# **SDSoC Programmers Guide**

UG1278 (v2018.2) July 2, 2018





# **Revision History**

The following table shows the revision history for this document.

Section	Revision Summary	
07/02/2018 Version 2018.2		
	Initial Xilinx release.	



# **Table of Contents**

Revision History	2
Chapter 1: Introduction to Programming with SDSoC	5
Software Acceleration with SDSoC	
Execution Model of an SDSoC Application	
SDSoC Build Process	
SDSoC Programming Flow Overview	
Chapter 2: Programming the Embedded Processor Application	15
Memory Allocation	16
Sequential/Parallel Accelerator Execution	18
Validating the Software to Hardware Conversion	20
Performance Estimation	22
Chapter 3: Programming Hardware Functions	23
Coding Guidelines	24
Data Movers	27
The Function Body	29
Datatypes	29
Array Configuration	31
Loops	35
Dataflow Optimization	41
Chapter 4: Using External I/O	45
Accessing External I/O using Memory Buffers	45
Accessing External I/O using Direct Hardware Connections	48
Appendix A: Exporting a Library for GCC	50
Building a Shared Library	
Compiling and Linking Against a Library	53
Exporting a Shared Library	54
Appendix B: SDSoC Environment API	55



Appendix C: Additional Resources and Legal Notices	
Xilinx Resources	
Documentation Navigator and Design Hubs	
References	
Training Resources	58
Place Pard: Important Logal Natices	





# Introduction to Programming with SDSoC

The SDSoC™ environment provides tools for developing embedded systems in Xilinx® Zynq®-7000 SoC and Zynq UltraScale+ MPSoC, or MicroBlaze embedded processor on Xilinx devices.

#### It includes:

- An Eclipse-based integrated development environment (IDE) with compilers, debuggers, and profilers for Arm® and MicroBlaze processors.
- A hardware emulator.
- A hardware compiler that synthesizes C/C++ functions into optimized hardware functions to be used in the programmable logic (PL).
- A system compiler that generates complete hardware/software systems, including custom hardware accelerators and data mover hardware blocks (e.g. DMA engines), from application code written in the C/C++ programming languages.

The sdscc/sds++ (referred to as sds++) system compiler provides options to perform hardware/ software event tracing, which provides detailed timeline visibility into accelerator tasks running in hardware, data transfers between accelerators and memory, and application code running on the CPUs.

Xilinx FPGAs and SoC devices offer many advantages, including a programmable hardware architecture for implementing custom data paths, multi-level distributed memory architectures, and interfacing to custom input/output devices, with full customizability of the hardware / software interface to high-performance embedded CPUs. By building a custom hardware system, you can achieve higher performance and lower power dissipation for your embedded applications. What is unique about the SDSoC environment is the ability it provides the programmer to create hardware and software, while working within familiar software development workflows including cross-compiling, linking, profiling, debugging, and running application binaries on target hardware and in an emulator. Using the sds++ system compiler, you can target parts of your application to be implemented as hardware accelerators running many times faster than optimized code running on a processor.



The programmer's view of the target device is heterogeneous computing, where code written in C/C++ will be running on multi-core Arm CPUs, as well as in custom hardware accelerators, typically with a non-uniform memory architecture and custom interfaces to input/output devices. More attention to where code will run, how data is mapped into memory, and how hardware and software interact will allow for better performance of your application. In general, application code should reflect the heterogeneity of the target system. Take into consideration that C/C++ code compiled into hardware accelerators benefits from programming idioms that reflect microarchitecture details, while code running on CPUs benefits from idioms that reflect the instruction set, cache, and memory architecture. when working in the SDSoC environment, the hardware/software interface between CPU and hardware accelerators will be described through function calls, and APIs specific to the underlying devices. Majority of the code will access accelerators through function calls rather than device driver APIs, with the sds++ system compiler generating highly efficient access from userspace, automatically managing low level considerations like cache management through custom drivers provided by the system compiler.



# 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<sup>™</sup> 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).



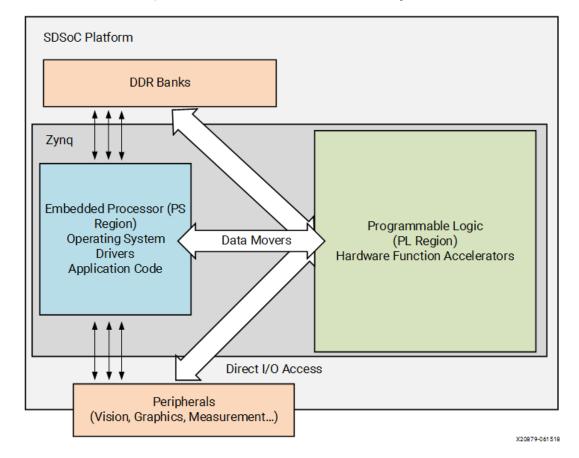


Figure 1: Architecture of an SDSoC System

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.

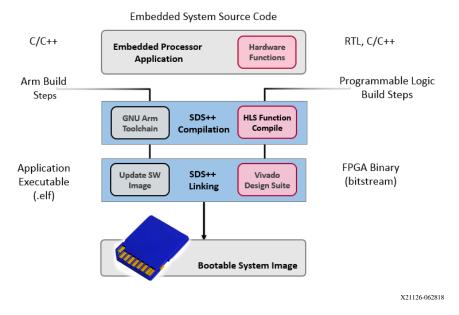


Figure 2: SDSoC Build Process

- 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 Programming Flow Overview**

The flow of developing an embedded system would follow the typical steps of: code development, compilation and link for the platform/device, profile the system for performance, and measure the actual performance. The SDSoC™ environment follows this standard software-centric flow, but also supports a more hardware-centric flow for defining hardware functions first, and then integrating those into the embedded application. What is unique about these two flows is that they are a heterogeneous programming model; meaning that writing code for the CPU side of the system is going to be different from writing the code for the programmable logic.

The software-centric approach focuses on the embedded processor application, and the acceleration of specific software functions into hardware functions running in the programmable logic (PL) region of the Xilinx® device. This requires converting the C or C++ code of the software function into Hardware Descriptive Language (HDL) that can be compiled for the programmable logic using Vivado® HLS.

A typical accelerated hardware function would be processor intensive (eg: complex computations that take a long time), processing lots of data. This code should be written so that data transfers are limited to streaming data into and from the accelerator, and should leverage instruction-level parallelism and task-level parallelism to take advantage of the massively parallel architecture of the programmable logic region of the device. The goal is to use parallelism to achieve the desired performance for accelerated functions. The goal of the accelerator would be to deliver, consume input data, process it, and output data as quickly as possible.

The performance of the accelerator is usually measured against the Initiation Interval (II), or the number of clock cycles before the function can accept new input data. The ideal would be one clock cycle. However, if the accelerator requires multiple clock cycles to input and process data, then achieving II=1 may not be possible, and it would also require more hardware resources for either buffering the input data, or for partitioning the data.

Once the processor and accelerator code is written, it can be compiled for emulation, or for compiling/linking to the hardware platform. For emulation, the code compiles faster allowing for quick design iterations, where it can be used to estimate performance as well as checking data integrity; but runs slower than on actual hardware. Emulation is very accurate to what will execute on the hardware, since the same CPU code will run both on QEMU and the target device. During emulation the communication to/from accelerators, and accelerator workloads, can be measured and monitored using specialized APIs in the sds\_lib.h or using the tracing feature of the SDSoC environment, which will measure transactional information between the CPU and accelerator(s) and provide a visual representation of these transactions. When building to the hardware platform, it will run exactly what is written for the processor and for the hardware accelerators. The benefits of running on hardware would be to measure actual runtime, as well as being able to adjust the builds later for in-circuit debugging, or performance analysis.



The hardware-centric approach is used by designers experienced with developing on an FPGA. This approach lets users control what functionality will be in the accelerator, and how data/commands will be transported between the logic and the CPU. This flow uses the Vivado Design Suite to create customized IP (Intellectual Property) containing AXI interfaces that are used to communicate between the PL region and the processor (PS). This IP can then be packaged with the  $sdx_pack$  command to map the IP's AXI interfaces to a header file to create a static library. Then using this resulting include file and static library is as simple as calling a typical library function. The key is to making sure that the header file's data width matches what is expected by the IP. refer to the SDSoC Environment User Guide (UG1027) for more information on creating and using C-Callable IP.





# Programming the Embedded Processor Application

Creating an application for an embedded processor in the SDSoC<sup>™</sup> environment, is similar to creating an application for any other SoC or embedded platform. However, there are some added considerations when accelerating an embedded processor application from SDK, for example. Programming for the SDSoC environment should include the following tasks:

- Identifying the appropriate function(s) in the processor application for acceleration in the programmable logic (PL) region.
- Allocating memory for the embedded processor application code and software functions running on the processing system (PS), and for the accelerated function(s) running on the PL regions of the device.
- Enabling task-level parallelism with multiple accelerators running concurrently to optimize system performance.
- Validating the software-to-hardware function code conversion, to insure things work as intended.

