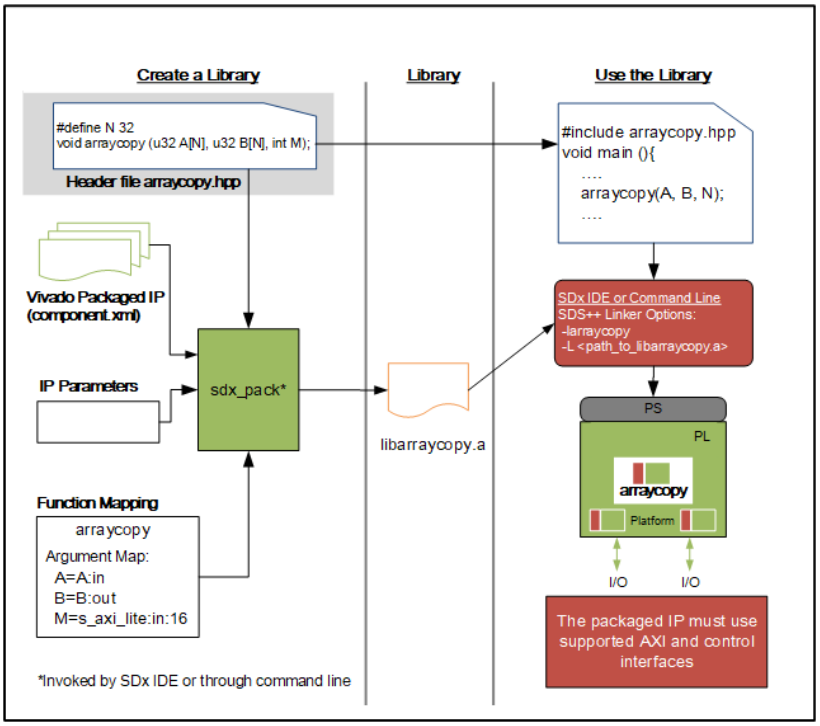
For example, you can query details about the `zcu102` platform by substituting the name `zcu102` for the `platform_name` in the following command:

```
sds++ -sds-pf-info <platform_name>
```

C-Callable IP Libraries

This section describes how to create a C-Callable IP library for IP blocks written in a hardware description language like VHDL or Verilog. User applications can statically link with such libraries using the SDSoC™ system compiler, and the IP blocks are instantiated into the generated hardware system.

Figure 29: **Create and Use a C-Callable Library**



SDSoC applications can use C-Callable IP libraries to access Vivado® Xilinx® packaged IP blocks written in a hardware description language (HDL) such as VHDL or Verilog or with a high-level synthesis tool like Vivado HLS. At times, hardware specific optimizations or microarchitectures are easier to implement through an HDL and can be delivered encapsulated within a C-Callable library.

Using a C-Callable IP library provides both the design reuse flexibility of a software library and the performance gain of optimized hardware IP blocks. With a bottom-up approach individual IP blocks can be designed, implemented, and tested prior to being placed into a C-callable library for broader use. The library of hardware-accelerated functions allows a means to insulate hardware and software development teams from low-level implementation details while still ensuring that both teams are cognizant of the functional interfaces.

Creating C-Callable IP Libraries

The SDx™ installation contains examples of C-Callable IP in the `<SDx_Install_Dir>/samples/rtl` directory. The `axis_arraycopy` example contains the Vivado®-packaged IP and the files used to create the `libarraycopy.a` library. The `arraycopy.hpp` header file contains the software function declarations associated with the hardware functionality provided in the packaged IP. The `component.xml` contains the meta-data used by the SDx IDE and the underlying `sdx_pack` tool to build the library.

Begin by launching the SDx IDE and specifying a workspace (for example: **sdx_workspace**) to create an Application project (**File** → **New** → **SDx Project**) with an **Output type** of C-Callable IP Library.

The project name becomes the name of the library created by concatenating the prefix `lib` and the `.a` suffix (`lib<project_name>.a`).

The following tables provide the actions necessary to create the library with the SDx™ IDE. The top row of each table contains the SDx IDE menu selection to begin each task.

- The **Dialog** column lists the names of the subsequent dialog boxes that open.
- **Selection** and **Action** columns indicate how to fill out the dialog boxes to complete the task.

The tables cover the following steps:

1. Create a C-Callable IP Library application project.
   - Each library is created for a specific `board_family`, `cpu_type`, `OS_type` tuple.
2. Import source files from the `samples/rtl` directory in the SDx installation.
   - Import both the header file and the packaged IP.
3. Identify the header file (`.hpp`) and the IP meta-data file (`component.xml`) to use as inputs to build the C-Callable IP library.

IMPORTANT!: *Ensure Step 2 (importing source files) is done prior to this step.*

4. Indicate how the arguments of each function in the C-Callable IP library maps to the hardware IP.

Table 4: **File → New → SDx Project**

File > New > SDx Project		
Dialog	**Selection or Field Name**	**Action**
Project Type	**Application**	Click **Next.**
Create a New SDx Project	**Project name:**	`arraycopy`
	Use default location	Use the check-mark.
		Click **Next.**
Platform	**Type**:	**Platform**
	Name	**zc702**
		Click **Next.**
System Configuration	**System configuration:**	**Linux**
	Runtime:	**C/C++**
	Domain:	**linux**
(pre-set)	**CPU:**	**cortex-a9**
(pre-set)	**OS:**	**linux**
	Linux Root File System:	Leave un-checked.
	Output type	**C-Callable Library**
	Allow hardware acceleration	check-mark (default)
		Click **Next.**
Templates	**Empty Application**	Click **Finish.**

Table 5: **Import Sources Dialog Options**

Import Sources		
Dialog	**Selection or Field Name**	**Action**
File system	From directory:	Browse to `axis_arraycopy` directory.
		Click **OK.**
	Files: `src/arraycopy.hpp`	check-marked.
	Directory: `ip`	check-marked.
	Into folder:	`arraycopy/src`
		Click **Finish**.

Table 6: **IP Customization → Add IP Customizations**

IP Customization window, Click Add IP Customizations (File icon w/ stylized "h")		
Dialog	**Selection or Field Name**	**Action**
Add IP Customizations	**Header File:**	Click **Browse.**
	Files:	Select `arraycopy.hpp`.

Table 6: **IP Customization → Add IP Customizations** *(cont'd)*

IP Customization window, Click Add IP Customizations (File icon w/ stylized "h")		
Dialog	**Selection or Field Name**	**Action**
	Qualifier:	Select `src/src/arraycopy.hpp`.
		Click **OK.**
	IP Path:	Click **Browse.**
	Files:	Select `component.xml`.
	Qualifier:	Select `src/ip/component.xml`.
		Click **OK.**
	Accelerator control:	Protocol: **AXI**
		Port: **s_axi_lite**
		Offset: **0.**
		Click **OK.**

This `axis_arraycopy` example uses the contents of the provided `samples/rtl/axis_arraycopy/src/Makefile` to complete the dialog box option.

The `component.xml` and, if provided, the `register_map.txt` files associated with the IP block can also be queried for information on the how the function arguments map to the hardware.

