SDSoC User Guide

Introduction to SDSoC

UG1027 (v2014.4) February 28, 2015

.

XILINX
ALL PROGRAMMABLE™

Revision History

The following table shows the revision history for this document.

Date	Version	Changes
02/28/2015	2014.4	Initial Release

Table of Contents

Overview

The Software Defined System on Chip (SDSoC) is part of a C/C++ based design environment for implementing heterogeneous embedded systems using the Zynq®-7000 All Programmable SoC. The SDSoC compiler and linker transform programs into complete hardware/software systems based on command line options that specify target platform and functions within the program to compile into programmable hardware.

Each hardware function runs as an independent thread, but SDSoC generates necessary hardware and software components to synchronize hardware and software and to preserve original program semantics, while enabling task level parallelism and pipelined communication and computation to achieve high performance.

Application code can involve many hardware functions, multiple instances of a specific hardware function, and invocations of a hardware function from different parts of the program. Based on program analysis, scheduling, and internal modeling of hardware functions and data transfers, SDSoC then generates a hardware system based on the selected platform, including hardware functions, DMAs, interconnects, hardware buffers, and other IP required to implement the program in Zynq. SDSoC selects data mover blocks based on transfer payload size as well as sender and receiver hardware interfaces.

SDSoC employs the Vivado® High Level Synthesis (HLS) tool to compile individual hardware functions to programmable logic, and invokes the standard GNU tool chain included in the design suite to compile source into object code for the ARM CPUs within the Zynq processing system. Through analysis of the program, SDSoC determines the dataflow between functions compiled to hardware and between hardware functions and software running on the CPU. The SDSoC linker automatically invokes tools within the Vivado® Design Suite to compile the system into an FPGA bitstream, i.e., firmware.

In addition to the hardware, SDSoC automatically generates, compiles and links hardware-specific software configuration code, and integrates into the program any associated drivers for generated IP blocks, while preserving the original program semantics. By generating complete applications from "single source", SDSoC allows you to iterate over design and architecture changes by refactoring at the program level, dramatically reducing the time to working program running on the target.

Getting Started

Download and install the SDSoC development environment according to the directions provided in the SDSoC Getting Started Guide, UG1028. It is important to ensure that any shell invoking SDSoC has its environment set properly using included setup scripts; the SDSoC development environment includes

supported versions of Vivado and Vivado HLS, and SDSoC tools, including an Eclipse/CDT-based GUI, command line tools, and ARM GNU toolchain.

User Design Flow

In this chapter, we briefly describe the top-level user flow for using the SDSoC design environment, assuming that the user already has the software application running on Zynq® Cortex-A9 ARM processors. Figure 1 shows the top-level user visible design flow.

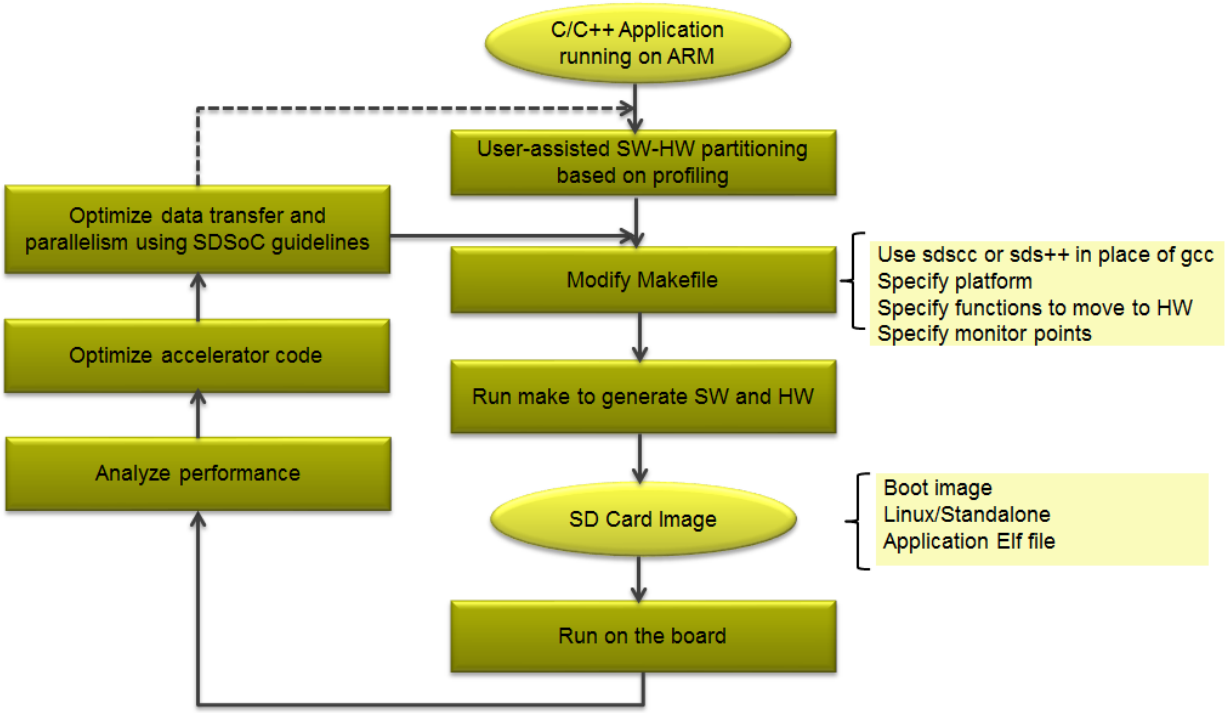


Figure 1: Top-level user design flow for using the SDSoC design environment

1. The first step is to identify compute-intensive hot spots in the application that can be migrated to FPGA fabric to achieve higher performance, and to isolate them into functions that you can compile for hardware.
2. Code compiled for FPGA fabric with SDSoC must conform to coding guidelines described in the chapter SDSoC Coding Guidelines. Furthermore, accelerator code must also conform to Vivado® HLS guidelines. For example, the code cannot invoke recursive functions, dynamically allocate of memory, or make unrestricted use of pointers. Please see the Vivado HLS documentation and training material for HLS guidelines.
   a. SDSoC can be invoked from the command line, from within a make file, or from the Eclipse-based GUI. The command line interface and make flows are well-suited to scripting flows. The

GUI provides a number of interactive features to simplify development, including project management and build automation. The SDSoC GUI also provides a number of error checking capabilities by extending static code analysis provided as part of the Eclipse CDT.

b. By default, the SDSoC linker creates an SD card image that consists of a boot image including the FPGA bitstream, a Linux image and the application binary. You can boot from this SD card and run the application on the target platform.

c. The SDSoC design environment provides a simple, source code annotation based time-stamping mechanism that can be used to measure application performance. SDSoC also integrates a more comprehensive set of performance and profiling capabilities, working in conjunction with its Eclipse-based GUI.

d. When the accelerators are the bottleneck to overall system performance, you will need to optimize the code compiled for FPGA fabric according to guidelines for Vivado HLS. Please refer to Vivado HLS documentation for more information. We also recommend that you take Vivado HLS training if you would like to get an in-depth understanding.

   Two main areas to focus on are:

   i. Specifying parallel execution using various pragmas so that Vivado HLS can generate parallel, high-performance hardware.

   ii. Increased bandwidth to local memories used by the accelerator.

e. SDSoC supports several pragmas to control parallelism and data transfer between CPU and accelerators, which are used to improve system performance. Please refer to the chapter SDSoC Pragma Specification.

f. You can iterate and do successive refinements to achieve your goal.

The next few chapters walk through some of these steps in a bit more detail using a simple matrix multiply example.

Overview

SDSoC includes command line tools with an interface patterned after standard compilers like gcc, as well as an Eclipse-based GUI. SDSoC command line options control compilation into hardware and software. SDSoC automatically invokes the arm-gcc compiler to target the ARM CPU in a Zynq® device, and Vivado® HLS to target FPGA fabric. You can of course invoke the arm-gcc compiler directly, but may find it convenient to invoke the SDSoC compiler (sdscc/sds++) on code at the hardware/software boundary (real or anticipated). When targeting FPGA fabric, you must invoke SDSoC rather than gcc.

Like gcc, the SDSoC compiler will typically be invoked using make to control the build process. You can use the SDSoC GUI to compile, run and debug code on an ARM processor, but in this document we will simply invoke make from a command shell.