To identify the functions that should be accelerators, or turned into hardware functions, you should determine what kind of computational load would be required. For example, functions where large amounts of data are computed, or modified, would be good candidates for hardware functions. However, functions written for a typical processor application may not benefit from hardware acceleration in the SDSoC environment, and might need to be restructured to get real performance improvements from acceleration.

Although you might not typically allocate memory for standard processor applications, leaving it to the compiler to define, when programming for hardware acceleration you should manually define the memory allocation in the code. The hardware functions require physically contiguous memory to meet the performance requirements of the hardware.

In addition, when you have defined more than one software function for acceleration, you can also manage the scheduling of these accelerators, deciding if they can run concurrently, or need to be sequential. Understanding and managing the dataflow between the hardware functions and the processor application is a key element of this process.



Finally, while converting software functions into hardware functions, you should test early and test often by checking the results of the hardware function's algorithm. Validate the data returned by the hardware function with the expected results returned by the original software function. Restructuring code into hardware functions may yield better performance, but you will want to check it for equivalent results.

As for the remaining application functions, it is a matter of determining if the application will be running on Linux, FreeRTOS, or standalone (baremetal). Each type will have its own pros and cons; for example, standalone will be the easiest to use since only the Arm® will be running the application host, but using features that are only for Linux or FreeRTOS will not be allowed.

# **Memory Allocation**

Knowing what data is going to be processed by the accelerators can help you write the application code to better allocate the memory being used. Generally, malloc/free will work in the main function, but allocating memory specific to an accelerator using  $sds_alloc/sds_free$  will yield better performance due to the data being allocated and stored in physically contiguous memory that yields faster reads and writes to the memory. Although you can use malloc, or simply leave allocation to the compiler, you will experience performance problems by using a scatter-gather approach to memory access. It is strongly recommended that you allocate memory using  $sds_alloc$  for data going to the hardware functions.

The types of memory used are classified as contiguous/non-contiguous, and cacheable/non-cacheable. For contiguous memory, an array would have all the elements of the array allocated physically next to each other allowing for faster access times (think sequential read/writes). Using non-cacheable memory means that the data being transferred is not intended to be used by the PS, allowing for a higher transaction speeds. When using a cached memory allocation, there is a performance hit for flushing the cache, as well as CPU access latencies.

You must allocate the data before calling the hardware function. The runtime will set up data movers for you, with consideration for how memory is allocated. For example, in a matrix multiplication design that contains 1024 elements (32x32), you must explicitly allocate memory for the hardware function in the main function. The following code directs the compiler to allocate memory on the heap in a physically contiguous, cacheable fashion:

```
int MatA[1024] = (int*)sds_alloc(1024*sizeof(int));
int MatB[1024] = (int*)sds_alloc(1024*sizeof(int));
int MatC[1024] = (int*)sds_alloc(1024*sizeof(int));
```

Allocating the memory on the heap allows for a lot more data to be processed, and to be executed with better performance. Once execution of this code is complete, you can release the memory using sds\_free.



Examples of memory allocation can be found in the SDSoC Examples available on the Xilinx<sup>®</sup> GitHub repository.. The following code is from the mmultadd example available in the  $<install_dir>/SDx/<version>/samples$  folder. The code shows allocating the memory in the main function, and performing a quick check to make sure it was properly allocated, and releases the allocated memory if there was a problem:

```
int main(int argc, char* argv[]){
     int test_passed = 0;
    float *A, *B, *C, *D, *D_sw;
    A = (float *)sds_alloc(N * N * sizeof(float));
    B = (float *)sds_alloc(N * N * sizeof(float));
    C = (float *)sds_alloc(N * N * sizeof(float));
    D = (float *)sds_alloc(N * N * sizeof(float));
    D_sw = (float *)malloc(N * N * sizeof(float));
     if (!A || !B || !C || !D || !D_sw) {
          if (A) sds_free(A);
          if (B) sds_free(B);
          if (C) sds_free(C);
          if (D) sds_free(D);
          if (D_sw) free(D_sw);
          return 2;
     }
}
```

In the example above you can see that variables used by the hardware functions are allocated using the  $sds_alloc$  function to insure physically contiguous memory is allocated, while the software-only variable (D\_sw) is allocated using malloc.

At the end of the main function, all of the allocated memory is released using sds\_free or free as appropriate:

```
sds_free(A);
sds_free(B);
sds_free(C);
sds_free(D);
free(D_sw);
```

The  $sds_alloc$  function, and other SDSoC<sup>TM</sup> specific functions for memory allocation/deallocation can be found in  $sds_lib$ . h. More information on these APIs can be found in the Appendix B: SDSoC Environment API.



# Sequential/Parallel Accelerator Execution

After defining the memory allocations needed for the accelerators, you should determine how to call the accelerators from the application code. There are multiple ways for the accelerators to operate in the context of the main application. For example in an application in which there is only one accelerator, calling the hardware function like any other function will achieve the desired results of a sequential dataflow. However, for multiple accelerators, knowing whether and how the data is shared between the accelerators will let you choose between two distinct flows: sequential (synchronous) dataflow, or parallel (asynchronous) dataflow. In a sequential dataflow, the accelerators will operate in sequence: one will execute followed by the next, providing some benefit of acceleration in the hardware implementation. In parallel dataflow, both accelerators can operate concurrently, granting your application task-level parallelism for significant performance improvement. You can also use pipelining (pragma HLS PIPELINE), which allows both accelerators to operate concurrently on slightly skewed parts of the data set. See the SDx Pragma Reference Guide (UG1253) for more information on the pragmas discussed here.

To implement asynchronous dataflow, you must specify #pragma SDS async(id) and #pragma SDS wait(id) in your embedded processor application. You must place these pragmas in the application code, before and after the hardware function call, as shown in the following example:

```
#pragma SDS async(1)
mmult(A, B, C);
#pragma SDS async(2)
madd(D, E, F);

// Do other SW functions

#pragma SDS wait(1)
#pragma SDS wait(2)
```



**TIP:** The advantage of using async/wait is that it lets the application perform other operations while the hardware functions are running; and lets you hold the application at the appropriate point to wait for a hardware function to return.

The preceding code example demonstrates a typical asynchronous method. Here, the provided IDs correspond to their respective function (id = 1 for mmult, id = 2 for madd). The mmult function will be loaded with the inputs values of A and B, and processed. Notice in this case where the accelerators are data independent (data is not being shared between the accelerators), asynchronous execution is beneficial. If you determine that the data for an accelerator is not needed by other functions on either the CPU or another accelerator, then async execution with non-cacheable physically contiguous data may provide the best performance.





**IMPORTANT!**: In cases where the data from one accelerator is required by a second accelerator, you should not use <code>async/wait</code>. The <code>async</code> pragma forgoes compiler driven syncing, and thus you could end up with incorrect results if one accelerator requires syncing prior to the start of another.

An example of direct connection is provided with the SDSoC Examples available on the Xilinx® GitHub repository. The parallel\_accel code offers a simple example of two hardware functions, matrix addition and matrix multiplication, to demonstrate async and wait which helps to achieve greater performance through system parallelism and concurrency.

The  $parallel_accel$  example provides both a sequential dataflow form of the two accelerators, and a paralllel dataflow form of the two accelerators, and uses performance monitor functions ( $seq_hw_ctr$ ,  $par_hw_ctr$ ) from the included  $sds_utils$ . h to measure the performance difference. The relevant code is provided below for examination:

```
//Two hw functions are called back to back. First the
//vadd_accel is executed, then vmul_accel is executed.
//The execution of both accelerators is sequential here.
//To prevent automatic dataflow between calls to the two
//hw functions, async and wait pragma is used here so as
//to ensure that the two hw functions will be running sequentially.
seq_hw_ctr.start();
// Launch Hardware Solution
for(int itr = 0; itr < MAX_NUM_TIMES; itr++)</pre>
  #pragma SDS async(1)
 vadd_accel(source_in1, source_in2, source_vadd_hw_results, size);
 #pragma SDS wait(1)
 #pragma SDS async(2)
 vmul_accel(source_in1, source_in2, source_vmul_hw_results, size);
 #pragma SDS wait(2)
seq_hw_ctr.stop();
//Two hw functions are called back to back.
//The program running on the hardware first transfers in1 and in2
//to the vadd_accel hardware and returns immediately. Then the program
//transfers in1 and in2 to the vmul_accel hardware and returns
//immediately. When the program later executes to the point of
//#pragma SDS wait(id), it waits for the particular output to be ready.
par_hw_ctr.start();
// Launch Hardware Solution
#pragma SDS async(1)
vadd_accel(source_in1, source_in2, source_vadd_hw_results, size);
#pragma SDS async(2)
vmul_accel(source_in1, source_in2, source_vmul_hw_results, size);
for(int itr = 0; itr < MAX_NUM_TIMES; itr++)</pre>
 #pragma SDS wait(1)
 #pragma SDS async(1)
 vadd_accel(source_in1, source_in2, source_vadd_hw_results, size);
 #pragma SDS wait(2)
 #pragma SDS async(2)
 vmul_accel(source_in1, source_in2, source_vmul_hw_results, size);
#pragma SDS wait(1)
#pragma SDS wait(2)
par_hw_ctr.stop();
```



In the sequential dataflow example, the async and wait pragmas are used to insure that the two hardware functions are run sequentially. The key is the use of the wait pragma before the call to the multiplier function, vmul\_accel, which insures that the addition function, vadd\_accel, completes before matrix multiplication begins. Notice also the use of the async(2) and wait(2) pragmas to insure that the application waits for the completion of the vmul\_accel hardware function before proceeding.