Table 7: **IP Customization → Add Function Mapping**

IP Customization → Add Function Mapping (+"icon)		
Dialog	**Selection or Field Name**	**Action**
Add Function Mapping	**Function name:**	Click the **++** icon on the right side.
		Select `arraycopy`.
		Click **OK.**
	Arguments mapped to AXILiteAXI4-Lite Interface:	Click **Add Function Argument Map** ("+" icon)
	Argument:	**M**
	AXILiteAXI4-Lite interface:	**s_axi_lite**
	Direction:	**IN**
	Offset:	**16**
		Click **OK.**
	Arguments mapped to `AXIS` interface:	Click **Add Function Argument Map.** ("+" icon)
	Argument:	**A**
	`AXIS` interface:	**A**
	Direction	**IN**

Table 7: **IP Customization → Add Function Mapping** *(cont'd)*

IP Customization → Add Function Mapping (+"icon)		
Dialog	**Selection or Field Name**	**Action**
		Click **OK.**
	Arguments mapped to `AXIS` interface:	Click **Add Function Argument Map** ("+" icon)
	Argument:	**B**
	Empty Application	**AXIS interface: B**
	Direction:	**OUT**
		Click **OK.**
	Complete the Add Function Mapping dialog box.	Click **OK.**

The C-Callable IP library is generated and placed in the build output directory of the application (for example, the `debug` directory) with the name `lib<application>.a` (`libarraycopy.a` for this example).

In addition to the SDx IDE method of creating a C-Callable IP library, a command line method that directly invokes the `sdx_pack` tool is available. The equivalent `sdx_pack` command to match the actions taken with the SDx IDE for the `axis_arraycopy` example is:

```
sdx_pack -header arraycopy.hpp -lib libarraycopy.a \
-func arraycopy -map A=A:in -map B=B:out -map M=s_axi_lite:in:16 -func-end
\
-ip ../ip/component.xml -control AXI=s_axi_lite:0 \
-target-family zynq -target-cpu cortex-a9 -target-os standalone \
-verbose
```

***Note*:** The C-Callable function and its argument map is listed between `-func` and `-func-end` options of the `sdx_pack` call.

The `axis_arraycopy` example is a library with a single accelerator function. Other examples, in particular the ones that begin with the `mfa_` prefix are multi-function accelerator (MFA) libraries where more than one function is mapped onto one IP block. The `sdx_pack` command for the `mfa_scalar_128_none` example that generates the `libmfa.a`.

The C-Callable IP library contains eight (8) functions and is shown in the following code example. This accelerator library uses control protocol `none`, indicating that the user explicitly controls the IP. This MFA example also demonstrates 128-bit scalar function arguments that map to AXI4-Lite interfaces as well as 128-bit array arguments that map to master AXI4-Stream interfaces.

```
sdx_pack -header mfa.hpp -I inc -lib libmfa.a \
-func mfa_reset -map inst=s_axi_AXILiteS:in:0x40 -func-end \
-func mfa_init -map inA=inA:in -map inst=s_axi_AXILiteS:in:0x40 -func-end \
-func mfa_copy -map outB=outB:out -map inst=s_axi_AXILiteS:in:0x40 -func-
end \
-func mfa_sum -map result=axi_AXILiteS:out:0x2c -map
```

```
inst=s_axi_AXILiteS:in:0x40 -func-end \
-func mfa_status -map return=axi_AXILiteS:out:0x10 -map
inst=s_axi_AXILiteS:in:0x40 -func-end \
-func mfa_status2 -map status=s_axi_AXILiteS:out:0x10 -map
inst=s_axi_AXILiteS:in:0x40 -func-end \
-func mfa_result -map result=s_axi_AXILiteS:out:0x2c -func-end \
-func mfa_stop -map inst=s_axi_AXILiteS:in:0x40 -func-end \
-ip ../ip/component.xml -control none \
-add-ip-repo ../dummy_ip  \
-add-ip-repo ../dummy_ip_repo \
-target-family zynq -target-cpu cortex-a9 -target-os linux -verbose
```

Another example of a MFA type of C-Callable library, is the `mfa_scalar_axi` accelerator. This accelerator library uses a generic AXI4-Lite control protocol, and shows the use of scalar function arguments that map to AXI4-Lite interfaces as well as array arguments that map to master AXI4 interfaces.

```
sdx_pack -header mfa.hpp -I inc -lib libmfa.a \
    -func mfa_reset -map status=s_axi_AXILiteS:out:0x20 -map
inst=s_axi_AXILiteS:in:0x34 -func-end \
    -func mfa_init -map inA=s_axi_AXILiteS:in:0x10,m_axi_inA:in -map
status=s_axi_AXILiteS:out:0x20 \
          -map inst=s_axi_AXILiteS:in:0x34 -func-end \
    -func mfa_copy -map outB=s_axi_AXILiteS:in:0x18,m_axi_outB:out -map
status=s_axi_AXILiteS:out:0x20 \
          -map inst=s_axi_AXILiteS:in:0x34 -func-end \
    -func mfa_sum -map result=s_axi_AXILiteS:out:0x28 -map
status=s_axi_AXILiteS:out:0x20 \
          -map inst=s_axi_AXILiteS:in:0x34 -func-end \
    -func mfa_status -map status=s_axi_AXILiteS:out:0x20 -func-end \
    -func mfa_stop -map status=s_axi_AXILiteS:out:0x20 -map
inst=s_axi_AXILiteS:in:0x34 -func-end \
    -ip ../ip/component.xml -control AXI=s_axi_AXILiteS:0x0 \
    -target-family zynq -target-cpu cortex-a9 -target-os linux -verbose
```

A register map showing the bit-level definition of the AXI4-Lite control protocol signals used in the `mfa_scalar_axi` example is provided in the `mfa_scalar_axi/ip/register_map.txt` file and excerpted below. In general, IP register mapping information is provided by the IP developer.

```
// ==============================================================
// File generated by Vivado(TM) HLS - High-Level Synthesis from C, C++ and
SystemC
// Version: 2018.2
// Copyright (C) 1986-2018 Xilinx, Inc. All Rights Reserved.
//
// ==============================================================

// AXILiteS
// 0x00 : Control signals
//        bit 0  - ap_start (Read/Write/COH)
//        bit 1  - ap_done (Read/COR)
//        bit 2  - ap_idle (Read)
//        bit 3  - ap_ready (Read)
//        bit 7  - auto_restart (Read/Write)
//        others - reserved
// 0x04 : Global Interrupt Enable Register
```

```
//        bit 0  - Global Interrupt Enable (Read/Write)
//        others - reserved
// 0x08 : IP Interrupt Enable Register (Read/Write)
//        bit 0  - Channel 0 (ap_done)
//        bit 1  - Channel 1 (ap_ready)
//        others - reserved
// 0x0c : IP Interrupt Status Register (Read/TOW)
//        bit 0  - Channel 0 (ap_done)
//        bit 1  - Channel 1 (ap_ready)
//        others - reserved
// 0x10 : Data signal of inA_offset
//        bit 31~0 - inA_offset[31:0] (Read/Write)
// 0x14 : reserved
// 0x18 : Data signal of outB_offset
//        bit 31~0 - outB_offset[31:0] (Read/Write)
// 0x1c : reserved
// 0x20 : Data signal of status
//        bit 31~0 - status[31:0] (Read)
// 0x24 : Control signal of status
//        bit 0  - status_ap_vld (Read/COR)
//        others - reserved
// 0x28 : Data signal of result
//        bit 31~0 - result[31:0] (Read)
// 0x2c : Data signal of result
//        bit 31~0 - result[63:32] (Read)
// 0x30 : Control signal of result
//        bit 0  - result_ap_vld (Read/COR)
//        others - reserved
// 0x34 : Data signal of inst
//        bit 31~0 - inst[31:0] (Read/Write)
// 0x38 : reserved
// (SC = Self Clear, COR = Clear on Read, TOW = Toggle on Write, COH =
Clear on Handshake)
```