In this warm-up chapter, you will compile and run a simple matrix multiplication algorithm on a Zynq device. The `mmult_sw` directory contains all source files. The main files are:

1. `makefile`: It will drive the compilation and link process from source code to full SD card image from which you can boot a Zynq device on the zc702 platform
2. `mmult_accel.cpp` : A text-book implementation of a matrix multiply algorithm
3. `mmult.cpp` : Contains test function main and supporting functions to call the matrix multiply. There is also another matrix multiply function, called `mmult_golden`, which is used as in conformance tests to make sure that the HW implementations produce correct results.

The steps to compile and run the code are as follows:

1. Table 1 shows a standard software `makefile` and a `makefile` that invokes the SDSoC compiler to generate an SD card image for a specific hardware platform (zc702). Initially, the code will run entirely in software.
2. Like gcc, there are two command-line scripts: `sdscc`, for compiling `C` code and `sds++` for compiling `C++` code. The first change is to define `CC = sds++` to invoke the SDSoC compiler.

   If desired, you can selectively call `sdscc/sds++` for some files and the ARM GNU compiler for others. You must, however, use `sdscc/sds++` for linking the code.
3. The next change is to add the SDSoC `-sds-pf` command option to the SDSoC flags (`SDSFLAGS`) to specify a target hardware platform.

   The SDSoC design environment currently supports a set of predefined platforms—please see the documentation for SDSoC options in the chapter **SDSCC/SDS++ Compiler and Linker Options**. SDSoC also supports user-defined platforms captured using the Vivado Design Suite.

```
APPSOURCES = mmult.cpp mmult_accel.cpp
EXECUTABLE = mmult.elf

CC = arm-xilinx-linux-gnueabi-g++
IDIRS = -I.
CFLAGS = -Wall -O3 -c ${IDIRS} -v



OBJECTS := $(APPSOURCES:.cpp=.o)

all: ${EXECUTABLE}

${EXECUTABLE}: ${OBJECTS}
	${CC} ${OBJECTS} -o $@
%.o: ../%.cpp
	${CC} ${CFLAGS} $< -o $@

clean:
	rm ${EXECUTABLE} ${OBJECTS}
```

(a)

```
APPSOURCES = mmult.cpp mmult_accel.cpp
EXECUTABLE = mmult.elf

SDSFLAGS = -sds-pf zc702
CC = sds++ ${SDSFLAGS}
IDIRS = -I.
CFLAGS = -Wall -O3 -c ${IDIRS}
LFLAGS = -O3

OBJECTS := $(APPSOURCES:.cpp=.o)

all: ${EXECUTABLE}

${EXECUTABLE}: ${OBJECTS}
	${CC} ${LFLAGS} ${OBJECTS} -o $@
%.o: %.cpp
	${CC} ${CFLAGS} $< -o $@

clean:
  rm  ${EXECUTABLE} ${OBJECTS}
```

(b)

Table 1: (a) A standard makefile for compiling code using ARM GCC

(b) Modified makefile to work with the SDSoC compiler

4. Run make to compile the code. The compiler will create a new subdirectory, called sd_card, which contains all the files needed to boot Linux and run the application on the targeted zc702 platform. This directory will contain the following:
   - Readme: Contains brief directions on how to run the code.
   - BOOT.BIN: This is the boot image that contains first stage boot loader (FSBL), boot program (Uboot) and the FPGA bitstream file.
   - uImage, devicetree.dtb, and uramdisk.image.gz: Linux boot image.
   - mmult.elf: Application binary
5. Copy the content of sd_card directory onto an SD card.
6. Insert the SD card, open a serial terminal window (e.g., using the SDSoC GUI Remote System Explorer perspective or a program like putty), and power on the board. Make sure that the board is setup to boot from the SD card – see the Readme file for the jumper settings.
7. Linux will boot, automatically log you in as root, and enter a bash shell. Currently, the SD card is mounted at /mnt.
8. Run the matrix multiply application using the following commands:

```
cd /mnt
./mmult.elf
```

The application will run and print the performance number as well as "Test passed" message.

Performance Measurement

As part of SDSoC, the sds_lib library provides a basic time stamp API to help you measure performance within a program.

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

By using this API to collect timestamps and differences, you can determine duration of key parts of your program. For example, you can measure data transfer or overall round trip execution time for hardware functions.

The file `mmult.cpp` provides an example usage; Table 2 demonstrates another.

```
#include "sds_lib.h"

unsigned long long total_run_time = 0;
unsigned int num_calls = 0;
unsigned long long count_val = 0;

#define sds_clk_start() {             \
     count_val = sds_clock_counter(); \
     num_calls++;                     \
}

#define sds_clk_stop()                      \
     long long tmp = sds_clock_counter(); \
     total_run_time += (tmp - count_val); \
}

#define avg_cpu_cycles()(total_run_time / num_calls)


#define NUM_TESTS 1024
extern void f();
void measure_f_runtime()
{
for (int i = 0; i < NUM_TESTS; i++) {
          sds_clk_start();
          f();
          sds_clk_stop();
}
printf("Average cpu cycles f(): %ld\n", avg_cpu_cycles());
}
```

Table 2 sds_clock_counter example

The SDSoC GUI provides a more comprehensive set of performance and profiling capabilities in conjunction with its command line tools.

Overview

This chapter shows how to update the code and the project to move the matrix multiply function from the ARM CPU to FPGA fabric. SDSoC automatically invokes Vivado® HLS as a cross-compiler for the matrix multiply function, and generates a system for the targeted hardware platform using data mover IP (either AXI DMA or AXI FIFO), as well as the code to control the accelerator and transfer data between software and hardware. The `mmult_hw` directory contains all the source files for this example.

To move the matrix multiply to HW, change the `SDSFLAGS` variable in the **`makefile`** to specify that the function `mmult_accel` should be migrated to HW:

```
SDSFLAGS = -sds-pf zc702 \
   -sds-hw mmult_accel mmult_accel.cpp -sds-end
```

The SDSoC flag **`-sds-hw`** specifies the name of the top-level function (and the file that it resides in) that should be cross-compiled into hardware using Vivado HLS. Each top-level function to be compiled for hardware must reside in a separate file, although this file can contain other functions including sub-functions of the top-level function. When compiling for hardware, it is also necessary to specify the target platform using the **`-sds-pf`** option so that SDSoC knows the device family.

The `mmult_hw` directory contains the modified **`makefile`**. Please follow the same steps as in the chapter **Running Code on an ARM Processor** to compile and run the application. The **`make`** compile and linking steps now take significantly longer than a pure software ARM compile, as the SDSoC compiler is calling Vivado HLS and Vivado implementation tools to generate the bitstream for the FPGA fabric.

The `sd_card_prebuilt` sub-directory contains a pre-built SD card image that you can use to run the example on the board while waiting for the hardware build to finish.

You will notice that the out-of-the-box application performance is poor, indeed slower than pure CPU-based execution. This occurs because, without additional guidance, HLS generates hardware that executes the matrix multiply loops sequentially at a much slower clock frequency than the ARM CPU (typically 4x or 6x slower), and because the hardware/software system involves DMA transfers whereas the CPU accesses cache memory for most of the computation. This is not to say that matrix multiply is a poor candidate for FPGA implementation using SDSoC. All of these considerations can be readily addressed using SDSoC (and Vivado HLS), but the point is that naively migrating functions from the ARM processing system to hardware may reduce overall system performance.

We will show how to optimize the code in the next few chapters, but before doing so, it is worthwhile to take an aside to describe at a conceptual level what is going on under the hood of the SDSoC compiler and linker.

SDSoC Compilation and System Linking

The SDSoC design environment performs the following steps to compile the matrix multiply application into a SW-HW system.

1. The compiler calls Vivado HLS to cross-compile the `mmult_accel` function for the programmable logic fabric in a Zynq device.
2. It then analyzes the application code starting from `mmult_accel` function calls to determine the data communication patterns and transfer requirements to and from the hardware function. In this case, all communication is between the hardware function and main memory.
3. It builds an AXI-based data motion network in hardware to communicate the data to the accelerator and back using the analysis results from the previous step. This network uses standard AXI IP for transport, but the specific transport protocols are abstracted completely (as can be seen in this example). There are pragmas to override some of the decisions taken by the compiler in building the data motion network. (See the chapter **SDSoC Pragma Specification** for details of various pragmas.)
4. SDSoC preserves the original function call abstraction at the application level, automatically and efficiently integrating device drivers and additional code to transport data through the system.
5. The SDSoC design environment builds upon a connectivity framework (CF) that supports multi-lingual, heterogeneous computing. The connectivity framework consists of an abstract channel-based data model, software APIs for data transfer and allocation, and a "system linking" tool for generating a Vivado-based hardware system from the internal data model.
6. SDSoC generates a "stub function" in addition to the original `mmult_accel` function without interface change, SDSoC also changes the caller code to call the "stub function" and then automatically links this stub function. This stub function contains calls to the SW APIs defined by the CF layer to transfer the data and control information throughout the accelerator network. Users can also employ these APIs directly, if desired,
7. SDSoC calls the `system_linker` tool provided by the CF layer to generate Vivado IPI project/TCL from the connectivity description generated in Step 5, and then to generate the bitstream using Vivado implementation tools.