**TIP:** The async/wait pragmas are not actually needed in the preceding example, as the compiler will automatically synchronize these functions in the manner described.

In the parallel dataflow example, the  $vadd_accel$  and  $vmul_accel$  functions are started in sequence, not waiting for one to complete before calling the next. This results in nearly parallel execution of the two hardware functions. These function calls are labeled async(1) and async(2). Then the for loop is called to repeat the functions a number of times (MAX\_NUM\_TIMES), but wait(1) and wait(2) are used to wait for the prior executions to complete before calling the functions again.

As with parallel code, you must explicitly synchronize the function calls so that the data is available for the application to complete the function. Failure to program this properly can result in deadlocks, or non-deterministic behavior. However, in some instances running concurrent accelerators may not provide the best performance compared to other means. An example of this is pipelining concurrent accelerators that are data dependent on each other. This would require the data to be synced on a pipeline stage before the accelerator can begin to process the data.

# Validating the Software to Hardware Conversion

Testing accelerators in the SDSoC™ environment is similar to testing any other function on a software platform. Generally, you can write a test bench to exercise and validate the application code, or this testing can be implemented as a function call from the main function with a golden dataset, and then comparing the outputs. Converting the C/C++ code of the software function to the HDL code of the hardware function may cause the behavior of the hardware function to change. It is a good idea to always run a verification test between the converted hardware code and the known good software code to make sure the algorithm is maintained through the complete build process.



**TIP:** For an application that has multiple accelerators, it is best to do a bottom-up testing approach, testing each accelerator individually, and then testing all accelerators together. This should shorten debug time. Refer to the SDSoC Environment Debugging Guide (UG1282) for more information.



Examples of verification code can be found in the SDSoC Examples available on the Xilinx® GitHub repository. The following code is from the mmultadd example available in the <install\_dir>/SDx/<version>/samples folder. The main.cpp file defines methods to calculate golden data for the matrix addition (madd\_golden) and multiplication (mmult\_golden). The code for mmult\_golden is provided below:

```
void mmult_golden(float *A, float *B, float *C)
{
    for (int row = 0; row < N; row++) {
        for (int col = 0; col < N; col++) {
            float result = 0.0;
            for (int k = 0; k < N; k++) {
                result += A[row*N+k] * B[k*N+col];
            }
            C[row*N+col] = result;
        }
}</pre>
```

Notice the function is essentially the same as the hardware function, mmult, which accelerates the matrix multiplication in the PL region of the device, while adding a few techniques such as array partitioning and pipelining to achieve optimal performance. The mmult\_golden simply calculates the expected value as golden data to be compared against the results returned by the accelerated function.

Finally, within the mmult\_test function, the verification process is called to generate the golden data and compare it to the results generated by the accelerated hardware functions. This section of the code is provided below:

```
int mmult_test(float *A,
                          float *B, float *C, float *D, float *D_sw)
    std::cout << "Testing " << NUM_TESTS << " iterations of " << N << "x" \,
        << N << " floating point mmultadd..." << std::endl;
    perf_counter hw_ctr, sw_ctr;
    for (int i = 0; i < NUM_TESTS; i++)</pre>
        init_arrays(A, B, C, D, D_sw);
        float tmp[N*N], tmp1[N*N];
        sw_ctr.start();
        mmult_golden(A, B, tmp);
        madd_golden(tmp, C, D_sw);
        sw_ctr.stop();
        hw_ctr.start();
        mmult(A, B, tmp1);
        madd(tmp1, C, D);
        hw_ctr.stop();
        if (result_check(D, D_sw))
```



```
return 1;
}
...
}
```

## **Performance Estimation**

In some cases, knowing the wall-clock time of the functions that may be turned into hardware functions may be necessary. You can accurately measure the execution time of functions by using special SDSoC API calls that measure activity based off of the free running clock of the Arm processor. The API functions include <code>sds\_clock\_counter()</code> and <code>sds\_clock\_frequency()</code>. These functions can be used to log the start and end times of a function. The function <code>sds\_clock\_counter()</code> returns the value of the free running clock register, while the function <code>sds\_clock\_frequency()</code> will return the speed in ticks/second of the Arm processor. Refer to Appendix B: SDSoC Environment API for more information on these functions.

**Note:** sds\_clock\_frequency() is a high performance counter and offers a fine-grained measurement of events.

A performance counter class is provided in the sds\_util.h available with the SDSoC Examples on the Xilinx® GitHub repository. The perf\_counter includes methods for capturing the start and stop clock times, and the number of function calls, as shown below:

```
#include "sds_lib.h"

class perf_counter
{
  public:
     uint64_t tot, cnt, calls;
     perf_counter() : tot(0), cnt(0), calls(0) {};
     inline void reset() { tot = cnt = calls = 0; }
     inline void start() { cnt = sds_clock_counter(); calls++; };
     inline void stop() { tot += (sds_clock_counter() - cnt); };
     inline uint64_t avg_cpu_cycles() { return ((tot+(calls>>1)) / calls); };
};
```

You can also use the  $avg_cpu_cycles()$  method to return the equivalent number of average cycles the task took in CPU cycle count.





# **Programming Hardware Functions**

Programming a function for hardware acceleration in the SDSoC<sup>m</sup> environment is as simple as writing a standard C/C++ function. However, to get the significant performance advantages of hardware acceleration through the sds++/sdscc (referred to as sds++) system compiler, there are a few considerations to keep in mind when writing the function, or modifying existing code to be implemented in programmable logic.

- Defining the function interface: the datatypes of inputs and outputs, and data transfers.
- What kind of memory access the function will have: DMA (interfacing with DDR), or FIFOs.
- How will the data be accessed: contiguous or non-contiguous.
- How will the data be processed: loops, arrays.

Determining what data is going to be processed in and out of an accelerator is the first step in creating a hardware function. Knowing the inputs and outputs of the hardware function, you can get an idea of what parallelism can be achieved. A critical element to writing a function for acceleration in programmable logic is that all data being used by the function can not change in size; arrays are static and cannot be resized. By being strict on the size of the array, the compiler can complete implementing the function into programmable logic, which cannot change during run-time.

#### **Exporting Hardware Functions as Libraries**

Once a hardware function, or a library of functions are written and optimized as needed, you can create an exported library for reuse in other projects. A general flow for exporting a library is to make sure that all the function definitions are grouped appropriately, and use the sds++/sdscc command with the -shared (which is interpreted as -fPIC for gcc) option to build a shared library when compiling the functions. More detailed information can be found in Appendix A: Exporting a Library for GCC.



#### **C-Callable IP**

An accelerator can also be created using RTL and provided as an IP core through the Vivado® Design Suite, called a C-Callable IP. In this case, to use the C-Callable IP in the application code, you will add the static library compiled for the appropriate Arm® Processor, typically an archive (. a) file, and the header file (. h/. hpp) as source files in the SDSoC application project. These files need be added to the source directory of the application or specified for library search (-L) and include search (-I), and properly included and linked with the main function. Now, like any other hardware function, it can be called as a typical C/C++ function, making sure the data sizes match to the function parameters. More information on creating and using C-Callable IP can be found in the SDSoC Environment User Guide (UG1027).

# **Coding Guidelines**

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

#### **General Hardware Function Guidelines**

- Hardware functions can execute concurrently under the control of a master thread. Multiple master threads are supported.
- A top-level hardware function must be a global function, not a class method, and it cannot be an overloaded function.
- There is no support for exception handling in hardware functions.
- It is an error to refer to a global variable within a hardware function or any of its sub-functions when this global variable is also referenced by other functions running in software.
- Hardware functions support scalar types up to 1024 bits, including double, long long, packed structs, etc.
- A hardware function must have at least one argument.
- An output or inout scalar argument to a hardware function can be assigned multiple times, but only the last written value will be read upon function exit.
- Use predefined macros to guard code with #ifdef and #ifndef preprocessor statements; the macro names begin and end with two underscore characters '\_'. For examples, see "SDSCC/SDS++ Compiler Commands" in the SDx Command and Utility Reference Guide (UG1279).



- The  $\_\_SDSCC\_\_$  macro is defined and passed as a -D option to sub-tools whenever sds++ is used to compile source files, and can be used to guard code dependent on whether it is compiled by sds++ or by another compiler, for example a GNU host compiler.
- When sds++ compiles source files targeted for hardware acceleration using Vivado HLS, the \_\_SDSVHLS\_\_ macro is defined and passed as a -D option, and can be used to guard code dependent on whether high-level synthesis is run or not.
- Vivado® HLS employs some 32-bit libraries irrespective of the host machine. Furthermore, the tool does not provide a true cross-compilation.