The `mfa_fir` C-Callable IP library example highlights instantiating IP parameters, using AXI4-Stream interfaces, a 24-bit data type, and a control protocol selection of **none**.

```
ibfir.a: fir.hpp
   sdx_pack -header fir.hpp -lib libfir.a \
   -func fir -map X=S_AXIS_DATA:in -map Y=M_AXIS_DATA:out -func-end \
   -func fir_reload -map H=S_AXIS_RELOAD:in -func-end \
   -func fir_config -map H=S_AXIS_CONFIG:in -func-end \
   -ip ../ip/fir_compiler_v7_2/component.xml -control none \
   -param DATA_Has_TLAST="Packet_Framing" \
   -param M_DATA_Has_TREADY="true" \
   -param Coefficient_Width="8" \
   -param Data_Width="8" \
   -param Quantization="Integer_Coefficients" \
   -param Output_Rounding_Mode="Full_Precision" \
   -param Coefficient_Reload="true" \
   -target-family zynq -target-cpu cortex-a9 -target-os linux -verbose
```

Considerations for C-Callable IP Libraries are, as follows:

1. Function arguments of `TYPE *a` or `TYPE &a` are interpreted as an `OUTPUT` scalar.
2. Arrays must be declared as `TYPE a[N]` or `TYPE a[]`.

3. Function return type can only be a scalar in the format of `TYPE: TYPE*` or `TYPE&` are not allowed.
4. C-Callable IP library header files cannot have SDS pragmas that use `MACRO` as parameters: `#pragma sds data copy (A[0:SIZE])` is not allowed when `SIZE` is a macro (for example: `#define SIZE 16)`.
5. Overlapped function calls of multi-function accelerators (MFAs) are not allowed, as there is only one IP instance in the hardware; therefore, `async` pragmas around MFA functions are very risky and not recommended unless there is no chance of overlapping during run-time.
6. Argument sizes must match between the software declared argument and the hardware port pair: `<project_name>/Debug/reports/sdx_pack.html report` file can be used to double-check if the library implemented the expected argument sizes, offsets, and bus interfaces.
7. If a function of an MFA is called twice, the first time it forms a direct connection, and the second time it does not. Add the `#pragma SDS resource(ID)"` around each function call to ensure that the tool generates the correct hardware.

The following are examples of the `sdx_pack` command:

```
sdx_pack -header<header.h/pp>-ip <component.xml>
```

```
[-param <name>="value"] [configuration options]
```

The following table provides further details on the `sdx_pack` tool.

sdx_pack Command Options

Option	Description
`-header header.h/.hpp`	(required) Header file (`.h`, `.hpp`) with function prototypes. sdx_pack generates C++ style library for a `.hpp` file and a C-style library for a `.h` file. Only one top-level header file is allowed per library. The top-level header can include other header files, using the `-I` option.
`-ip component.xml`	(required) Path to Vivado®packaged IP. Only one top-level IP per library. The top-level IP can invoke other IP blocks, using the `-add-ip-repo` option.

Option	Description
`-control protocol [=port [:offset]]`	(required) IP control protocol options: • `AXI` Automatic control protocol based on an AXI4-Lite control register, typically at offset `0x0` (for example: `-control AXI=s_axi_AXILiteS:0`) • `none` User application explicitly controls the IP (for example `-control none`)
`-func function_name -map swName=hwNAME:direction[:offset[aximm_name:direction]] -func-end`	(required) Specify a list of C-Callable IP functions. Each function is listed between a `-func` and `-func-end` option pair. Function arguments are mapped to each IP port with the `-map` option. The `-map` option is then used as follows: • `Scalars` map onto an AXI4-Lite interface ◦ Map input scalar (example: `int a`) with `-map a=s_axi_AXILiteS:in:offset` ◦ Map output scalar (for example: `int *a`, or `int &a`) with `-map a=s_axi_AXILiteS:out:offset` ◦ Map function return scalar (return type can only be a scalar) with `-map return=s_axi_AXILiteS:out:offset` • `Arrays` map onto AXI4, AXI4-Stream, or AXI4-Lite interfaces. The arrays must be one-dimensional (for example: `int a[N]`, or `int a[]`). ◦ Map to AXI4, with `-map a=s_axiAXILiteS:in:offset,a_hwName:direction` - Not allowed when `control=none`The first part is mapping to the address, and the second part is mapping to the data port. ◦ Map to AXI4-Stream with `-map a=a_hwName:direction.` ◦ Map to AXI4-Lite with `-map a=s_axi_AXILiteS:in:offset`. Array must be one-dimensional and of constant size.
`-param name="value"`	IP parameter name-value pairs to instantiate IP parameters. Use one `-param` option per pair.
`-lib libname`	Use specified `libname` for naming the generated library. By default `lib header.a` is used.
`-I path`	If the file named with the `-header` option includes other files, this option specifies the path to the additionally included files. Multiple `-I` options can be used. For easier library distribution, place all include files into a single directory.

Option	Description
`-add-ip-repo path`	Add all IP found in the listed repository into the library. Although multiple `-add-ip-repo` options can be used to specify multiple paths. It is recommended to place all required IP into a single directory and use a single `-add-ip-repo` option.
`-target-family board_family`	The target board family supported by the IP (for example: `zynq` (default), `zynquplus`).
`-target-cpu cpu_type`	Specify target CPU: • `cortex-a9` (default) • `cortex-a53` • `cortex-r5` • `microblaze`
`-target-os name`	Specify target OS: • `linux` (default) • `standalone` (bare-metal)
`-verbose`	Print verbose output to `STDOUT`.
`-version`	Print the `sdx_pack` version information to `STDOUT`.
`-h, -help, --help`	Display `sdx_pack` option usage and descriptions.

Example:

```
sdx_pack -header arraycopy.hpp -lib libarraycopy.a \
-func arraycopy -map A=A:in -map B=B:out \
-map M=s_axi_lite:in:16 -func-end \
-ip ../ip/component.xml -control AXI=s_axi_lite:0 \
-target-family zynq -target-cpu cortex-a9 -target-os standalone \
-verbose
```

Where:

- `arraycopy.hpp` specifies the header file defining the function prototype for the `arraycopy` function.
- `component.xml` of the IP generates the packaged Vivado® IP for SDx.
- `-control` specifies the IP control protocol.
- `-map` specifies the mapping of an argument from the software function to a port on the Vivado IP. Notice the option is used three times in the example above to map function arguments `A, B` and `M` to IP ports.
- The `-target-os` option specifies the target operating system.

The `sdx_pack` utility generates a C-Callable IP library to match the name in the `-lib` option, `libarraycopy.a` in this case.

Using C-Callable IP Libraries

After you generate the C-Callable library, create a new SDSoC™ Application project to use the library. The **Output** type for this application is an executable file (ELF), and is linked with the C-Callable IP Library.

1. Create an SDx application project to output an executable file.

***Note*:** The tuple consisting of `board_family`, `cpu_type`, and `os_type` must match that of the C-Callable IP library.

Table 8: **SDx Project**

File > New > SDx Project		
Dialog Box	**Selection or Field Name**	**Action**
Project Type	**Application**	Click **Next**.
Create a New SDx Project	**Project name:**	`app_arraycopy`
	Use default location	check-marked
		Click **Next**.
Platform	**Type:**	**Platform**
	Name	`zc702`
		Click **Next**.
System Configuration	**System configuration:**	**Linux**
	Runtime:	**C/C++**
	Domain:	**linux**
(pre-set)	**CPU:**	**cortex-a9**
(pre-set)	**OS:**	**linux**
	Linux Root File System:	un-checked
	Output type	Executable (elf)
	Allow hardware acceleration	check-marked (default)
		Click **Next**.
Templates	**Empty Application**	Click **Finish**.