8. SDSoC then calls the ARM GNU compiler on the rest of the code and the stub generated in Step 6, and links them with predefined SW API libraries to generate the ELF file.
9. The last step is to generate a SD card image (as described earlier) using the bitstream, the ELF file and the prebuilt Linux kernel image.

Software Stub Implementation

The SDSoC design environment builds upon a connectivity framework layer consisting of a high-level description of logical and physical connections between system components (hardware and software), and a SW API to transfer data across connections captured in the system description.

The automatically generated software stub functions employ these APIs to synchronize control between CPU and hardware functions, and to transfer data between hardware and software components. The additional stub function for our example, reproduced in Table 3, is contained in `_sds/swstubs/mmult_accel.cpp`.

The highlights are:

1. The first `cf_send_i` sends data to enable the accelerator and the subsequent `cf_wait` waits for the initialization to complete.
2. The next two `cf_send_i` send the two arguments to the accelerator.
3. The `cf_receive_i` receives the result from the accelerator.
4. Finally, the three `cf_wait` wait for all the transactions to complete.

IMPORTANT: An important thing to note is that the code is completely independent of the actual method used for data communication. The implementation of `cf_send_i` and `cf_receive_i` get translated to appropriate driver calls based on the configuration data provided by the system_linker tool.

```
void _p0_mmult_accel (float in_A[A_NROWS*A_NCOLS],
                      float in_B[A_NCOLS*B_NCOLS],
                      float out_C[A_NROWS*B_NCOLS])
{
  int start_seq[3];
  start_seq[0] = 0x00000003;
  start_seq[1] = 0x00010001;
  start_seq[2] = 0x00020000;
  cf_request_handle_t request_swinst_mmult_accel_cmd;
  cf_send_i(&(swinst_mmult_accel.cmd_mmult_accel), start_seq, 3*sizeof(int),
&request_swinst_mmult_accel_cmd);
  cf_wait(request_swinst_mmult_accel_cmd);

  cf_send_i(&(swinst_mmult_accel.in_A_PORTA), in_A, 1024 * 4, &request_0);
  cf_send_i(&(swinst_mmult_accel.in_B_PORTA), in_B, 1024 * 4, &request_1);

  size_t num_out_C_PORTA;
  cf_receive_i(&(swinst_mmult_accel.out_C_PORTA), out_C, 1024 * 4,
&num_out_C_PORTA, &request_2);

  cf_wait(request_0);
  cf_wait(request_1);
  cf_wait(request_2);
}
```

Table 3: The replacement stub function generated by the SDSoC design environment

Overview

The matrix multiply algorithm used in the chapter **Moving the Matrix Multiply into Hardware**, was not optimized for Vivado® HLS. In this chapter, we add Vivado HLS pragma/directives to the code and restructure it to improve the performance. This chapter is not a tutorial on how to optimize the code using Vivado HLS. The intent simply is to show that these optimizations can lead to significant performance gains. Please see Vivado HLS User Guide, (UG-902) for details on how to write high-performance code for Vivado HLS.

There are two main optimizations described below. The code in the directory `mmult_optimized` reflects both these optimizations. You can start with the code in `mmult_hw` and apply these optimizations one at a time.

Increasing the Parallelism

With the absence of any directives, Vivado HLS generates the hardware to execute the three loops sequentially. The first step to improve the performance is to add the following pragma after the second loop:

```
#pragma AP PIPELINE II=1
```

This pragma tells the compiler to optimize/parallelize the code so that the successive iterations of the second loop can be scheduled one cycle apart. To accomplish this, one of the optimizations that Vivado HLS tool performs is to unroll the innermost loop. Note that HLS cannot guarantee it will find a schedule with II=1, but if it does not, it will schedule the loop with the lowest achievable II.

With this change, performance improves significantly and is now close to the SW execution time.

Increasing the Memory Bandwidth for the Inner Loop

Since the pragma introduced in the last section implies that the innermost loop is unrolled, all 32 multiplications of the elements of a row of `in_A` with the elements of a column of `in_B` can be done in parallel. That, however, requires that 32 elements of `in_A` and 32 elements of `in_B` can be read in parallel from their respective memories. The default in Vivado HLS is to provide 1 or 2 ports to memories, but there are pragmas to partition an array into sub-arrays (or banked memories) to increase

the memory access throughput. The following two pragmas specify the partitioning for the matrix multiply example:

```
#pragma HLS array_partition variable=in_A block factor=16 dim=2
#pragma HLS array_partition variable=in_B block factor=16 dim=1
```

The first pragma directs the tool to partition array `in_A` into 16 partitions, all of which can be accessed in parallel. Moreover, the partitioning is done so that different elements of a row are in different partitions by specifying `dim = 2`. Similar comments apply to the second pragma. Please consult the Vivado HLS User Guide for more details about these pragmas.

To keep the external interface identical to the previous design, we make these partitioned arrays internal to the Vivado HLS code and perform an explicit copy from the matrix multiply arguments to these arrays. The code in the directory `mmult_optimized` has been restructured for this reason.

These two optimizations significantly improve the performance of the code, even in the presence of the overhead of explicit copying.

Overview

In this chapter, we discuss some of the optimizations that are specific to the SDSoC design environment. The optimized code and **makefile** are in the `mmult_optimized_apf` directory.

Memory Allocation for Efficient Data Transfer

The code that calls matrix multiply in `mmult.cpp` uses standard C **malloc** to allocate the arrays. **malloc** allocates memory that is contiguous in the virtual memory space, but the allocated memory may be distributed over pages that are not contiguous in the physical memory. Transferring such an array from the main memory to the accelerator requires a number of steps to prepare the data to transfer. For example, map virtual addresses to physical addresses, collect all the pages, "pin" all the pages to ensure they are in the memory during the transfer, etc. All these operations contribute to overhead. Furthermore, the hardware implementation requires additional logic to access the paged memory.

Linux also provides a driver for allocating physically contiguous memory, and allocating arrays using this driver can reduce the overhead associated with **malloc**. The SDSoC design environment provides the following two APIs that encapsulate the contiguous memory allocation driver:

```
sds_alloc (size_t size) - To allocate memory
sds_free (void *memptr) - To free the memory
```

The code in `mmult.cpp` in the `mmult_optimized_apf` directory replaces **malloc**/**free** with these APIs to allocate and free the memory. Note that in order to use these APIs 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.

Clock Frequency

Each platform supported by the SDSoC design environment provides a fixed number of clocks that are numbered 0,1,2… with pre-specified frequencies. Please refer to the chapter **SDSCC/SDS++ Compiler and Linker Options** for the details on the clocks frequencies supported by each platform. The current default is to use the 100MHz clock for both the accelerator and the data motion network. You can change the clock frequencies by adding the following options to sdscc/sds++ in the **makefile**:

- `-clkid n`: Specifies that the clock number n should be used for the hardware accelerator. Add this to the command line that compiles a function into HW accelerator.
- `-dmclkid n`: Like **`clkid`**, it specifies the clock to use for the data motion network (e.g., DMA, FIFO, AXI-interconnect, etc). Currently, all AXI interfaces in the design run at this clock frequency. Non-default values must be specified on the command line for the link phase.

IMPORTANT: *In this release of SDSoC you can specify only a single clock ID for both the hardware functions and the data motion network. This restriction will be removed in a future release.*

Synchronization Method

SDSoC uses two low level mechanisms to synchronize between an accelerator and the CPU – interrupts and polling. In many cases, polling gives better performance, as interrupt handling has some overhead in Linux. You can control the synchronization using the following option to to sdscc/sds++ in the **`makefile`** (the default is interrupt).

`-poll-mode 1`: It specifies that the compiler should use polling instead of interrupts for synchronization between the CPU and the accelerator.

The **`makefile`** has been modified to include the clock frequency and polling options.

With the optimizations in this chapter, you should see further performance improvements.

Overview

If there are multiple calls to an accelerator in your application, the SDSoC compiler allows you to pipeline these calls to overlap the data transfer with the accelerator computation. In the case of a matrix multiply call, the following events take place:

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

Figure 2 shows a matrix multiply design on the left side and a time-chart of these events for two successive calls that are executing sequentially. The second call starts executing when the first one completes.