All object code for the Arm® CPUs is generated with the GNU toolchains, but the sds++ compiler, built upon Clang/LLVM frameworks, is generally less forgiving of C/C++ language violations than the GNU compilers. As a result, you might find that some libraries needed for your application cause front-end compiler errors when using sds++. In such cases, compile the source files directly through the GNU toolchain rather than through sds++, either in your makefiles or by setting the compiler to arm-linux-gnueabihf-g++ by right-clicking on the file (or folder) in the **Project Explorer** and selecting **C/C++ Build**  $\rightarrow$  **Settings**  $\rightarrow$  **SDSCC/SDS++ Compiler**. Refer to "Compiling and Running Applications on an Arm® Processor" in the *SDSoC Environment User Guide* (UG1027) for more information.

## **Hardware Function Argument Types**

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

```
#pragma SDS data copy|zero_copy
#pragma SDS data access_pattern
```

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

• Vivado HLS provides arbitrary precision types ap\_fixed<int>, ap\_int<int>, and an hls::stream class. In the SDSoC environment, ap\_fixed<int> types must be specified as having widths greater than 7 but less than 1025 (7 < width < 1025). The hls::stream data type is not supported as the function argument to any hardware function.



- By default, an array argument to a hardware function is transferred by copying the data, that is, it is equivalent to using #pragma SDS data copy. As a consequence, an array argument must be either used as an input or produced as an output, but not both. For an array that is both read and written by the hardware function, you must use #pragma SDS data zero\_copy to tell the compiler that the array should be kept in the shared memory and not copied.
- To ensure alignment across the hardware/software interface, do not use hardware function arguments that are an array of bool.

#### **Pointers**

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

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

The DATA ZERO\_COPY pragma provides pointer semantics using shared memory.

```
#pragma SDS data zero_copy
```

• The DATA COPY and DATA ACCESS\_PATTERN pragma pair maps the argument onto a stream, and requires that the array elements are accessed in sequential index order.

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

The DATA COPY pragma is only required when the sds++ system compiler is unable to determine the data transfer size and issues an error.



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

#### **Hardware Function Call Guidelines**

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

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





Where the <C\_size\_expr> must compile in the scope of the function declaration.

The zero\_copy pragma directs the SDSoC environment to map the argument into shared memory.



**IMPORTANT!:** Be aware that mismatches between intended and actual data transfer sizes can cause the system to hang at runtime, requiring laborious hardware debugging. Refer to the SDSoC Environment Debugging Guide (UG1282).

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

Note that in order to use sds\_alloc() from sds\_lib.h, it is necessary to include stdlib.h before including sds\_lib.h. stdlib.h is included to provide the size\_t type.

### **Data Movers**

A data mover is added by the sds++/sdscc compiler to move data into and out of the hardware accelerator. The data mover will be inferred by the compiler based on the volume of data being transferred, characteristics of memory being transferred, and access pattern expectations of the accelerator consuming or producing the data in the hardware function. The data mover is implemented into the PL region to support data transfers to and from the hardware function. The system compiler will implement one or more accelerators in the programmable region, including the data movers, automating control signals, and interrupts.

You can specify the SDS DATA COPY or DATA ZERO\_COPY pragmas in the source code of the hardware function to influence the behavior of the data movers. The DATA COPY pragma indicates that data is explicitly copied between memory and the hardware function, using a suitable data mover for the transfer. The DATA ZERO\_COPY pragma means that the hardware function accesses the data directly from shared memory through an AXI master bus interface.



With the data size boundaries known, methods of transferring data can be optimized. As an example, if the data is a large image (1920x1080) accessing the data would be easier by storing it in DDR in a physically contiguous way. Retrieving the data can be done through direct interfacing to the DDR and then determining if it needs to be sequentially accessed or not. By default, sds+ compiler will infer the data movers, but in cases where the data being transferred is large, you will need to identify the appropriate data mover to use. Refer to the SDSoC Environment Profiling and Optimization Guide (UG1235) for information on setting up and optimizing the data motion network.

To incorporate these types of data movers, pragmas are used to tell the compiler how to interface the accelerator to the rest of the system. For example, for storing and retrieving an image in DDR the following SDS pragmas can be used:

```
#pragma SDS data copy(data[0:size])
#pragma SDS data copy(out[0:size])
#pragma DATA ACCESS_PATTERN(data:SEQUENTIAL, out:SEQUENTIAL)
void accelerator(float *data, int *out, int size) { ... }
```



**IMPORTANT!**: Using the *COPY* pragma also requires the *ACCESS\_PATTERN* pragma to be used.

The DATA ACCESS\_PATTERN pragma is used to specify how the array data is being transferred. It is need when the data is determined at runtime, but of unbounded size at compile. For SEQUENTIAL data access, SDSoC will create a streaming interface like a FIFO, while a RANDOM data access pattern will create a RAM interface, which is the default. A key point is that SEQUENTIAL will only access elements from the array one time, while RANDOM can access any array value in any order. Note though that using a random access pattern will still transfer the complete volume of data, regardless of the volume being accessed.

You can also use the SDS pragma DATA MEM\_ATTRIBUTE as a hint to the compiler to trust that memory is physically contiguous or not, and is cacheable or not.

Knowing how the data is being allocated can help tune the system performance depending on the accelerator. For example, the compiler can use simple DMA transfer for physically contiguous memory, which is smaller and faster than AXI\_DMA\_SG. For physically contiguous memory, you must use  $sds_alloc$ , while for non-physically contiguous memory use malloc. Specifying the DATA MEM ATTRIBUTE helps determine what kind of data mover the compiler will use.

See the SDx Pragma Reference Guide (UG1253) for more information on SDS pragmas and examples.

You can also direct the creation of data movers as elements of a packaged C-Callable IP by applying pragmas to the software function signature defined in the header file associated with the C-Callable IP. Refer to the SDSoC Environment User Guide (UG1027) for more information.



# The Function Body

After determining the function interfaces and the data transfer mechanism, writing the function body is all that remains. The body of a hardware function should not be all that different from a function written for the processor. However, a key point to remember is that there are opportunities to improve the performance of the accelerator, and of the overall application. To this end, you can examine and rewrite the structure of hardware functions to increase instruction-level or task-level parallelism, use bit-accurate datatypes, manage loop unrolling and pipelining, and overall dataflow.

# **Datatypes**

As it is faster to write and verify the code by using native C datatypes such as int, float, or double, it is a common practice to use these datatypes when coding for the first time. However, the code is implemented in hardware, and all the operator sizes used in the hardware are dependent on the datatypes used in the accelerator code. The default native C/C++ datatypes can result in larger and slower hardware resources that can limit the performance of the . Instead, you should consider using bit-accurate datatypes to ensure the code is optimized for implementation in hardware. Using bit accurate, or arbitrary precision datatypes results in hardware operators which are smaller and faster. This allows more logic to be placed into the programmable logic, and also allows the logic to execute at higher clock frequencies, while using less power.



**RECOMMENDED:** Consider using bit-accurate datatypes instead of native C/C++ datatypes in your code.

# **Arbitrary Precision Integer Types**

Arbitrary precision integer datatypes are defined by <code>ap\_int</code> or <code>ap\_uint</code> for signed and unsigned integer respectively inside the header file <code>ap\_int</code>. h. To use arbitrary precision integer data type:

- Add header file ap\_int.h to the source code.
- Change the bit types to ap\_int<N> or ap\_uint<N>, where N is a bit-size from 1 to 1024.



The following example shows how the header file is added and the two variables are implemented to use 9-bit integer and 10-bit unsigned integer.

```
#include "ap_int.h"
ap_int<9> var1 // 9 bit signed integer
ap_uint<10> var2 // 10 bit unsigned integer
```

### **Arbitrary Precision Fixed-Point Data Types**

Some existing applications use floating point data types as they are written for other hardware architectures. However, fixed-point data types are a useful replacement for floating point types which require many clock cycles to complete. You should carefully evaluate trade-offs in power, cost, productivity, and precision when choosing to implement floating-point vs. fixed-point arithmetic for your application and accelerators.

As discussed in *Deep Learning with INT8 Optimization on Xilinx Devices* (WP486), using fixed point arithmetic instead of floating point for applications like machine learning can increase power efficiency, and lower the total power required. Unless the entire range of the floating-point type is required, the same accuracy can often be implemented with a fixed-point type resulting in the same accuracy with smaller and faster hardware. The paper *Reduce Power and Cost by Converting from Floating Point to Fixed Point* (WP491) provides some examples of this conversion.

Fixed-point data types model the data as an integer and fraction bits. The fixed point data type requires the ap\_fixed header, and supports both a signed and unsigned form as follows:

- Header file: ap\_fixed.h
- Signed fixed point: ap\_fixed<W,I,Q,O,N>
- Unsigned fixed point: ap\_ufixed<W,I,Q,O,N>
  - W = Total width <1024 bits</p>
  - I = Integer bit width. The value of I must be less than or equal to the width (W). The number of bits to represent the fractional part is W minus I. Only a constant integer expression can be used to specify the integer width.
  - Q = Quantization mode. Only predefined enumerated values can be used to specify Q. The accepted values are:
    - AP\_RND: Rounding to plus infinity.
    - AP RND ZERO: Rounding to zero.
    - AP RND MIN INF: Rounding to minus infinity.
    - AP RND INF: Rounding to infinity.
    - AP\_RND\_CONV: Convergent rounding.
    - AP\_TRN: Truncation. This is the default value when Q is not specified.