2. Import the function declarations header file (`.hpp`) common to both the library and the application.

Table 9: **Select Import Sources**

Project Explorer window, Right-Click app_arraycopy, Select Import Sources		
Dialog	**Selection or Field Name**	**Action**
File system	From directory:	Browse to **axis_arraycopy/src** directory.
		Click **OK.**
	Files: arraycopy.hpp	check-marked
	Into folder:	`app_arraycopy/src`
		Click **Finish**.

3. Open the Importing Sources dialog box again to get the main application code.

Table 10: **Select Import Sources**

Project Explorer window, Right-Click app_arraycopy, Select Import Sources		
Dialog	**Selection or Field Name**	**Action**
File system	**From directory:**	Browse to the `axis_arraycopy/app` directory.
		Click **OK**.
	Files: main.cpp	check-marked
	Into folder:	`app_arraycopy/src`
		Click **Finish**.

Now that imported the source files for the `app_arraycopy` application are imported, update the **C/C++ Build Settings** to have the linker use the `arraycopy.a` C-Callable library when building the application.

4. Update the **C/C++ Build Settings** as shown in the following table:

Table 11: **C/C++ Build Settings**

Project Explorer window, Right-Click app_arraycopy, Select C/C++ Build Settings		
Dialog	**Selection or Field Name**	**Action**
Settings → Tool Settings	**SDS++ Linker**	Select libraries.
		Click the **Add** symbol (with "+" icon) in the Libraries (`-l`) window.
	Libraries(-l)	`arraycopy`
		Click **Add** symbol (with "+" icon) in the Library search path (`-L`) window
		Click **Workspace**.
	Folder:	Navigate to and select **arraycopy/Debug**
		Click **OK**.

Table 11: **C/C++ Build Settings** *(cont'd)*

Project Explorer window, Right-Click app_arraycopy, Select C/C++ Build Settings		
Dialog	**Selection or Field Name**	**Action**
(pre-set)	**Directory:**	`${workspace_loc:/arraycopy/Debug}`
		Click **OK**.
		Click **Apply**.
		Click **OK**.

Building the Application in the Assistant Window

You can use the Assistant window to build the `app_arraycopy` application. In the Assistant window:

Under **app_arraycopy[SDSoC]**, right-click **Debug[Hardware]** and select **Build**.

The Console window shows the build progression including the `sds++` system compiler invocation.

After the application successfully builds the target executable file (`app_arraycopy.elf`), the Assistant window populates with a **Data Motion Network** Report, the **Compilation Log**, and an **SD Card Image** menu. Through the **SD Card Image** menu, the contents of the generated (`sd_card`) files directory is available for you view using the Project Explorer, a file browser, or a command shell window.

After the build completes, you can write the contents of the generated `sd_card` directory to the root of a FAT32-formatted SD card and boot and run the `app_arraycopy.elf` application on a ZC702 board. The `sd_card` directory includes a `README.txt` for boot setup instructions, a bootable `BOOT.BIN` file, and the `image.ub` file used to boot Linux.

The SDx IDE builds the application with the `sds++` system compiler using the C-callable library and the application code. The main application is compiled to produce an object file and then it is linked with the C-callable library (`arraycopy`). The issued commands are as follows:

Examples

Compilation of `main.cpp`:

```
sds++ -Wall -O0 -g -I../src -c -fmessage-length=0 -MTsrc/main.o -MMD -MP -
MFsrc/main.d \
 -MTsrc/main.o -o src/main.o ../src/main.cpp \
 -sds-sys-config linux -sds-proc linux -sds-pf zc702
```

Linking `main.o` with `arraycopy` library to produce executable application `app_arraycopy.elf`:

```
sds++ -L <path_to_arraycopy/Debug> --remote_ip_cache ../ip_cache \
 -o app_arraycopy.elf ./src/main.o -l arraycopy -dmclkid 2 \
 -sds-sys-config linux -sds-pf zc702
```

Chapter 5

Debugging Techniques

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

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

***Note*:** As the debug environment is initialized, Xilinx® recommends that users switch to the **Debug** perspective when prompted by the SDx IDE.

The debug perspective view provides the ability to debug the standard C/C++ portions of the application, by single-stepping code, setting and removing breakpoints, displaying variables, dumping registers, viewing memory, and controlling the code flow with "run until" and "jump to" type debugging directives. Inputs and outputs can be observed pre- and post- function call to determine correct behavior.

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

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

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

Chapter 6

Profiling and Optimization

There are two distinct areas for you to consider when performing algorithm optimization in the SDSoC™ environment:

- Application code optimization
- Hardware function optimization

Most application developers are familiar with optimizing software targeted to a CPU. This usually requires programmers to analyze algorithmic complexities, overall system performance, and data locality. There are many methodology guides and software tools to guide the developer identifying performance bottlenecks. These same techniques can be applied to the functions targeting hardware acceleration in the SDSoC environment.

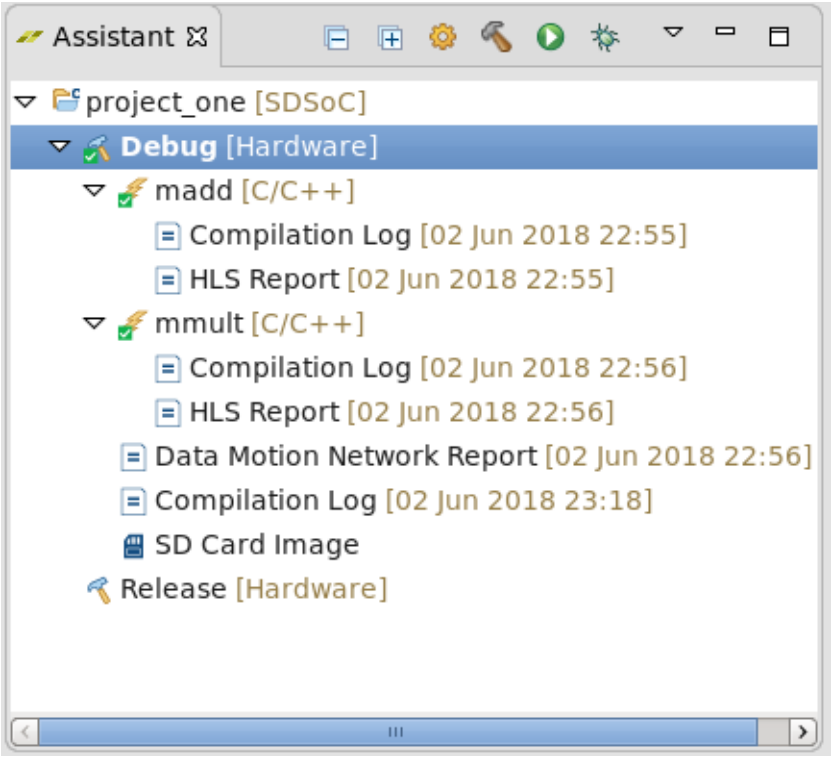
As a first step, programmers should optimize their overall program performance independently of the final target. The main difference between SDSoC and general purpose software is: In SDSoC projects, part of the core compute algorithms are pushed onto the FPGA. This implies that the developer must also be aware of algorithm concurrency, data transfers, memory usage and consumption, and the fact that programmable logic is targeted.