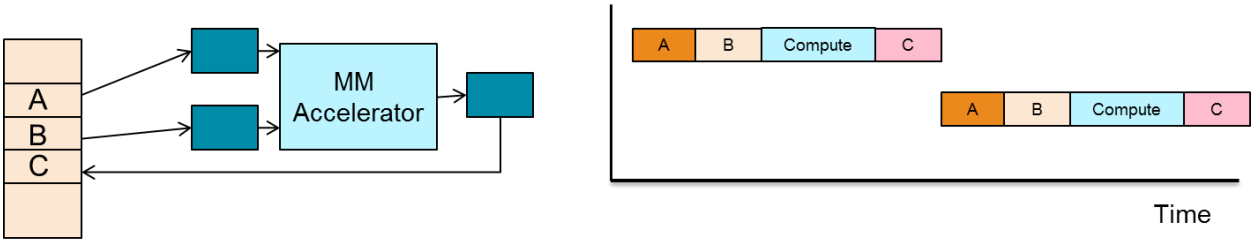


Figure 2: Sequential execution of matrix multiply calls

Figure 3 shows the two calls executing in a pipelined fashion. The data transfer for the second call starts as soon as the data transfer for the first call is finished and overlaps with the execution of the first call. To enable the pipelining, however, we need to provide extra local memory to store the second set of arguments while the accelerator is computing with the first set of arguments. The SDSoC design environment will generate these memories, called *multi-buffers*, under the guidance of the user.

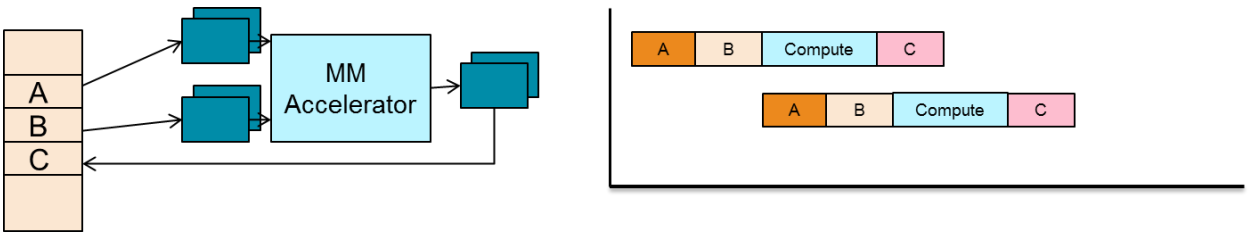


Figure 3: Pipelined execution of matrix multiply calls

Specifying the task level pipelining requires rewriting the calling code using the following pragma and API. The code for this example is in the `mmult_pipelined` directory.

- **Async pragma:** **`#pragma SDS async(ID)`**

  This pragma tells the compiler that the function call that immediately follows should be executed asynchronously. That is, the processor initiates the call but continues with its own execution rather than waiting for the call to finish. `ID` specifies the unique ID of the accelerator that is used to execute the call, and the same ID is used in a subsequent wait statement to synchronize with the accelerator. SDSoC will create a separate instance of the accelerator for each unique ID.

- **`sds_wait(ID)` or `#pragma SDS wait(ID)`:** Used to wait for an asynchronous call to complete. We assume that tasks executing on an accelerator complete in the same order as they were issued. That is, they never go out-of-order. The SDSoC runtime keeps a queue associated with each ID (for each accelerator), which is used to keep the wait-handles associated with the tasks. When an asynchronous function starts, the wait-handles for all its arguments and results are pushed in the queue. An `sds_wait` waits on the handles associate with the task at the front of the queue.

The pipelined code in `mmult.cpp` consists of three loops corresponding to the following three stages:

- **Pipeline fill stage**: This stage fills the pipeline by issuing a fixed number of calls.
- **Steady state**: In this stage, a new task is issued when an earlier task has finished.
- **Pipeline drain stage**: This stage waits for all calls in the pipeline to complete.

A number of points to note about the use of async pragma and `sds_wait`:

- An asynchronous function call implies that the call can access its inputs and write its outputs anytime between the start of the call and the corresponding wait. In other words, the application should not access any argument to an asynchronous call between the start of the call and the corresponding `sds_wait`. The arguments should be accessed only after the `sds_wait`; otherwise, the result will be non-deterministic.
- An asynchronous call can complete anytime between its start and the corresponding wait. As a result, it is correct to assume that the call can complete at the time it was issued. In other words, the code can still be executed sequentially.

Chapter 8 Accelerator to Accelerator Communication

Overview

When the output of an accelerator is an input to another accelerator, it is sometimes unnecessary for the functions to communicate through main memory, for example when the data streams sequentially, and when the data transfers between hardware functions can be buffered in programmable logic. In these cases, you can significantly improve performance and system efficiency by avoiding the round trip from the first function to DDR and back to the second function.

The SDSoC design environment determines these accelerator-to-accelerator connections by analyzing the code that calls accelerators.

The `mmult_cascade` directory contains an example of this type of communication. The code in `mmult.cpp` has the following two calls to `mmult_accel`.

```
mmult_accel(tin1Buf, tin2Buf, toutBufHwInter);
mmult_accel(toutBufHwInter, tin2Buf, toutBufHw);
```

The compiler determines that `toutBufHwInter` produced by the first call is consumed by the second call without any other access to the buffer. Therefore, the data can be communicated directly from the first accelerator to the second.

Chapter 9 Multiple Instances of an Accelerator

Overview

Multiple calls to a hardware function are usually mapped to a single accelerator. However, by using the `async` pragma discussed earlier, you can direct the compiler to generate multiple accelerators for a function and map different calls to different accelerators to improve the performance of the application. For example, the following code will generate two instances of the matrix multiply accelerator, one for each unique Id.

```
#pragma SDS async(1)
 mmult_accel(…);

#pragma SDS async(2)
 mmult_accel(…);
. . .
 sds_wait(1);
 sds_wait(2);
```

If there are direct connections between accelerators, the compiler *may* decide to create multiple accelerators to improve performance when it can deduce semantics preserving behavior. For example, the code discussed in the earlier chapter **Accelerator to Accelerator Communication** will actually result into two accelerators being created because the two calls to `mmult_accel` have very different connectivity. Rather than communicating between the two calls to mmult_accel through main memory, the compiler creates two instances of the hardware function, and avoids the round trip transfer of the output from the first call to the input of the second call.

Chapter 10 Using Shared Memory Regions

Overview

Instead of transferring data from the processor's memory to the accelerator's local memory (BRAM), the processor and the accelerator can access a common region of shared memory. This is generally the easiest way to convert an existing software function into an accelerator, although some pragma's need to be added in order to satisfy current Vivado® HLS requirements.

Assume the original looks like the following:

```
void foo( int *array_ptr, int num_elems, int start_index) {
   for(…) { array_ptr[index] += 1); }
}
int main(…) {
   int *array_ptr = (int *)malloc(num_elems*sizeof(int));
   // increment the first half of the array
   foo(array_ptr, num_elems/2, 0);
}
```

This code can be accelerated with a few small changes described below:

```
int foo( int *array_ptr, int num_elems, int start_index) {
#pragma HLS INTERFACE m_axi port=array_ptr depth=19200 offset=direct
#pragma HLS INTERFACE ap_ctrl_hs port=return
   for(…) { array_ptr[index] += 1); }
   return 0;
}
int main(…) {
   int *array_ptr = (int *)sds_alloc(num_elems*sizeof(int));
   // increment the first half of the array
   foo(array_ptr, num_elems/2, 0);
}
```

The main changes to the accelerated function are:

- The HLS pragmas cause an AXI Master port interface to be generated for the accelerator hardware.

The main changes to the caller of the accelerated function are:

- `malloc()` is replaced by `sds_alloc()` to allocate physically contiguous memory to allow the accelerator to access a single chunk of memory instead of a scattered list of virtual pages.

The pragma "`SDS data copy…`" is used to indicate how much memory is going to be shared. This is useful in case the allocation is not done in the same function that calls `foo()`.

Note: These SDS pragmas are placed in the header file for `foo()` while the HLS pragmas are placed in the `body(source)` of `foo()`. Presently, we require any accelerators using AXI Master ports to return a value back.

Sample code using shared memory is available in the `samples/mmult_aximm_all` directory.

When `main()` calls `foo()` and passes a shared memory region to `foo()`, `main()` should not modify or access the shared memory region until `foo()` completes. It is left to the user to implement any fine grain sharing or synchronization between `main()` and `foo()`, if needed.