- AP\_TRN\_ZERO: Truncation to zero.
- O = Overflow mode. Only predefined enumerated values can be used to specify O. The accepted values are:
  - AP\_SAT: Saturation.
  - AP\_SAT\_ZERO: Saturation to zero.
  - AP\_SAT\_SYM: Symmetrical saturation.
  - AP\_WRAP: Wrap-around. This is the default value when O is not specified.
  - AP\_WRAP\_SM: Sign magnitude wrap-around.
- N= The number of saturation bits in the overflow WRAP modes. Only a constant integer expression can be used as the parameter value. The default value is zero.



**TIP:** The  $ap\_fixed$  and  $ap\_ufixed$  datatypes permit shorthand definition, with only W and I being required, and other parameters assigned default values. However, to define Q or N, you must also specify the parameters before those, even if you just specify the default values.

In the example code below, the ap\_fixed type is used to define a signed 18-bit variable with 6 bits representing the integer value above the binary point, and by implication, 12-bits representing the fractional value below the binary point. The quantization mode is set to round to plus infinity (AP\_RND). Since the overflow mode and saturation bits are not specified, the defaults AP\_WRAP and 0 are used.

```
#include <ap_fixed.h>
...
ap_fixed<18,6,AP_RND> my_type;
...
```

When performing calculations where the variables have different numbers of bits (W), or different precision (I), the binary point is automatically aligned. Refer to "C++ Arbitrary Precision Fixed-Point Types" in the *Vivado Design Suite User Guide: High-Level Synthesis* (UG902) for more information on using fixed-point datatypes.

# **Array Configuration**

The compiler maps large arrays to the memory (known as block RAM or BRAM) on the programmable region. These BRAM can have a maximum of two access points or ports. This can limit the performance of the application as all the elements of an array cannot be accessed in parallel when implemented in hardware.





Depending on the performance requirements, you may need to access some or all of the elements of an array in the same clock cycle. To achieve this, the #pragma HLS ARRAY\_PARTITION can be used to instruct the compiler to split the elements of an array and map it to smaller arrays, or to individual registers. The compiler provides three types of array partitioning, as shown in the following figure. The three types of partitioning are:

- block: The original array is split into equally sized blocks of consecutive elements of the original array.
- cyclic: The original array is split into equally sized blocks interleaving the elements of the original array.
- complete: Split the array into its individual elements. This corresponds to resolving a memory into individual registers. This is the default for the ARRAY\_PARTITION pragma.

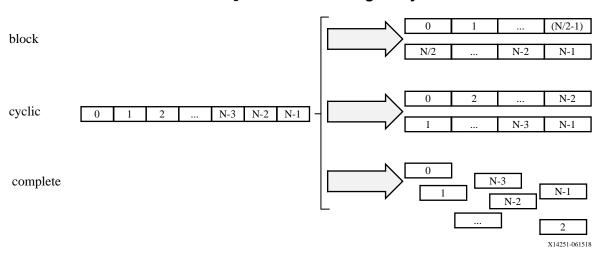


Figure 3: Partitioning Arrays

For block and cyclic partitioning, the factor option specifies the number of arrays that are created. In the preceding figure, a factor of 2 is used to split the array into two smaller arrays. If the number of elements in the array is not an integer multiple of the factor, the later arrays will have fewer elements.

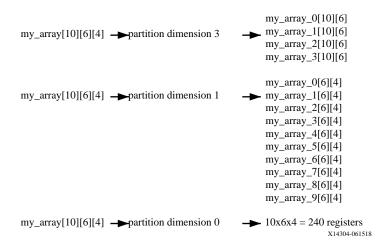
When partitioning multi-dimensional arrays, the dimension option is used to specify which dimension is partitioned. The following figure shows how the dimension option is used to partition the following example code in three different ways

```
void foo (...) {
   // my_array[dim=1][dim=2][dim=3]
   // The following three pragma results are shown in the figure below
   // #pragma HLS ARRAY_PARTITION variable=my_array dim=3 <block|cyclic>
factor=2
   // #pragma HLS ARRAY_PARTITION variable=my_array dim=1 <block|cyclic>
```



```
factor=2
  // #pragma HLS ARRAY_PARTITION variable=my_array dim=0 complete
  int my_array[10][6][4];
  ...
}
```

Figure 4: Partitioning the Dimensions of an Array



The examples in the figure demonstrate how partitioning dimension 3 results in 4 separate arrays and partitioning dimension 1 results in 10 separate arrays. If 0 is specified as the dimension, all dimensions are partitioned.

#### The Importance of Careful Partitioning

A complete partition of the array maps all the array elements to the individual registers. This helps in improving the kernel performance because all of these registers can be accessed concurrently in a same cycle. However, a word of caution, complete partitioning of the large arrays consumes a lot of FPGA resources. It may even cause the compilation process to slow down and face capacity issue. You should partition the array only when it is needed. Consider selectively consider partitioning a particular dimension, or doing block or cycle partitioning. The following is an example of array partitioning using a common matrix multiplication algorithm.

#### **Choosing a Specific Dimension to Partition**

Suppose A and B are two-dimensional arrays representing two matrices. Consider the Matrix Multiplication algorithm below:

```
int A[64][64];
int B[64][64];

ROW_WISE: for (int i = 0; i < 64; i++) {
   COL_WISE : for (int j = 0; j < 64; j++) {
    #pragma HLS PIPELINE
   int result = 0;
   COMPUTE_LOOP: for (int k = 0; k < 64; k++) {</pre>
```





```
result += A[i ][ k] * B[k ][ j];
}
C[i][ j] = result;
}
```

Due to the PIPELINE pragma, the ROW\_WISE and COL\_WISE loop will be flattened together and COMPUTE\_LOOP will get fully unrolled. In order to concurrently execute each iteration (k) of the COMPUTE\_LOOP, the code must access each column of matrix A and each row of matrix B in parallel. Therefore, the matrix A should be split in the second dimension, and matrix B should be split in the first dimension.

```
#pragma HLS ARRAY_PARTITION variable=A dim=2 complete
#pragma HLS ARRAY_PARTITION variable=B dim=1 complete
```

#### **Choosing Between Cyclic and Block Partitions**

Here the same matrix multiplication algorithm is used to demonstrate choosing between cyclic and block partitioning, and determining the appropriate factor, by understanding the array access pattern of the underlying algorithm.

```
int A[64 * 64];
int B[64 * 64];
#pragma HLS ARRAY_PARTITION variable=A dim=1 cyclic factor=64
#pragma HLS ARRAY_PARTITION variable=B dim=1 block factor=64

ROW_WISE: for (int i = 0; i < 64; i++) {
    COL_WISE : for (int j = 0; j < 64; j++) {
        #pragma HLS PIPELINE
        int result = 0;
    COMPUTE_LOOP: for (int k = 0; k < 64; k++) {
        result += A[i * 64 + k] * B[k * 64 + j];
    }
    C[i* 64 + j] = result;
}</pre>
```

In this version of the code A and B are now one-dimensional arrays. In order to access each column of matrix A, and each row of matrix B in parallel, cyclic and block partitions are used as shown in the above example. To access each column of matrix A in parallel, cyclic partitioning is applied with the factor specified as the row size, in this case 64. Similarly, to access each row of matrix B in parallel, block partitioning is applied with the factor specified as the column size, or 64.

#### **Minimizing Array Accesses with Caching**

As arrays are mapped to BRAM with limited number of access ports, repeated array accesses can limit the performance of the accelerator. You should have a good understanding of the array access pattern of the algorithm, and limit the array accesses by locally caching the data to improve the performance of the .





The following code example shows a case in which accesses to an array can limit performance in the final implementation. In this example, there are three accesses to the array mem[N] to create a summed result.

```
#include "array_mem_bottleneck.h"
dout_t array_mem_bottleneck(din_t mem[N]) {
  dout_t sum=0;
  int i;
  SUM_LOOP:for(i=2;i<N;++i)
    sum += mem[i] + mem[i-1] + mem[i-2];
  return sum;
}</pre>
```

The code in the preceding example can be rewritten as shown in the following example to allow the code to be pipelined with a II=1. By performing pre-reads and manually pipelining the data accesses, there is only one array read specified inside each iteration of the loop. This ensures that only a single-port BRAM is needed to achieve the performance.

```
#include "array_mem_perform.h"
dout_t array_mem_perform(din_t mem[N]) {
    din_t tmp0, tmp1, tmp2;
    dout_t sum=0;
    int i;
    tmp0 = mem[0];
    tmp1 = mem[1];
    SUM_LOOP:for (i = 2; i < N; i++) {
        tmp2 = mem[i];
        sum += tmp2 + tmp1 + tmp0;
        tmp0 = tmp1;
        tmp1 = tmp2;
    }
    return sum;
}</pre>
```



**RECOMMENDED:** Consider minimizing the array access by caching to local registers to improve the pipelining performance depending on the algorithm.