Generally, you need to identify the section of the algorithm to be accelerated and how best to keep the hardware accelerator busy while transferring data to and from the accelerator. The primary objective is to reduce the overall computation time taken by the combined hardware accelerator and data motion network versus the CPU software only approach.

Software running on the CPU must efficiently manage the hardware function(s), optimize its data transfers, and perform any necessary pre- or post- processing steps.

The SDSoC environment is designed to support your efforts to optimize these areas, by generating reports that help you analyze the application and the hardware functions in some detail. The reports are generated automatically when you build the project, and listed in the Assistant view of the SDx™ IDE, as shown in the following figure. Double-click a listed report to open it.

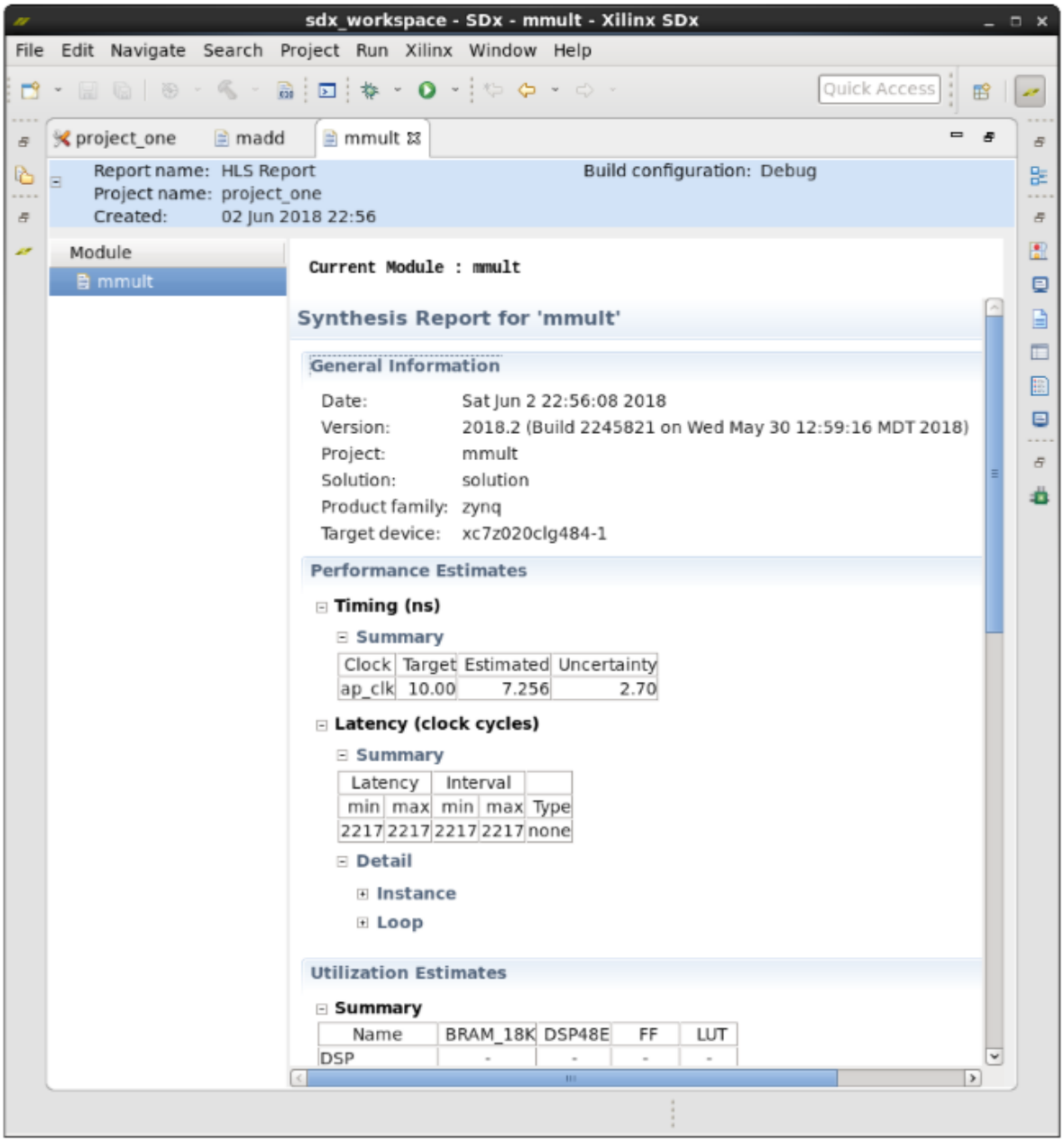
Figure 30: **Assistant View**



The following figures show the two main reports: the HLS Report, and Data Motion Network Report.

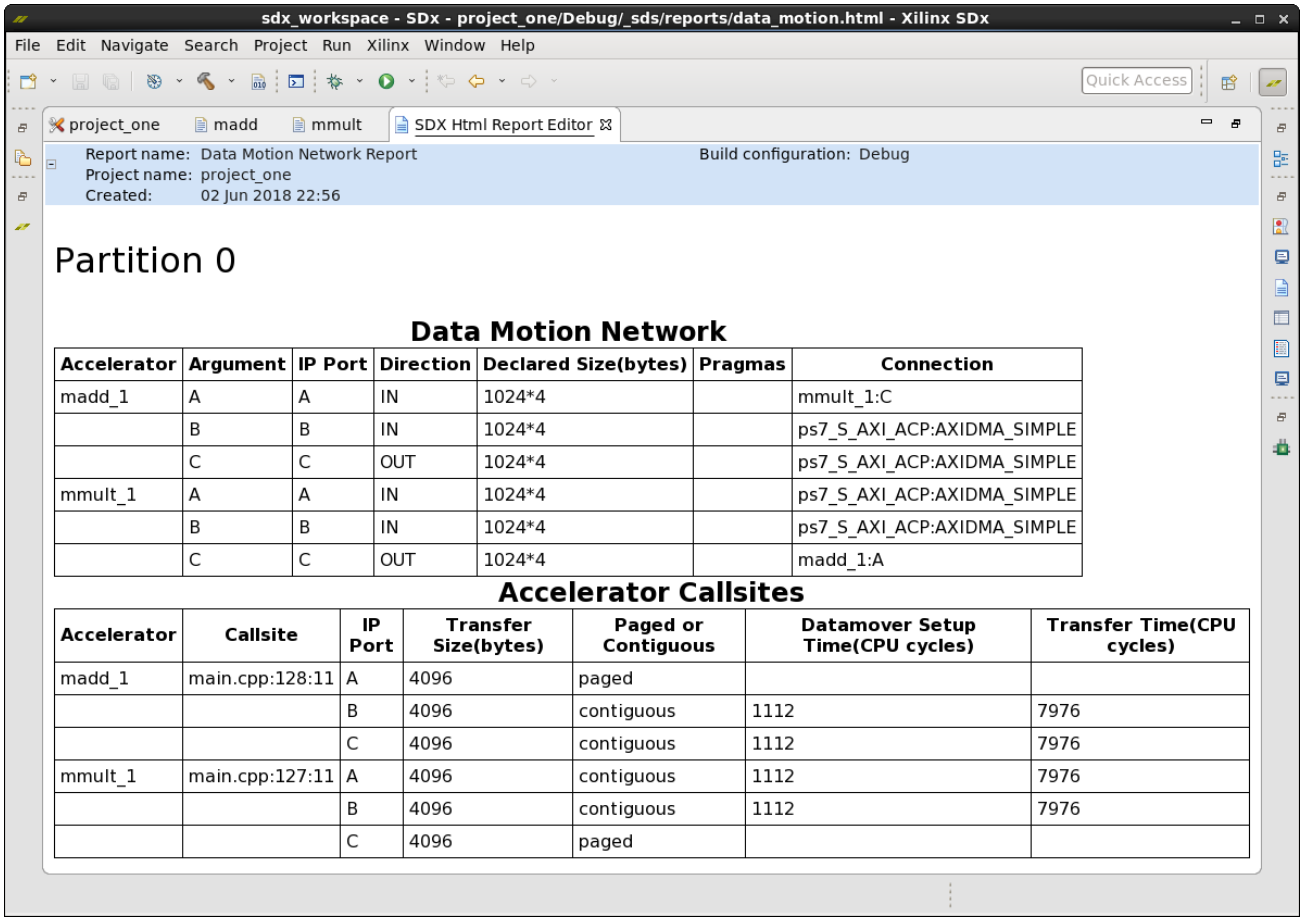
To access these reports from the GUI, ensure the **Assistant** view is visible. This view is typically below the **Project Explorer** view. You can use the **Window** → **Show View** → **Other** → **Xilinx** → **Assistant** menu command to display the Assistant view if it is not displayed.