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.

Fast Performance Estimation

This chapter shows how to get an estimate of the performance improvement as a result of moving different functions to the hardware for acceleration. Typically the build process for moving functions to hardware takes an hour or more. Using the performance estimation flow, you can get an estimate of performance impact of moving one or more functions to hardware in a matter of minutes.

Before obtaining an estimate of performance improvement, the flow requires an initial software based run on the board, which collects performance related data when the code runs only on the processor. This is done by following steps 1 and 2 listed below. Once the software performance data file is obtained from the board, it is possible to estimate the resources used and performance improvement for the functions chosen by following the step 3 below. An example of this flow can be seen in the directory `mmult_performance_estimation`, which uses the code in `mmult_optimized_apf` and estimates the improvement of moving the function `mmult_accel()` to hardware. In the supplied example, it is possible to run the estimation flow directly using the supplied software data file that is already present.

1. To collect software run data, add the options `-perf-root` and `-perf-funcs` to the `makefile`.
   a. The option `-perf-root` is typically the function `main()`, but the option exists in case that the user code does not have main() and the top level is called through some library which is the main driver.
   b. `-perf-funcs` is the list of functions (comma separated) which the user might be interested in moving to hardware.
2. Run the `makefile` and boot up the generated sd_card on the board. Run the application executable on hardware to produce a file called sw_perf_data.xml on the sd card. Copy this xml file back to the workspace.

> ***CAUTION!*** *: After running the sd_card image on the board for collecting profile data, please run cd /; sync; umount /mnt; This ensures that the sw_perf_data.xml file is written out to the sd card*

3. To get an estimate of performance improvement and resource usage, add the flag `-perf-est` to the `makefile` and build the target.
   a. This causes the performance estimation flow to get invoked and the regular flow is disabled. Instead of generating the hardware bitstream, SDSoC will call Vivado HLS on the functions

chosen for acceleration and run the estimation tool, which reports the estimated performance and resource usage.

Note: In the presence of asynchronous calls (using async pragma, sds_wait, etc), we do not support the fast performance estimation flow.

Overview

To use an IP library, include the appropriate header file in your source code. When compiling the code that calls a library function, provide the path to the header file using the –I switch.

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

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

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

In the example above, the compiler will link against the library `libfir.a` indicated by -lfir and located at `<path to library>`.

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

Vivado HLS Libraries

Vivado HLS libraries are delivered as source code in the Vivado HLS installation. To use an HLS library, follow the HLs examples and include the appropriate header in your source code.

You can find an example of using the HLS math library in the `samples/hls_math` directory. Two files are created to use hls_math.h and implement a square root function.

The file `my_sqrt.h` contains:

```
#ifndef _MY_SQRT_H_
#define _MY_SQRT_H_

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

```
#endif

void my_sqrt(float x, float *ret);

#endif // _SQRT_H_
```

The file `my_sqrt.cpp` contains:

```
#include "my_sqrt.h"

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

The Makefile has the commands to compile these files:

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

Overview

In addition to Linux applications that run on Zynq® family devices, SDSoC supports standalone mode, which allows users to compile their programs to run directly on the hardware, without any operating system. SDSoC links in a library that provides the services normally provided by the target operating system.

Usage

In order to compile and link an SDSoC program for standalone mode, the Makefile should include "-target-os standalone" in CFLAGS, as well as LFLAGS.

The SD boot image consists of a single file BOOT.BIN in the sd_card directory, that contains the first stage boot loader (FSBL) as well as the user application, which is invoked directly after powering on the board.

Supported Platforms

Standalone mode is supported for four platforms namely, zc702, zc706, zed and microzed.

Limitations

Currently, standalone mode in SDSoC does not support multi-threading, virtual memory, or address protection as documented in UG643. Access to the filsystem is not through the usual C api, but instead through a special api using libxilffs. Please refer to the sample program file_io_manr_sobel_standalone to see an example of its use. This program can be compared with the linux version i.e. file_io_manr_sobel to see what changes are necessary for accessing the filesystem. In general, the procedure to access the filesystem is to include a few extra files, use different types (e.g. FIL instead of FILE), use a slightly different api for filesystem access (e.g. f_open instead of fopen) and to disable DCache before doing any file ops.

IMPORTANT: *Note that on zed board, serial connection to the board takes a couple of seconds. If your program runs for a time shorter than that, you will never see its output. When the zed board is power cycled, the serial connection goes down and it is not possible to see the output in the subsequent run either. zc702 and zc706 boards keep the serial connection alive across power cycles and hence do not suffer from this limitation.*

Overview

In addition to Linux applications that run on Zynq® family devices, SDSoC supports applications that use the FreeRTOS real time operating system from Real Time Engineers Ltd, which allows users to compile their programs with a real time kernel using APIs for scheduling, inter-task communication, timing and synchronization.

SDSoC includes FreeRTOS v8.1.2 header files and a pre-configured library containing the real time kernel, API functions and Zynq-specific platform code. It also builds the standalone library that provides drivers and functions required to support a C/C++ bare-metal application.

Usage

In order to compile and link an SDSoC program for FreeRTOS, the Makefile should include the option "-target-os freertos" in all compiler and linker invocations in the Makefile. This is typically specified in an SDSoC variable, which in turn is included in a compiler toolchain variable, as shown below:

```
SDSFLAGS = -sds-pf zc702 -target-os freertos \
        -sds-hw mmult_accel mmult_accel.cpp -sds-end \
        -poll-mode 1
CPP = sds++ ${SDSFLAGS}
CC = sds ${SDSFLAGS}
:
all: ${EXECUTABLE}

${EXECUTABLE}: ${OBJECTS}
        ${CPP} ${LFLAGS} ${OBJECTS} -o $@

%.o: %.cpp
        ${CPP} ${CFLAGS} $< -o $@
:
```

When SDSoC links the application ELF file, it builds a standalone (bare-metal) library for you, provides a predefined linker script and uses a pre-configured FreeRTOS kernel using headers and a pre-built library, and includes their paths when it calls the ARM GNU toolchain (you do not need to specify the paths in your Makefile):

```
<path_to_install>/SDSoC/2014.4/arm-xilinx-eabi/include/freertos
<path_to_install>/SDSoC/2014.4/arm-xilinx-eabi/lib/freerttos
```

The SD boot image consists of a single file BOOT.BIN in the sd_card directory that contains the first stage boot loader (FSBL) as well as the user application, which is invoked directly after powering on the board.

SDSoC GUI flows for working with FreeRTOS applications are the same as those for standalone (bare-metal) applications, except the target OS is specified as FreeRTOS.

The user application code needs to include the following:

- hardware configuration function
- task functions and task creation calls using the xTaskCreate() API function
- scheduler start call using the vTaskStartScheduler() API function
- callback functions such as vApplicationMallocFailedHook(), vApplicationStackOverflowHook(), vApplicationIdelHook(), vAssertCalled(), vApplicationTickHook(), and vInitialiseTimerForRunTimeStats

Simple SDSoC applications based on the Zynq demo included in the FreeRTOS v8.1.2 software distribution are available in the SDSoC GUI application wizard and in the SDSoC installation:

```
<path_to_install>/SDSoC/2014.4/samples/mmult_datasize_freertos
<path_to_install>/SDSoC/2014.4/samples/mmult_optimized_sds_freerttos
```

User or sample applications that normally target the Standalone BSP can be built using the –target-os freertos option compile and link, but the FreeRTOS linker script is used and predefined callback functions found in the prebuilt FreeRTOS library are used. Applications built this way do not explicitly call FreeRTOS API functions and run as standalone applications. While it is possible to begin FreeRTOS application development in this way, it is recommended that FreeRTOS API functions and callbacks be incorporated as early as possible.

Supported Platforms

FreeRTOS mode is supported for two platforms namely, zc702 and zc706.

Limitations and Implementation Notes

SDSoC FreeRTOS support uses the standalone board support package (BSP) library and includes the same limitations as standalone mode.