# Loops

Loops are an important aspect for a high performance accelerator. Generally, loops are either pipelined or unrolled to take advantage of the highly distributed and parallel FPGA architecture to provide a performance boost compared to running on a CPU.

By default, loops are neither pipelined nor unrolled. Each iteration of the loop takes at least one clock cycle to execute in hardware. Thinking from the hardware perspective, there is an implicit "wait until clock" for the loop body. The next iteration of a loop only starts when the previous iteration is finished.



## **Loop Pipelining**

By default, every iteration of a loop only starts when the previous iteration has finished. In the loop example below, a single iteration of the loop adds two variables, and stores the result in a third variable. Assume that in hardware this loop takes 3 cycles to finish one iteration. Also, assume that the loop variable len is 20, i.e., the vadd loop runs for 20 iterations in the . Therefore, it requires a total of 60 clock cycles (20 iterations \* 3 cycles) to complete all the operations of this loop.

```
vadd: for(int i = 0; i < len; i++) {
  c[i] = a[i] + b[i];
}</pre>
```



**TIP:** It is good practice to always label a loop as shown in the above code example (vadd:...). This practice helps with debugging when working in the environment. Do note that the labels will generate warnings during compilation, which can be safely ignored.

Pipelining the loop executes subsequent iterations in a pipelined manner. This means that subsequent iterations of the loop overlap and run concurrently, executing at different sections of the loop-body. Pipelining a loop can be enabled by the pragma HLS PIPELINE. Note that the pragma is placed inside the body of the loop.

```
vadd: for(int i = 0; i < len; i++) {
    #pragma HLS PIPELINE
    c[i] = a[i] + b[i];
}</pre>
```

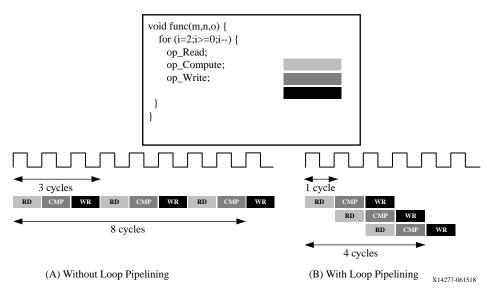
In the example above, it is assumed that every iteration of the loop takes three cycles: read, add, write. Without pipelining, each successive iteration of the loop starts in every third cycle. With pipelining the loop can start subsequent iterations of the loop in fewer than three cycles, such as in every second cycle, or in every cycle.

The number of cycles it takes to start the next iteration of a loop is called the initiation interval (II) of the pipelined loop. So II=2 means each successive iteration of the loop starts every two cycles. An II=1 is the ideal case, where each iteration of the loop starts in the very next cycle. When you use the pragma HLS PIPELINE the compiler always tries to achieve II=1 performance.

The following figure illustrates the difference in execution between pipelined and non-pipelined loops. In this figure, (A) shows the default sequential operation where there are 3 clock cycles between each input read (II=3), and it requires 8 clock cycles before the last output write is performed.







In the pipelined version of the loop shown in (B), a new input sample is read every cycle (II=1) and the final output is written after only 4 clock cycles: substantially improving both the II and latency while using the same hardware resources.



**IMPORTANT!:** Pipelining a loop causes any loops nested inside the pipelined loop to get unrolled

If there are data dependencies inside a loop it may not be possible to achieve II=1, and a larger initiation interval may be the result. Loop dependencies are discussed in Loop Dependencies.

#### **Loop Unrolling**

The compiler can also unroll a loop, either partially or completely to perform multiple loop iterations in parallel. This is done using the <code>HLS UNROLL</code> pragma. Unrolling a loop can lead to a very fast design, with significant parallelism. However, because all the operations of the loop iterations will be executed in parallel, a large amount of programmable logic resource will be required to implement the hardware. As a result, the compiler can face challenges dealing with such a large number of resources and can face capacity problems that slow down the compilation process. It is a good guideline to unroll loops that have a small loop body, or a small number of iterations.

```
vadd: for(int i = 0; i < 20; i++) {
    #pragma HLS UNROLL
    c[i] = a[i] + b[i];
}</pre>
```



In the preceding example, you can see pragma HLS UNROLL has been inserted into the body of the loop to instruct the compiler to unroll the loop completely. In other words, all 20 iterations of the loop will be executed in parallel if that is permitted by any data dependency.

Completely unrolling a loop can consume significant device resources, while partially unrolling the loop provides some performance improvement without causing a significant impact on hardware resources.

#### **Partially Unrolled loop**

In order to completely unroll a loop, the loop must have a constant bound (20 in the example above). However, partial unrolling is possible for loops with a variable bound. A partially unrolled loop means that only a certain number of loop iterations can be executed in parallel.

The following code examples illustrates how partially unrolled loops work:

```
array_sum:for(int i=0;i<4;i++){
    #pragma HLS UNROLL factor=2
    sum += arr[i];
}</pre>
```

In the above example the UNROLL pragma is given a factor of 2. This is the equivalent of manually duplicating the loop body and running the two loops concurrently for half as many iterations. The code below shows how this would be written. This transformation allows two iterations of the above loop to execute in parallel.

```
array_sum_unrolled:for(int i=0;i<2;i+=2){
  // Manual unroll by a factor 2
  sum += arr[i];
  sum += arr[i+1];
}</pre>
```

Just like data dependencies inside a loop impact the initiation interval of a pipelined loop, an unrolled loop will perform operations in parallel only if data dependencies allow it. If operations in one iteration of the loop require the result from a previous iteration, they cannot execute in parallel, but will execute as soon as the data from one iteration is available to the next.



**RECOMMENDED:** A good methodology is to PIPELINE loops first, and then UNROLL loops with small loop bodies and limited iterations to improve performance further.



#### **Loop Dependencies**

Data dependencies in loops can impact the results of loop pipelining or unrolling. These loop dependencies can be within a single iteration of a loop, or between different iterations of a loop. The easiest way to understand loop dependencies is to examine an extreme example. In the following code example, the result of the loop is used as the loop continuation or exit condition. Each iteration of the loop must finish before the next can start.

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

This loop cannot be pipelined. The next iteration of the loop cannot begin until the previous iteration ends.

Dealing with various types of dependencies with the compiler is an extensive topic requiring a detailed understanding of the high-level synthesis procedures underlying the compiler. Refer to the *Vivado Design Suite User Guide*: *High-Level Synthesis* (UG902) for more information on "Dependencies with Vivado HLS."

#### **Nested Loops**

Coding with nested loops is a common practice. Understanding how loops are pipelined in a nested loop structure is key to achieving the desired performance.

If the pragma HLS PIPELINE is applied to a loop nested inside another loop, the compiler attempts to flatten the loops to create a single loop, and apply the PIPELINE pragma to the constructed loop. The loop flattening helps in improving the performance of the .

The compiler is able to flatten the following types of nested loops:

- 1. Perfect nested loop:
  - Only the inner loop has a loop body.
  - There is no logic or operations specified between the loop declarations.
  - All the loop bounds are constant.
- 2. Semi-perfect nested loop:
  - Only the inner loop has a loop body.
  - There is no logic or operations specified between the loop declarations.
  - The inner loop bound must constant, but the outer loop bound can be variable.





The following code example illustrates the structure of a perfect nested loop:

```
ROW_LOOP: for(int i=0; i< MAX_HEIGHT; i++) {
   COL_LOOP: For(int j=0; j< MAX_WIDTH; j++) {
     #pragma HLS PIPELINE
     // Main computation per pixel
   }
}</pre>
```

The above example shows a nested loop structure with two loops that performs some computation on incoming pixel data. In most cases, you will want to process a pixel in every cycle, hence PIPELINE is applied to the nested loop body structure. The compiler will be able to flatten the nested loop structure in the example because it is a perfect nested loop.

The nested loop in the preceding example contains no logic between the two loop declarations. No logic is placed between the ROW\_LOOP and COL\_LOOP; all of the processing logic is inside the COL\_LOOP. Also, both the loops have a fixed number of iterations. These two criteria help the compiler flatten the loops and apply the PIPELINE constraint.



**RECOMMENDED:** If the outer loop has a variable boundary, then the compiler can still flatten the loop. You should always try to have a constant boundary for the inner loop.

#### **Sequential Loops**

If there are multiple loops in the design, by default they do not overlap, and will execute sequentially. This section introduces the concept of dataflow optimization for sequential loops. Consider the following code example:

```
void adder(unsigned int *in, unsigned int *out, int inc, int size) {
  unsigned int in_internal[MAX_SIZE];
  unsigned int out_internal[MAX_SIZE];
  mem_rd: for (int i = 0 ; i < size ; i++){
    #pragma HLS PIPELINE
    // Reading from the input vector "in" and saving to internal variable in_internal[i] = in[i];
  }
  compute: for (int i=0; i<size; i++) {
    #pragma HLS PIPELINE
    out_internal[i] = in_internal[i] + inc;
  }
  mem_wr: for(int i=0; i<size; i++) {
    #pragma HLS PIPELINE
    out[i] = out_internal[i];
  }
}</pre>
```

In the above example, three sequential loops are shown: mem\_rd, compute, and mem\_wr.