Figure 31: **HLS Report Window**



The HLS Report provides details about the High-Level Synthesis process (HLS) program that translates the C/C++ model into a hardware description language responsible for implementing the functionality on the FPGA. The details of this report let you see the impact of the design on the hardware implementation. You can then optimize the hardware function(s) based on the information.

Figure 32: **Data Motion Network Report**



The Data Motion Network Report describes the hardware/software connectivity for each hardware function. The **Data Motion Network** table shows (from the right-most column to the left-most) what sort of data mover is used for transport of each hardware function argument, and to which system port that datamover is attached. The **Pragmas** shows any SDS-based pragmas used for the hardware function.

The **Accelerator Callsites** table shows the following:

- Accelerator instance name and Accelerator argument.
- Name of the port on the IP that pertains to the Accelerator argument (typically the same as the previous, except when bundling).
- Direction of the data motion transfer.
- Size, in bytes, of data to be transfered, to the degree in which the compiler can deduce that size. If the run time determines the transfer size, this is zero.

- List of pragmas related to this argument.
- System Port and data mover `<system port>:<datamover>`, if applicable. Indicates which platform port and which data mover is used for transport of this argument.
- Accelerator(s) that are used, the inferred compiler as being used, and the CPU cycles used for setup and transfer of the memory.

Generally, the Data Motion report page indicates first:

- What characteristics are specified in pragmas.
- In the absence of a pragma, what the compiler was able to infer.

The distinction is that the compiler might not be able to deduce certain program properties. In particular, the most important distinction here is cacheability. If the Data Motion report indicates cacheable, and the data is in fact uncacheable (or vice-versa), correct cache behavior would occur at run time; it is not necessary to structure your program such that the compiler can identify data as being uncacheable to remove flushes.

Additional details for each report, as well as a profiling and optimization methodology, and coding guidelines can be found in the *SDSoC Environment Profiling and Optimization Guide* (UG1235) .

Appendix A

Getting Started with Examples

All Xilinx® SDx™ environments are provided with example designs. These examples can:

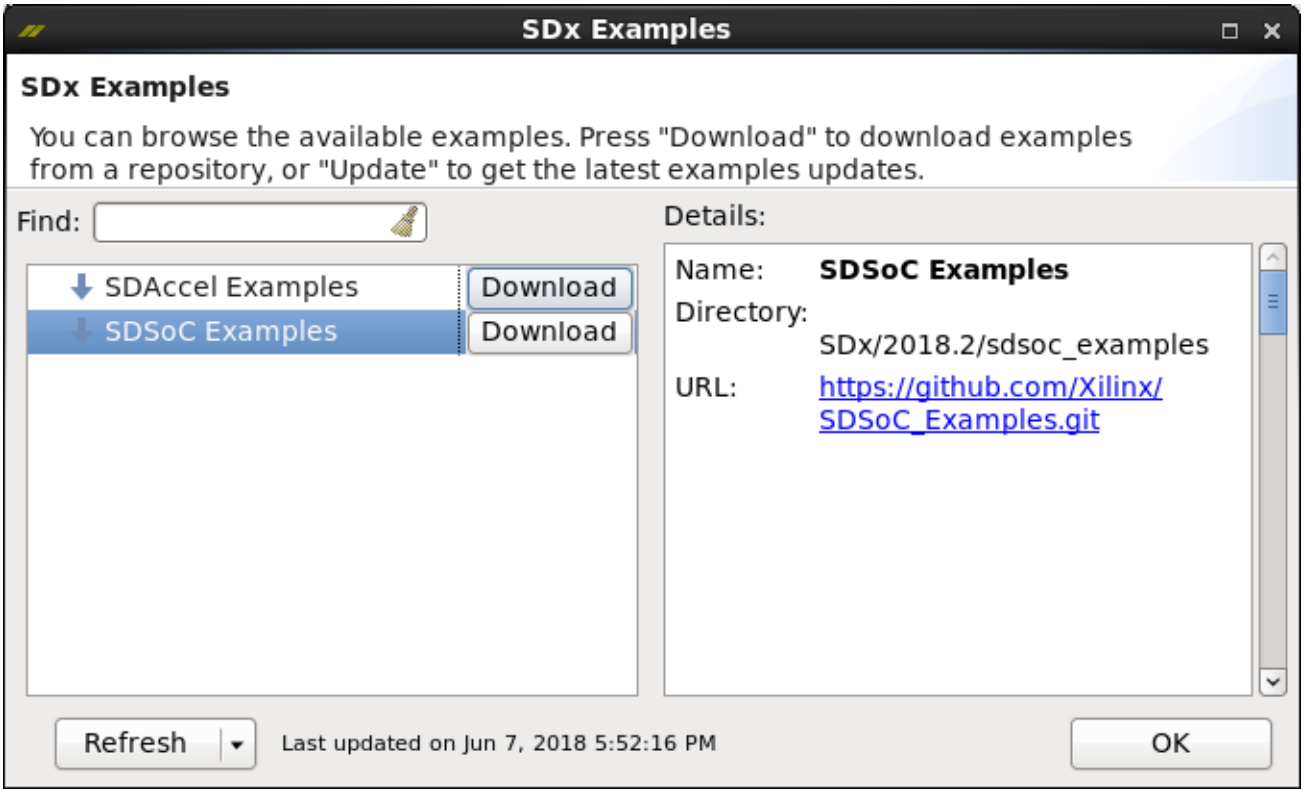
- Be a useful learning tool for both the SDx IDE and compilation flows such as makefile flows.
- Help you quickly get started in creating a new application project.
- Demonstrate useful coding styles.
- Highlight important optimization techniques.

Every platform provided within the SDx environment contains sample designs to get the user started, and are accessible through the project creation flow as described in Creating an Application Project. Furthermore, each of these designs, which are found in `<sdx_root>/samples` provides a Makefile so you can build, emulate, and run the code working entirely on the command line if you prefer. In addition, many examples are available to be downloaded from the Xilinx® GitHub repository, although a limited number are also included in the `samples` folder of the software installation.

Installing Examples

You will be asked to select a template for new projects when working through the **New SDx Project** wizard. You can also load template projects from within an existing project, by selecting **Xilinx → SDx Examples**.