SDSoC uses a pre-configured FreeRTOS v8.1.2 library that has been prebuilt for the user, and dynamically built (at application link time) standalone library. Characteristics of the FreeRTOS library include:

- Uses the standard FreeRTOS v8.1.2 distribution for platform independent code, platform dependent code for Uses the default FreeRTOSConfig.h file included with FreeRTOS v8.1.2 (see the FreeRTOS

reference http://www.freertos.org/a00110.html , with downloads available at http://sourceforge.net/projects/freertos/files/FreeRTOS)

- Uses heap_3.c for its memory allocation implementation (see the FreeRTOS reference http://www.freertos.org/a00111.html)
- Uses source from these FreeRTOS v8.1.2 distribution folders: Demo/CORTEX_A9_Zynq_ZC702/RTOSDemo/src, Source, Source/include, Source/portable/GCC/ARM_CA9 and Source/portable/MemMang
- Uses a linker script found in (to temporarily use a modified version of this file instead, make a copy of the file and add the linker option –Wl,-T –Wl,<path_to_your_linker_script> to the sdscc/sds++ command line used to create the ELF file) <path_to_install>/SDSoC/2014.4 /platforms/<platform>/freertos/lscript.ld
- Based on the porting description for Zync ZC702 found at http://www.freertos.org/RTOS-Xilinx-Zynq.html, including replacement functions for memcpy(), memset() and memcmp() as part of the prebuilt library rather than user application code – does not use a Xilinx® SDK-based BSP package
- Includes pre-defined callback functions to enable standalone applications to be linked with the sdscc/sds++ –target-os freertos option (it is recommended that the user define their own versions of these functions as part of their application)
    - vApplicationMallocFailedHook
    - vApplicationStackOverflowHook
    - vApplicationIdleHook
    - vAssertCalled
    - vApplicationTickHook
    - vInitialiseTimerForRunTimeStats

Changing the FreeRTOS Configuration or Version

SDSoC FreeRTOS support uses a prebuilt library using the default FreeRTOSConfig.h file included with the v8.1.2 software distribution, along with a predefined linker script.

If you wish to change the FreeRTOS v8.1.2 configuration or its linker script, or use a different version of FreeRTOS, follow the steps below (replace zc702 with zc702 if you are using that platform):

- Copy the folder <path_to_install>/SDSoC/2014.4/platforms/zc702 to your own folder
- If you only wish to modify the default linker script, modify the file <path_to_your_platform>/zc702/freertos/lscript.ld
- If you wish to change the FreeRTOS configuration (FreeRTOSConfig.h) or version, build a FeeRTOS library as libfreertos.a, add include files to the folder <path_to_your_platform>/zc702/freertos/include, add the library libfreertos.a to <path_to_your_platform>/zc702/freertos/lib, and change the paths in

<path_to_your_platform>/zc702/zc702_sw.pfm for the section containing the line “xd:os=”freertos” (xd:includeDir=”freertos/include” and xd:libDir=”freertos/lib”)

- In your Makefile, change the SDSoC platform option from –sds-pf zc702 to –sds-pf <path_to_your_platform>/zc702

The SDSoC folder <path_to_install>/SDSoC/2014.4/tps/FreeRTOS includes the source files used to build the pre-configured FreeRTOS v8.1.2 library libfreertos.a, along with a simple Makefile and a SDSoC_readme.txt file. To rebuild the library, open a command shell, run the SDSoC <path_to_install>/SDSoC/2014.4/settings64 script to set up the environment to run command line tools (including the ARM GNU toolchain for Zynq), copy the folder to a local folder, modify FreeRTOSConfig.h and run the make command. See the SDSoC_readme.txt file for additional requirements and instructions.

If you are not using FreeRTOS v8.1.2, refer to the notes in the SDSoC_readme.txt file describing how the source was created from the official software distribution. After uncompressing the ZIP file, a very small number of changes were made (incorporate memcpy, memset and memcmp from the demo application main.c into a library source file, change include file references from Task.h to task.h) and the folder structure is the same as the original. If the folder structure is preserved, the Makefile created to build the preconfigured FreeRTOS v8.1.2 library can be used.

Overview

It is sometimes useful to read video data from a file and write back the processed data to a file, instead of reading and writing frame buffers. A simple example, called `file_io_manr_sobel`, illustrates the methodology. The example uses the base zc702 platform. The overall structure of the `main()` function is simply:

```
main() {
   read_frames(in_filename, frames, rows, cols, …);
         process_frames(frames, …);
               write_frames(out_filename, frames, rows, cols, …);
}
```

Since there is no need for synchronization of the input and output with the video hardware, the software loop in `process_frames()` is simple.

```
for (int loop_cnt = 0; loop_cnt<frames; loop_cnt++) {
    // set up manr_in_current and manr_in_prev frames
    manr( nr_strength,manr_in_current, manr_in_prev, yc_out_tmp);
    sobel_filter(yc_out_tmp, out_frames[frame]);
  }
```

The input and output video files are in YUV422 format. The `platform` directory contains sources for converting these files to/from the frame arrays used in the accelerator code. The makefile in the top level directory compiles the application sources along with the platform sources to generate the application binary.

Overview

In addition to compiling C/C++ code into hardware using Vivado HLS, SDSoC can map a hardware function onto an IP block written in a hardware description language (HDL) like Verilog or VHDL. A previous chapter described how to link IP libraries into a program using SDSoC, but in this chapter we describe by example how to link an IP block into a program as a C-callable function directly from the SDSoC command line.

We will step through the process of integrating into SDSoC the Xilinx FIR Filter Compiler v7.1, an IP block in the Vivado Design Suite that computes many digital signal processing functions and micro-architectures. Please refer to the `samples/hdl_fir_filter` directory in the SDSoC install area.

HDL IP

You must first package your HDL IP into the Vivado Design Suite as described in *Chapter 8: Creating and Packaging IP* in the *Vivado Design Suite User Guide UG896: Designing with IP* [1].

The IP Packager creates a directory structure for the HDL and other source files, and an IP Definition file (`component.xml`) that conforms to the IEEE-1685 IP-XACT standard. Please refer to the UG for more details on the IP Packaging and installation process. Once you have integrated your HDL IP into Vivado as described in UG896, the process to integrate into an SDSoC flow can be the same as the process for the FIR filter.

IMPORTANT: *For details on how to integrate HDL IP into Vivado Design Suite., refer to Chapter 8 of the Vivado Design Suite User Guide (UG896)* [1]

In this document, we expose only the most basic software interface to the FIR Filter Compiler, a fixed coefficient, single rate finite impulse response filter. For additional details including instructions for creating a C-callable library for your IP, please refer to the document *SDSoC Platforms and Libraries*[2].

SDSoC-Related Files

SDSoC requires several meta-data files to integrate your IP into the compilation and linking process, described in subsequent sections of this document.

- `fir.c` – contains a C/C++ function interface definition for calling IP
- `fir.params.xml` – contains IP parameters values that should be set by SDSoC
- `fir.fcnmap.xml` – contains a mapping from the function interface to the IP interface

C/C++ Function Call Definition for HDL IP

You must first decide how application software will invoke your IP, by specifying a function signature that SDSoC will use to call your IP. In this example we restrict ourselves to a single function, but [2] describes how to support multiple functions that target your IP block.

In addition to the function prototype, you must provide a function body, which can be empty. When SDSoC compiles your function with `-sds-hw` option, it overwrites the function body with data transfer API calls to communicate between the CPU and the hardware function; all other contents of the file are retained in the stub function.

Optionally, you can provide a function definition that can be cross-compiled for ARM when compiled by SDSoC without the `-sds-hw` option. The file `fir.c` contains a simple function definition for the FIR filter that can also be cross-compiled for the ARM CPU, but this provides no guarantee of being a bit-accurate model for the IP core (such a model exists on xilinx.com for x86 architecture).

```
#include "fir.h"
void fir(signed char X[N], short Y[N])
{
     // SDSoC replaces function body by CF API
     // calls for data transfer between CPU / IP

     // filter coefficient vector
     static signed char H[N] =
          {  6,   0,  -4,  -3,   5,   6,  -6, -13,
             7,  44,  64,  44,   7, -13,  -6,   6,
             5,  -3,  -4,   0,   6,   0,   0,   0,
             0,   0,  0,    0,   0,   0,   0,   0};
     // delay line
     static signed char D[N] =
          {0,  0,  0,  0,  0,  0,  0,  0,
           0,  0,  0,  0,  0,  0,  0,  0,
           0,  0,  0,  0,  0,  0,  0,  0,
           0,  0,  0,  0,  0,  0,  0,  0};
     int i, j, k;
     for (i = 0; i < N; i++) {
          Y[i] = 0;
          D[i] = X[i];
          for (j = 0, k = i; k >= 0; j++, k--)
               Y[i] += D[j] * H[k];
     }
```

```
}
```

IP Parameters

The FIR Filter Compiler has many user-customizable parameters. In our example, we override IP default values set during the IP packaging process by specifying the parameters and values in `fir.params.xml`.