- The mem\_rd loop reads input vector data from the memory interface and stores it in internal storage.
- The main compute loop reads from the internal storage and performs an increment operation and saves the result to another internal storage.
- The mem\_wr loop writes the data back to memory from the internal storage.

By default, these loops are executed sequentially without any overlap. First the mem\_rd loop will finish reading all the input data before the compute loop will start its operation. Similarly the compute loop will finish processing the data before the mem\_wr loop will start to write the data. However, the execution of these loops can be overlapped, allowing the compute (or mem\_wr) loop to start as soon as there is enough data available to feed its operation, before the mem\_rd (or compute) loop has finished processing its data.

The loop execution can be overlapped using dataflow optimization as described in the Dataflow Optimization section.

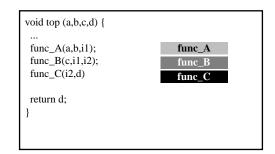
### **Dataflow Optimization**

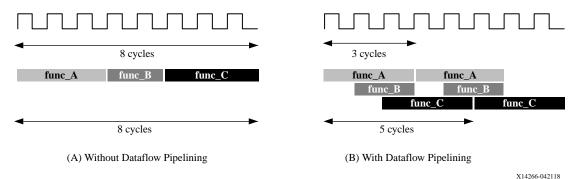
Dataflow optimization is a powerful technique to improve the performance by enabling task level pipelining and parallelism. It allows the compiler to schedule multiple functions to run concurrently to achieve higher throughput and lower latency. This is also known as task-level parallelism.

The following figure shows a conceptual view of dataflow pipelining. The default behavior is to execute and complete <code>func\_A</code>, then <code>func\_B</code>, and finally <code>func\_C</code>. With the <code>HLS DATAFLOW</code> pragma enabled, the compiler can schedule each function to execute as soon as data is available. In this example, the original <code>top</code> function has a latency and interval of eight clock cycles. With <code>DATAFLOW</code> optimization, the interval is reduced to only three clock cycles.



Figure 6: Dataflow Optimization





**Dataflow Coding example** 

In the following dataflow coding example you should notice the following:

- 1. The HLS DATAFLOW pragma is applied to instruct the compiler to enable dataflow optimization.
- 2. The stream class is used as a data transferring channel between each of the functions in the dataflow region.



**TIP:** The stream class infers a first-in first-out (FIFO) memory circuit in the programmable logic. This memory circuit, which acts as a queue in software programming, provides data-level synchronization between the functions and achieves better performance. For additional details on the hls:stream class, refer to the Vivado Design Suite User Guide: High-Level Synthesis (UG902).

```
void compute_kernel(ap_int<256> *inx, ap_int<256> *outx, DTYPE alpha) {
  hls::stream<unsigned int>inFifo;
  #pragma HLS STREAM variable=inFifo depth=32
  hls::stream<unsigned int>outFifo
  #pragma HLS STREAM variable=inFifo depth=32

  #pragma HLS DATAFLOW
  read_data(inx, inFifo);
  // Do computation with the acquired data
  compute(inFifo, outFifo, alpha);
  write_data(outx, outFifo);
  return;
}
```



#### **Canonical Forms of Dataflow Optimization**

Xilinx recommends writing the code inside a dataflow region using canonical forms. There are canonical forms for dataflow optimizations for both functions and loops.

- Functions: The canonical form coding guideline for dataflow inside a function specifies:
  - 1. Use only the following types of variables inside the dataflow region
    - a. Local non-static scalar/array/pointer variables.
    - b. Local static hls::stream variables.
  - 2. Function calls transfer data only in the forward direction.
  - 3. Array or hls::stream should have only one producer function and one consumer function.
  - 4. The function arguments (variables coming from outside the dataflow region) should only be read, or written, not both. If performing both read and write on the same function argument then read should happen before write.
  - 5. The local variables (those that are transferring data in forward direction) should be written before being read.

The following code example illustrates the canonical form for dataflow within a function. Note that the first function (func1) reads the inputs and the last function (func3) writes the outputs. Also note that one function creates output values that are passed to the next function as input parameters.

```
void dataflow(Input0, Input1, Output0, Output1) {
   UserDataType C0, C1, C2;
   #pragma HLS DATAFLOW
   func1(read Input0, read Input1, write C0, write C1);
   func2(read C0, read C1, write C2);
   func3(read C2, write Output0, write Output1);
}
```

- Loop: The canonical form coding guideline for dataflow inside a loop body includes the coding guidelines for a function defined above, and also specifies the following:
  - 1. Initial value 0.
  - 2. The loop condition is formed by a comparison of the loop variable with a numerical constant or variable that does not vary inside the loop body.
  - 3. Increment by 1.



The following code example illustrates the canonical form for dataflow within a loop.

```
void dataflow(Input0, Input1, Output0, Output1) {
    UserDataType C0, C1, C2;
    for (int i = 0; i < N; ++i) {
        #pragma HLS DATAFLOW
        func1(read Input0, read Input1, write C0, write C1);
        func2(read C0, read C0, read C1, write C2);
        func3(read C2, write Output0, write Output1);
    }
}</pre>
```

#### Troubleshooting Dataflow

The following behaviors can prevent the compiler from performing DATAFLOW optimizations:

- 1. Single producer-consumer violations.
- 2. Bypassing tasks.
- 3. Feedback between tasks.
- 4. Conditional execution of tasks.
- 5. Loops with multiple exit conditions or conditions defined within the loop.

If any of the above conditions occur inside the dataflow region, you may need to re-architect the code to successfully achieve dataflow optimization.





## Using External I/O

Hardware accelerators generated in the SDSoC<sup>™</sup> environment can communicate with system inputs and outputs either directly through hardware connections, or though memory buffers (e.g., a frame buffer). Examples of system I/O include analog-to-digital and digital-to-analog converters, image, radar, LiDAR, and ultrasonic sensors, and HDMI<sup>™</sup> multimedia streams.

A platform exports stream connections in hardware that can be accessed in software by calling platform library functions as described in Accessing External I/O using Memory Buffers and Accessing External I/O using Direct Hardware Connections. Direct hardware connections are implemented over AXI4-Stream channels, and connections to memory buffers are realized through function calls implemented by the standard data movers supported in the SDSoC Environment.

For information and examples that show how to create SDSoC platforms, refer to the SDSoC Environment Platform Development Guide (UG1146).

# Accessing External I/O using Memory Buffers

This section uses the motion-detect ZC702 + HDMI IO FMC or the ZC706 + HDMI IO FMC platform found under the "Boards, Kits, & Modules" page of the SDSoC Development Environment. The preconfigured SDSoC™ platform is responsible for the HDMI data transfer to external memory. The application must call the platform interfaces to process the data from the frame buffer in DDR memory. The following figure shows an example of how the design is configured.



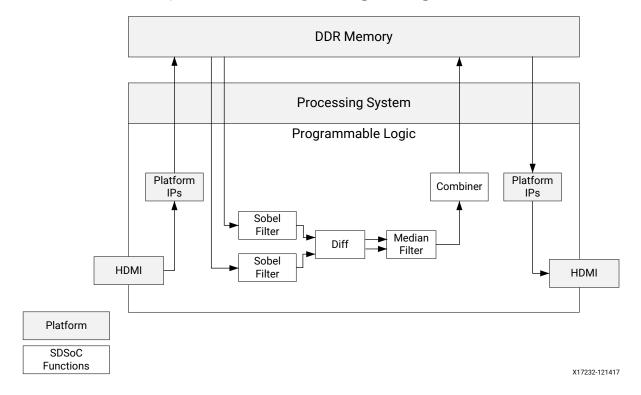


Figure 7: Motion Detect Design Configuration

The SDSoC Environment accesses the external frame buffer through an accelerator interface to the platform. The  $zc702\_hdmi$  platform provides a software interface to access the video frame buffer through the Video4Linux2 (V4L2) API. The V4L2 framework provides an API accessing a collection of device drivers supporting real-time video capture in Linux. This API is the application development platform I/O entry point.

The motion\_demo\_processing function in the following code snippet from m2m\_sw\_pipeline.c demonstrates the function call interface.



The application accesses this API in motion\_detect.c, where motion\_demo\_procesing is defined and called by the img\_process function.

Finally, img\_process calls the various filters and transforms to process the data.

By using a platform API to access the frame buffers, the application developer does not program at the driver level to process the video frames.

You can find the platform used for the code snippets on the SDSoC Downloads Page with the name ZC702 + HDMI IO FMC or ZC706 + HDMI IO FMC. To access the project in the SDSoC environment use the following steps:

- 1. Download and extract the platform to your system.
- 2. Open SDx<sup>™</sup> and create a new application project.
- 3. From the Platform dialog box, select **Add Custom Platform**.
- 4. From the **Specify Custom Platform Location** dialog box, select the location of the downloaded platform, and click **OK**.
- 5. From the Platform dialog box, select the custom platform named **zc702\_trd** or **zc706\_trd**, and click **Next**.
- 6. From the System Configuration dialog box, leave the default settings and click **Next**.
- 7. From the Templates dialog box, select the Dense Optical Flow (1PPC) template, and click Finish.