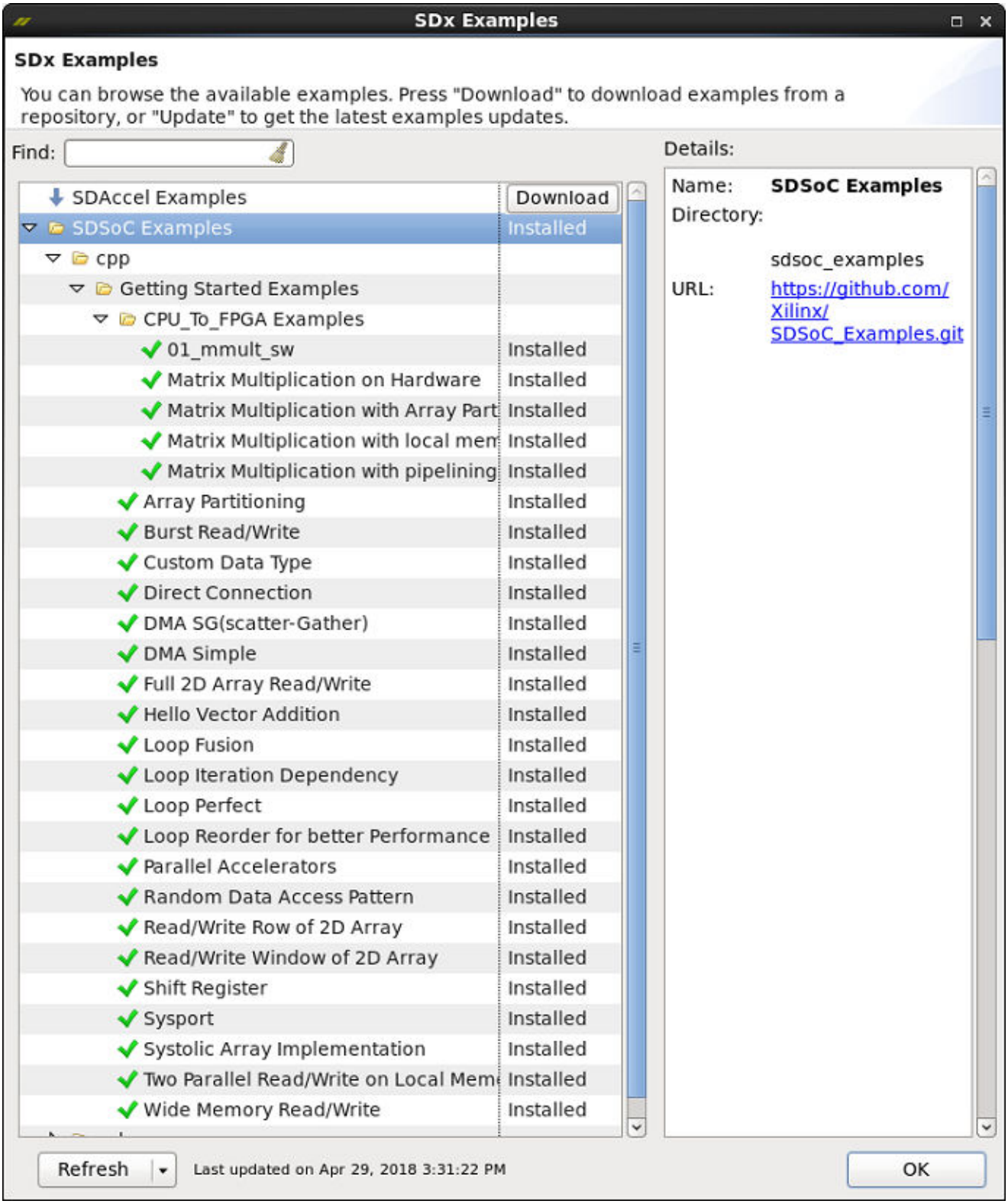
Figure 33: **SDSoC Examples - Empty**



The left side of the dialog box shows SDSoC Examples, and has a download command for each category. The right side of the dialog box shows the directory to where the examples downloaded and the URL from where the examples are downloaded

Click **Download** next to SDSoC Examples to download the examples and populate the dialog box. The examples are downloaded as shown in the following figure:

Figure 34: **SDSoC Examples -Populated**



The command menu at the bottom left of the SDx Examples dialog box provides two commands to manage the repository of examples:

- **Refresh:** Refreshes the list of downloaded examples to download any updates from the GitHub repository.
- **Reset:** Deletes the downloaded examples from the `.Xilinx` folder.

Using Local Copies

While you must download the examples to add Templates when you create new projects, the SDx™ IDE always downloads the examples into your local `.Xilinx/SDx/<version>`folder:

- On Windows: `C:\Users\<user_name>\.Xilinx\SDx\<version>`
- On Linux: `~/.Xilinx/SDx/<version>`

The download directory cannot be changed from the SDx Examples dialog box. You might want to download the example files to a different location from the `.Xilinx` folder. To do so, use the `git` command from a command shell to specify a new destination folder for the downloaded examples:

```
git clone https://github.com/Xilinx/SDSoC_Examples
<workspace>/examples
```

When you clone the examples using the `git` command as shown above, you can use the example files as a resource for application and kernel code to use in your own projects. However, many of the files use include statements to include other example files that are managed in the Makefiles of the various examples. These include files are automatically populated into the `src` folder of a project when the Template is added through the New SDx Project wizard. To make the files local, locate the files and manually make them local to your project.

You can find the needed files by searching for the file from the location of the cloned repository. For example, you can run the following command from the `examples` folder to find the `xcl2.hpp` file needed for the `vadd` example:

```
find -name xcl2.hpp
```

In addition, the Makefile in each of the example projects has a special command to localize any include files into the project. Use the following form of the `make` command in the example project:

```
make local-files
```

C++ Design Libraries

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

- reVISION resources:
  - reVISION Product Page
  - GitHub xfOpenCV
- Machine Learning-Deep Neural Network Accelerator Library
- Arbitrary Precision Data Types
- HLS Stream
- HLS Math
- HLS Video
- HLS IP
- HLS Linear Algebra
- HLS DSP

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

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

Wrapping HLS Functions

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

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

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

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

See also: Coding Guidelines in the *SDSoC Environment Programmers Guide* (UG1278).

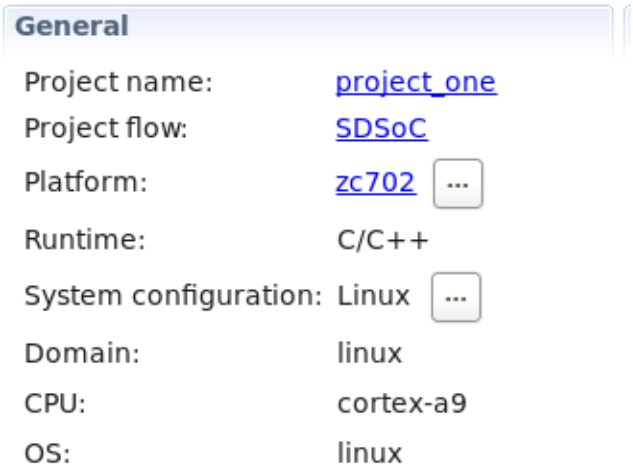
Appendix B

Managing Platforms and Repositories

The SDx™ environment comes with built-in platforms. If you need to use a custom platform for your project, you must make that platform available for application implementation.