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

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

Argument Mapping to the IP Interface

Having provided a C/C++ function interface and the IP customization parameters, we now must define a mapping between the function called by software and the hardware IP interface that SDSoC will connect to the data interconnection network. This mapping is captured in an XML file `fir.fcnmap.xml`.

SDSoC incorporates this information into its program-level dataflow analysis to determine how the CPU will communicate with your IP, and into the software stub generation phase so application code will transfer data between CPU and hardware while preserving the original function call semantics.

Specific meta-data includes the following.

- Function name – the name of the software function
- Component name – is the IP type name, i.e., the IP-XACT "name" for the Vivado-packaged IP
- C argument name – an address expression for a function argument, for example "`x`" (pass scalar by value) or "`*p`" (pass by pointer).
- Function argument direction – either "in" or "out".
- Bus interface – the name of the IP port corresponding to a function argument.
- Port interface type – the corresponding IP port interface type, which currently must be either "aximm" (slave only), "axis".
- Address offset – hex address, e.g., "0x40", required for arguments mapping onto "aximm" slave ports.
- Data width – number of bits per datum.
- Array size – number of elements in an array argument.

The arguments must be specified in the XML file in the same order as they occur in the function signature.

The example file for the FIR Compiler example is `fir.fcnmap.xml`.

```
<?xml version="1.0" encoding="UTF-8"?>
<xd:repository xmlns:xd="http://www.xilinx.com/xd">
  <xd:fcnMap xd:fcnName="fir" xd:componentRef="fir_compiler">
    <xd:arg xd:name="X"
            xd:direction="in"
            xd:portInterfaceType="axis"
            xd:dataWidth="8"
            xd:busInterfaceRef="S_AXIS_DATA"
            xd:arraySize="32"/>
    <xd:arg xd:name="Y"
            xd:direction="out"
            xd:portInterfaceType="axis"
            xd:dataWidth="16"
            xd:busInterfaceRef="M_AXIS_DATA"
xd:arraySize="32"/>
  </xd:fcnMap>
</xd:repository>
```

Calling an IP Hardware Function in SDSoC

To compile and link code that invokes an HDL IP hardware function, you must call SDSoC with the –sds-pf and –sds-hw flags. In addition, you must specify several other command line arguments.

- `-vlnv <vendor:library:name:version>` – the IP-XACT identifier for the IP core
- `-ip-map <fcnmap.xml>` – the hardware function map described in this document
- `-ip-params <ip_params.xml>` – the IP parameters file described in this document

Example:

```
sdscc –sds-pf zc702:Linux -sds-hw fir fir.c –vlnv xilinx.com:ip:fir_compiler:7.1
     -ip-map fir.fcnmap.xml –sds-end fir.c
```

By default, sdscc searches for the HDL IP in <vivado_installation_folder>/data/ip. To add a path to a folder containing packaged user IP repository, add the option `-ip-repo <repository_path>`, and sdscc will add the folder to its search path ahead of the Vivado repository. If you specify multiple `-ip-repo` options, they are searched in the order listed on the sdscc command line.

File Organization

The SDSoC compilers (`sdscc / sds++`) and sds++ linker invoke multiple sub-executables, including GNU gcc/g++, LLVM, Clang, and VivadoHLS, during compilation, system analysis, and system generation phases, but ultimately, all ARM object code is compiled through GNU gcc/g++ toolchains. A large software project will often include many libraries that are unrelated to the hardware acceleration network that SDSoC layers on top of a platform.

The only source files that must be compiled with `sdscc/sds++` are those containing code that

- defines a hardware function,
- calls a hardware function, or
- uses sds_lib functions, e.g., to allocate or mmap buffers sent to hardware functions.

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

Note:

- You must always use `sds++` for final linking because the sds linker first processes sds-compiled `.o` files before calling the GNU compiler and linker to generate code for the ARM CPUs.
- If SDSoC is unable to infer buffer allocation properties for an argument passed to a hardware function, e.g., due to pointer aliasing, it will default to interpreting as paged, cacheable memory. The SDSoC `mem_attribute` pragma must be used when the default is incorrect.

The SDSoC compilers (`sdscc / sds++`) require the user to specify via command line `-sds-hw` option each top-level function that is mapped to hardware.

In addition,

- Each top-level hardware function must reside in a separate file. This function can invoke sub-functions that reside in different files and sub-function definitions can be shared by multiple top-level hardware functions. Vivado® HLS in-lines rather than invokes shared sub-functions across different accelerators. These sub-functions should generally be declared file static.
- Code that calls a hardware function must be in a different file from the file that contains the accelerated function.

Makefile Guidelines

In addition to the integrated development environment, SDSoC provides a command line interface for compilation and linking, and the user can incorporate this command line into make files according to local customs.

The **`Makefiles`** provided with SDSoC sample designs consolidate all hardware function options to the compiler and linker into a single command line that is interpreted in the context of the `-c` option for compilation, and linking in absence of the `-c` option. This practice is not required, but has the benefit of preserving overall control structure and dependencies within a makefile without requiring change to the makefile actions for files containing a hardware function.

- You can define make variables to capture the entire SDSoC command line, e.g.

  ```
  CC = sds++ ${SDSFLAGS}
  ```

  for C++ files, invoking `sdscc` for C files. In this way, all SDSoC options are consolidated in the `${CC}` variable. Define the platform and target OS once in the variable.

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

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

  For the list of the SDSoC compiler and linker options, please use `sdscc --help`.

General C/C++ Guidelines

- Floating point arithmetic expressions and functions that are compiled to hardware using Vivado® HLS are not guaranteed to be bit accurate to ARM-compiled C/C++ code. You may encounter HW-SW implementation differences, especially for sub-normals. Consult the Vivado® HLS documentation for more information on floating point usage.
- Make sure that all floating-point constants are explicitly defined, e.g., "0.5f" instead of "0.5".
- Use contiguous single dimensional arrays when possible for best DMA data transfer efficiency. SDSoC will automatically invoke very efficient data movers when it infers contiguous physical allocation.
- Pre-defined macros enable the user to guard code with `#ifdef` and `#ifndef` preprocessor statements; the macro names begin and end with two underscore characters '_'.
  - The `__SDSCC__` macro is defined whenever sdscc or sds++ is used to compile source files, and can be used to guard code dependent on whether it is compiled by sdscc/sds++ or not, for example a GNU host compiler.

- When sdscc or sds++ compiles source files targeted for hardware acceleration using Vivado HLS, the `__SDSVHLS__` macro is also defined, and can be used to guard code dependent on whether high level synthesis is run or not.
- For examples, see the chapter SDSCC/SDS++ Compiler and Linker Options.

Vivado® HLS Hardware Function Guidelines

- A top-level hardware function name cannot be overloaded, i.e., the function name must be unique in the entire program.
- Do not refer to global variables within a hardware function or any of its sub-functions when this variable is also used in other functions running software. Otherwise, Vivado HLS will be forced to generate new ports that are not part of the function call interface.
- A hardware function's non-void return type must be a scalar type that fits in a 32-bit container.
- A hardware function must have at least one argument.
- An HLS hardware function with arguments that map onto an AXI memory-mapped master interface must have a non-void return type or include at least one output scalar argument.
- The type of a hardware function argument must resolve to a C99 basic arithmetic type, a pointer to, array of, or a struct whose members flatten to a C99 basic arithmetic type (hierarchical structs are supported). Scalar arguments must fit in a 32-bit container.
- An array argument to a hardware function argument must be either an input or an output (but not both), unless it is implemented in hardware with an AXI memory-mapped master interface.
- Vivado® HLS supports a wide array of interface types onto which array arguments can be mapped. SDSoC supports the following interface types:
  - **RAM** – using `#pragma HLS INTERFACE bram port=argument` in the accelerator function implementation. SDSoC will automatically map onto a packetized AXI-Stream channel compatible with the DMA protocol, with optional multi-buffering at the accelerator.

    The example `<sdsoc_install_dir>/samples/mmult_hls_bram` demonstrates how to use HLS bram interfaces in SDSoC.

IMPORTANT: *an argument that maps to a RAM interface cannot be directly connected to an argument that maps to an explicit axis interface. This restriction will be removed in a future SDSoC release.*

  - **FIFO** – using `#pragma HLS INTERFACE ap_fifo port=argument_name` in the accelerator function implementation. SDSoC will automatically map onto a packetized AXI-Stream channel compatible with the DMA protocol.

The example `<sdsoc_install_dir>/samples/mmult_hls_ap_fifo` demonstrates how to use HLS ap_fifo interfaces in SDSoC.