# Accessing External I/O using Direct Hardware Connections

Whereas the example in Accessing External I/O using Memory Buffers demonstrated how applications can access system I/O through memory buffers, a platform can also provide direct hardware connectivity to hardware accelerators within an application generated in the SDSoC™ Environment. The following figure shows how a function s2mm\_data\_copy communicates to the platform using an AXI4-Stream channel, and writes to DDR memory using the zero\_copy datamover (implemented as an AXI4 master interface). This design template is called aximm design example in the samples/platforms/zc702\_axis\_io platform.

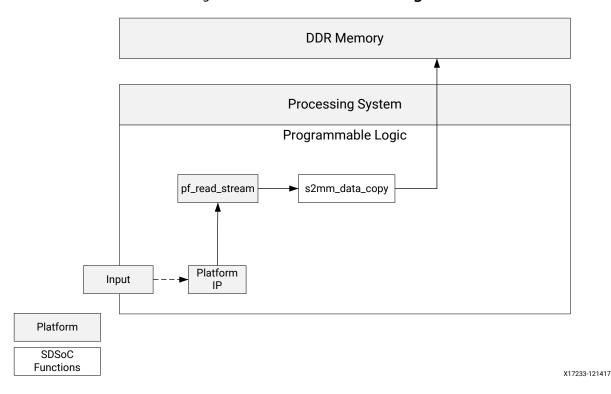


Figure 8: AXI4 Data Mover Design

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



The direct I/O interface is specified using the  $sys_port$  pragma for a stream port. In the code snippet below, the direct connection is specified for  $stream_fifo_M_AXIS$ .

```
#pragma SDS data sys_port (fifo:stream_fifo_M_AXIS)
#pragma SDS data zero_copy(buf)
int s2mm_data_copy(unsigned *fifo, unsigned buf[BUF_SIZE])
{
#pragma HLS interface axis port=fifo
    for(int i=0; i<BUF_SIZE; i++) {
#pragma HLS pipeline
        buf[i] = *fifo;
    }
    return 0;
}</pre>
```

In the main application code below, the rbuf0 variable maps the fifo input stream to the s2mm\_data\_copy function, so the sds++ compiler creates a direct hardware connection over an AXI4-Stream channel. Because the s2mm\_data\_copy function transfers buf using the zero\_copy data mover, the buffer must be allocated in physically contiguous memory using sds\_alloc, and released using sds\_free, as shown below.

```
int main()
     unsigned *bufs[NUM_BUFFERS];
     bool error = false;
     unsigned* rbuf0;
     for(int i=0; i<NUM_BUFFERS; i++) {</pre>
           bufs[i] = (unsigned*) sds_alloc(BUF_SIZE * sizeof(unsigned));
     // Flush the platform FIFO of start-up garbage
     s2mm_data_copy(rbuf0, bufs[0]);
     s2mm_data_copy(rbuf0, bufs[0]);
s2mm_data_copy(rbuf0, bufs[0]);
     for(int i=0; i<NUM_BUFFERS; i++) {</pre>
       s2mm_data_copy(rbuf0, bufs[i]);
     error = check(bufs);
     printf("TEST %s\n\r", (error ? "FAILED" : "PASSED"));
     for(int i=0; i<NUM_BUFFERS; i++) {</pre>
          sds_free(bufs[i]);
     return 0;
}
```

Information on creating a platform using AXI4-Stream to write to memory directly can be found in the "SDSoC Platform Examples" appendix of SDSoC Environment Platform Development Guide (UG1146).





## **Exporting a Library for GCC**

This chapter demonstrates how to use the sds++ compiler to build a library of software functions with entry points into hardware functions implemented in programmable logic. This library can be linked into applications using the standard GCC linker for Zynq®-7000 SoC and Zynq UltraScale+ MPSoC devices.

In addition to the library, sds++ generates a complete boot image for the hardware functions and data motion network, including an FPGA bitstream. Using this library, you can develop software applications that call the hardware functions (and fixed hardware) using standard GCC toolchains. You are still targeting the same hardware system and using the sds++-generated boot environment, but you are free to develop your software application using the GNU toolchain in the software development environment of your choice.



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

## **Building a Shared Library**



**TIP:** Shared libraries can be only created for application projects targeting Linux OS.

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

- mmult\_accel.cpp Accelerator code for the matrix multiplier
- mmult\_accel.h Header file for the matrix multiplier
- madd\_accel.cpp Accelerator code for the matrix adder
- madd\_accel.h Header file for the matrix adder
- matrix.cpp Code that calls the accelerators and determines the data motion network
- matrix.h Header file for the library



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

```
/* matrix.cpp */
#include "madd_accel.h"
#include "mmult_accel.h"
void madd(float in_A[MSIZE*MSIZE], float in_B[MSIZE*MSIZE], float
out_C[MSIZE*MSIZE])
 madd_accel(in_A, in_B, out_C);
}
void mmult(float in_A[MSIZE*MSIZE], float in_B[MSIZE*MSIZE], float
out_C[MSIZE*MSIZE])
 mmult_accel(in_A, in_B, out_C);
}
void mmultadd(float in_A[MSIZE*MSIZE], float in_B[MSIZE*MSIZE], float
in_C[MSIZE*MSIZE],
float out_D[MSIZE*MSIZE])
  float tmp[MSIZE * MSIZE];
 mmult_accel(in_A, in_B, tmp);
 madd_accel(tmp, in_C, out_D);
```

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

```
/* matrix.h */
#ifndef MATRIX_H_
#define MATRIX_H_
#define MSIZE 16

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

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

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

#endif /* MATRIX_H_ */
```



In the samples/libmatrix/build/shared folder, the Makefile shows how the project is built by specifying that the functions mmult\_accel, madd, and mmult\_add must be implemented in programmable logic.

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

Object files are generated with position independent code (PIC), like standard shared libraries, using the GCC standard -fpic option:

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

Link the objects files by using the using the GCC standard -shared switch:

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

After building the project, the following files will be generated:

- libmatrix.so Shared library suitable for linking using GCC and for runtime use
- sd\_card Directory containing an SD card image for booting the board

#### **Delivering a Library**

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

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



**IMPORTANT!:** The  $sd\_card$  folder is to be copied into an SD card and used to boot the board. This image includes a copy of the libmatrix.so file that is used at runtime.



## **Compiling and Linking Against a Library**

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

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

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

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



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

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

To link against the library, the linker needs the library. The path to the library is specified using the -L switch. Also, ask the linker to link against the library using the -1 switch.

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

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

#### Use a library at runtime

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





For example, if the sd\_card is mounted at /mnt, use this command:

export LD\_LIBRARY\_PATH=/mnt

## **Exporting a Shared Library**

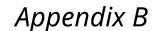
The following steps demonstrate how to export an SDSoC environment shared library with the corresponding SD card boot image using the SDSoC environment GUI.

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

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

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

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





## SDSoC Environment API

This section describes functions in  $sds_lib$  available for applications developed in the SDSoC Environment.

**Note:** To use the library, #include "sds\_lib.h" in source files. You must include stdlib.h before including sds\_lib.h to provide the size\_t type declaration.

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

- void sds\_wait(unsigned int id): Wait for the first accelerator in the queue identified by id, to complete. The recommended alternative is the use #pragma SDS wait(id), as described in pragma SDS async in the SDx Pragma Reference Guide (UG1253).
- void \*sds\_alloc(size\_t size): Allocate a physically contiguous array of size bytes.
- void \*sds\_alloc\_non\_cacheable(size\_t size):

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

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

- void sds\_free(void \*memptr): Free an array allocated through sds\_alloc()
- void \*sds\_mmap(void \*physical\_addr, size\_t size, void \*virtual\_addr): Create a virtual address mapping to access a memory of size bytes located at physical address physical\_addr.
  - physical\_addr: physical address to be mapped.
  - size: size of physical address to be mapped.
  - virtual\_addr:
    - of If not null, it is considered to be the virtual-address already mapped to the physical\_addr, and sds\_mmap keeps track of the mapping.
    - o If null, sds\_mmap invokes mmap() to generate the virtual address, and virtual\_addr is assigned this value.



- void \*sds\_munmap(void \*virtual\_addr): Unmaps a virtual address associated with a physical address created using sds\_mmap().
- unsigned long long sds\_clock\_counter(void): Returns the value associated with a free-running counter used for fine grain time interval measurements.
- unsigned long long sds\_clock\_frequency(void): Returns the frequency (in ticks/second) associated with the free-running counter that is read by calls to sds\_clock\_counter. This is used to translate counter ticks to seconds.





# Additional Resources and Legal Notices

#### Xilinx Resources

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

## Documentation Navigator and Design Hubs

Xilinx<sup>®</sup> 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**

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



## **Please Read: Important Legal Notices**

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

#### **AUTOMOTIVE APPLICATIONS DISCLAIMER**

AUTOMOTIVE PRODUCTS (IDENTIFIED AS "XA" IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE ("SAFETY APPLICATION") UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD ("SAFETY DESIGN"). CUSTOMER SHALL, PRIOR TO USING 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

© Copyright 2018 Xilinx, Inc. Xilinx, the Xilinx logo, Artix, ISE, Kintex, Spartan, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission by Khronos. All other trademarks are the property of their respective owners.