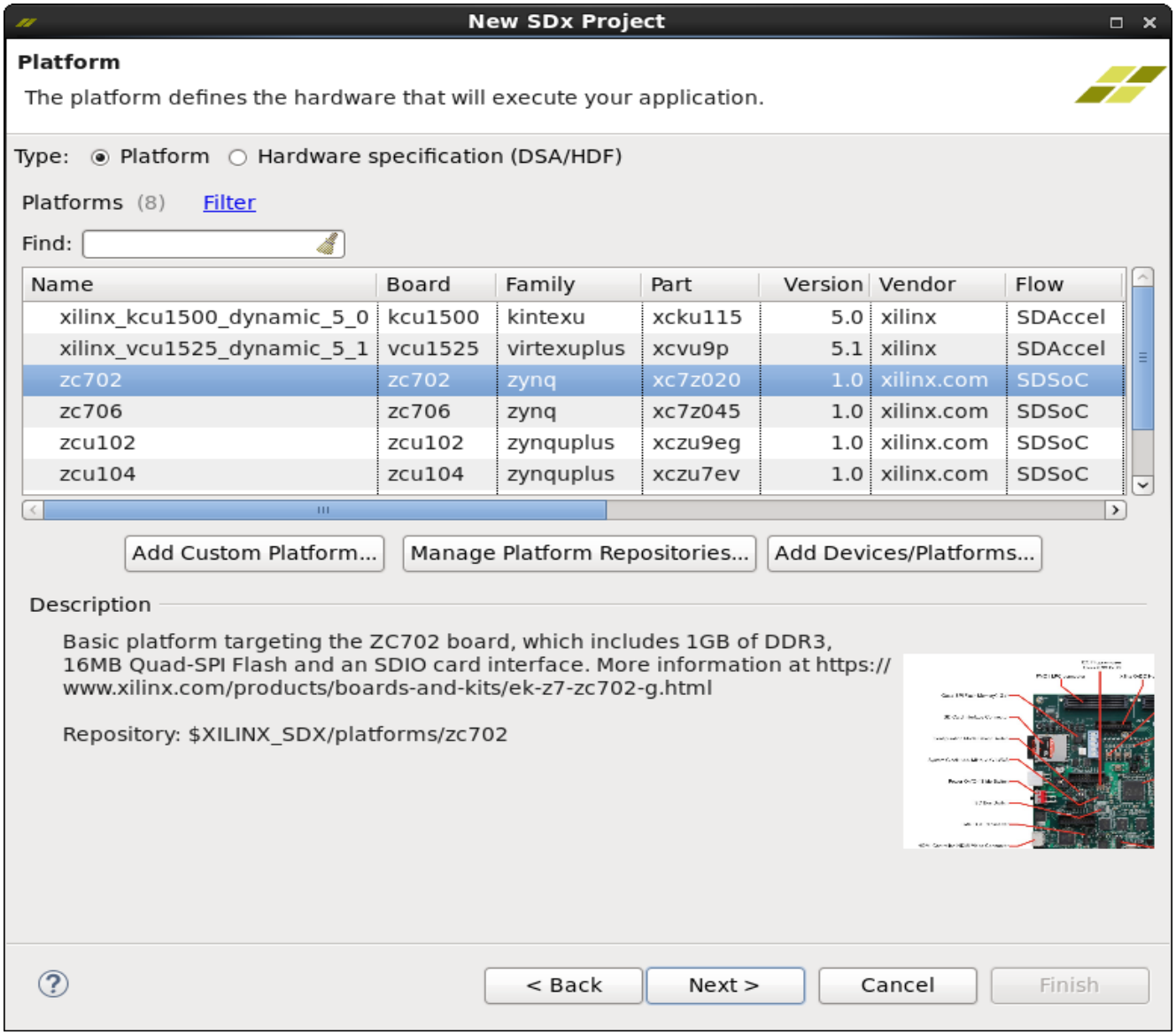
You can manage the platforms and repositories from an opened project by clicking **Browse** (...) next to the **Platform** link in the General panel of the Project Settings window.

Figure 35: **SDSoC Platform Browse**



This opens the Hardware Platforms dialog box, which lets you manage the available platforms and platform repositories.

Figure 36: **Platform Selection**



- **Add Custom Platform:** Lets you add your own platform to the list of available platforms. Navigate to the top-level directory of the custom platform, select it, and click **OK** to add the new platform. The custom platform is immediately available for selection from the list of available platforms. Select **Xilinx** → **Add Custom Platform** to directly add custom platforms to the tool.

- **Manage Repositories:** Lets you add or remove standard and custom platforms. If a custom platform is added, the path to the new platform is automatically added to the repositories. Removing any platform from the list of repositories removes the platform from the list of available platforms.
- **Add Devices/Platforms:** Lets you manage which Xilinx® devices and platforms are installed. If a device or platform was not selected during the installation process, you can add it at a later time using this command. This command launches the SDx Installer to let you select extra content to install. Select **Help** → **Add Devices/Platforms** to directly add custom platforms to the tool.

Appendix C

Compiling and Running Applications

This appendix contains the following topics:

- Compiling and Running Applications on a MicroBlaze Processor
- Compiling and Running Applications on an Arm Processor

RECOMMENDED: *When you make code changes, including changes to hardware functions, it is valuable to rerun a software-only compile to verify that your changes did not adversely change your program. A software-only compile is much faster than a full-system compile.*

Compiling and Running Applications on a MicroBlaze Processor

A MicroBlaze™ platform in the SDSoC™ Environment is a standard MicroBlaze processor system built using the Vivado® tools and SDK that must be a self-contained system with a local memory bus (LMB) memory, MicroBlaze Debug Module (MDM), UART, and AXI timer.

The SDSoC Environment includes the standard SDK toolchain for MicroBlaze processors, including `microblaze-xilinx-elf` for developing standalone ("bare-metal") and FreeRTOS applications.

By default, the SDSoC system compilers do not generate an SD card image for projects targeting a MicroBlaze platform. You can package the bitstream and corresponding ELF executable as needed for your application.

To run an application, the bitstream must be programmed onto the device before the ELF can be downloaded to the MicroBlaze core. The SDSoC Environment includes Vivado tools and SDK facilities to create MCS files, insert an ELF file into the bitstream, and boot the system from an SD card.

Compiling and Running Applications on an Arm Processor

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

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

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

For Arm Cortex™-A53 CPUs within the Zynq® devices, the SDSoC environment includes two toolchains:

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

For the Arm Cortex-R5 CPU provided in the Zynq UltraScale+ MPSoC, the following toolchain is include in the SDSoC environment: `armr5-none-eabi` - for developing standalone ("bare-metal") applications.

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

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

- `README.TXT`: Contains brief instructions on how to run the application
- `BOOT.BIN`: The boot image contains first stage boot loader (FSBL), boot program (U-Boot), and the FPGA bitstream.
- `image.ub`: Contains the Linux boot image. Platforms can be created that include the following:
  - `uImage`
  - `devicetree.dtb`

  - `uramdisk.image.gz` files
- `<app>.elf`: The application binary executable

To run the application:

1. Copy the contents of `sd_card` directory onto an SD card and insert into the target board.
2. Open a serial terminal connection to the target and power up the board.

   Linux boots, automatically logs you in as `root`, and enters a bash shell. The SD card is mounted at `/mnt`, and from that directory you can run `<app>.elf`.

Appendix D

Additional Resources and Legal Notices

Xilinx Resources

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

Solution Centers

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

Xilinx Resources

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

Documentation Navigator and Design Hubs

Xilinx® Documentation Navigator provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. The Xilinx Documentation Navigator (DocNav) is installed with SDSoC™ and SDAccel™. To open it:

- On Windows, select **Start** → **All Programs** → **Xilinx Design Tools** → **DocNav**.
- At the Linux command prompt, enter `docnav`.

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

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

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

References

These documents provide supplemental material useful with this guide:

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

Training Resources

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

Please Read: Important Legal Notices

OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.

Copyright