IMPORTANT: *an argument that maps to a RAM interface cannot be directly connected to an argument that maps to an explicit axis interface. This restriction will be removed in a future SDSoC release.*

- **SCALAR** – SDSoC will automatically map arguments with basic arithmetic types onto a register accessible over an AXI-Lite interface. SDSoC will FIFO the register to support task barrier synchronization and multiple in-flight task invocations. A hardware function cannot contain scalar register mapped and explicit axilite mapped arguments.
- **AXI-Lite** – using `#pragma HLS INTERFACE s_axilite port=argument` in the hardware function. Also requires `#pragma HLS INTERFACE s_axilite port=return` to generate a control interface in HLS. There will be no FIFOs on the command interface or on scalar arguments.

  A hardware function can have only one explicit axilite interface; you must bundle all ports, including ap_control, into a single axilite interface.

  The example `<sdsoc_install_dir>/samples/mmult_hls_aximm` demonstrates how to use HLS AXI-lite interfaces in SDSoC.
- **AXI-memory mapped (AXI-MM) master** – Using the VHLS pragma `#pragma HLS INTERFACE m_axi port=argument` to pass physical addresses over AXI-Lite. In this mode, the hardware function acts as its own data mover.

  When a hardware function maps an argument onto an AXI-MM master, it must also include an output scalar argument or a return value.

  The example `<sdsoc_install_dir>/samples/mmult_hls_aximm` demonstrates how to use HLS AXI-MM interfaces in SDSoC.
- **AXI4-stream** – using `#pragma HLS INTERFACE axis port=argument` in the hardware function. SDSoC supports direct connections between hardware functions with commensurate AXI-S interfaces.

  The example `<sdsoc_install_dir>/samples/mmult_hls_axis` demonstrates how to use HLS AXI4-stream interfaces in SDSoC.

IMPORTANT: *data transport using a DMA datamover requires AXI stream TLAST, TKKEP side band signals, which must be explicitly coded within HLS code.*

- A hardware function that SDSoC compiles using Vivado® HLS must have no more than eight input array arguments and eight output array arguments that map onto BRAM or FIFO interfaces; additional array arguments can be mapped onto HLS AXI4-stream or AXI-memory mapped master interfaces.

- An HLS hardware function can have up to 8 inout scalars, no more than 16 input scalars or 16 output scalars, with no more than 24 scalars total, or must explicitly map all scalars to an HLS-generated AXI4-Lite interface using HLS pragmas.

  Note: Several of these HLS guidelines are due to limitations of the `axis_accelerator_adapter` IP block that SDSoC automatically inserts for stream adapters, BRAM multi-buffering, and FIFOing of commands and scalars. This IP block in return supports task pipelining and very efficient hardware-based task control.

Hardware Function Calling Code Guidelines

- SDSoC-generated stub functions transfer the exact number of bytes according the compile-time determinable array bound of the corresponding argument in the hardware function declaration. If a hardware function admits a variable payload size, e.g., specified by a C-compilable arithmetic expression of scalar arguments to the function, you can direct SDSoC to generate a stub to transfer a payload size defined by an arithmetic expression via

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

  Be aware that mismatches between intended and actual data transfer sizes can cause system hangs typically resolved only through laborious hardware debugging.

- Align arrays transferred by DMA on cache-line boundaries (L1 and L2). Use the `sds_alloc` API provided with SDSoC or `posix_memalign()` instead of `malloc()` to allocate these arrays SDSoC will automatically ensure cache alignment when you use

  ```
  #pragma SDS data data_mover(<arg>:AXIDMA_<dmtype>)
  ```

- Align arrays to page boundaries to minimize the number of pages transferred with scatter-gather DMA, e.g., for arrays allocated with `malloc`.

- If the hardware function uses an AXI Memory-mapped Master interface to directly access the same memory as the processor, or if you use pragmas to specify Simple-DMA or 2D-DMA datamovers, the corresponding buffers/arrays must be allocated using sds_alloc(). 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.

Typical Errors in the SDSoC Flows

Three classes of issues are typically encountered in the SDSoC flow:

- compile/link time errors
- run-time errors
- performance issues

Compile/link time errors can be the result of typical software syntax errors caught by software compilers, or errors specific to the SDSoC flow, such as the design being too large to fit on the target platform. Run time errors can be the result of typical software issues such as null-pointer access, or SDSoC specific issues such as incorrect data being transferred to/from accelerators. Performance issues are related to the choice of the algorithms used for acceleration, the time taken for transferring the data to/from the accelerator, and the actual speed at which the accelerators and the data motion network operate.

Compile/Link Time Errors

Typical compile/link time errors are indicated by an error message such as when running **`make`**. To probe further, please look at the `log` files and `rpt` files in the `_sds/reports` sub directory created by SDSoC in the build directory. The last log file generated usually shows the cause of the error, such as a syntax error in the corresponding input file, or some error generated by the tool chain while synthesizing the accelerator hardware or the datamotion network.

Some tips for dealing with SDSoC specific errors.

- Tool errors reported by tools in the SDSoC chain.
    - Check whether the corresponding code adheres to the coding guidelines specified earlier.
    - Check the syntax of pragmas.
    - Check for typos in pragmas that might prevent them from being applied to the correct function.
- Vivado HLS cannot meet timing requirement.
    - Select a slower clock frequency for the accelerator (sdscc/sds++ command line parameter).
    - Modify the code structure to allow Vivado HLS to generate a faster implementation. See Vivado HLS documents for more information on how to do this.
- Vivado design tools cannot meet timing.

  - Select a slower clock frequency for the datamotion network or accelerator, or both (sdscc/sds++ command line parameters).
  - Try to synthesize the HLS block to a higher clock frequency so that the synthesis/implementation tools have a bigger margin.
  - Modify the C/C++ code passed to HLS, or add more HLS directives to make the HLS block go faster
  - Reduce the size of the design in case the resource usage (see the `vivado` report in `_sds/ipi/*.log` and other log files in the sub-directories there) exceeds 80% or so. See the next item for ways to reduce the design size.

- Design too large to fit.
  - Try to reduce the number of accelerated functions.
  - Change the coding style for an accelerator function to produce a more compact accelerator. See the Vivado HLS documents for more information on how to do this.
  - Modify pragmas and coding styles (pipelining) that cause multiple instances of accelerators to be created.
  - Use pragmas to select smaller datamovers such as AXIFIFO instead of AXIDMA_SG.
  - Structure the accelerator to have fewer input and output parameters/arguments, especially in cases where the inputs/outputs are continuous `stream(ap_fifo)` types which prevent sharing of datamover hardware.

Runtime Errors

Software developers use debuggers to catch runtime errors. Programs compiled using sdscc/sds++ can be debugged using the standard debuggers supplied with the SDSoC GUI or Xilinx SDK.

Typical runtime errors are incorrect results, premature program exit, and program “hang”. The first two kinds of errors are familiar to C/C++ programmers, and can be debugged by stepping through the code using a debugger.

Program “hang” is a new type of runtime error that is caused by errors in specifying the amount of data to be transferred in streaming connections. SDSoC allows the user to specify streaming interfaces to hardware functions that are synthesized through Vivado HLS. In addition to this, C-callable hardware functions in pre-built libraries could have streaming hardware interfaces. A program “hang” happens when the consumer of a stream is waiting for more data from the producer but the producer has stopped sending data. Consider the following code fragment that results in streaming input/output from a hardware function.

```
void f1(int in_a[20], int out_b[20]) {
#pragma HLS INTERFACE ap_fifo port=in_a
#pragma HLS INTERFACE ap_fifo port=out_b
    int i;
    for (i=0; i < 19; i++) {
         out_b[i] = in_a[i];
    }
}
```

Notice that the loop reads the in_a stream 19 times but the size of in_a[] is 20, so the caller of f1 would wait forever (or "hang") if it waited for f1 to consume all the data that was streamed to it. Similarly, the caller would wait forever if it waited for f1 to send 20 int values because f1 sends only 19. Program "hang" can be detected by instrumenting the code to flag streaming access errors such as non-sequential access or wrong access_count. Streaming access issues are typically flagged as "improper streaming access" warnings in the log file, and it is left to the user to determine if these are actual errors.

Other sources of "hang" or bad results:

- Placement of `wait()` statements – Improper placement of wait statements could result in:
    - Software reads data before the hardware accelerator has written the value.
    - System hangs because a blocking `wait()` is called before a related accelerator is started.
- Incorrect usage of memory consistency pragmas.

Using Debuggers

SDSoC allows projects to be created and debugged using the SDSoC IDE. Projects can also be created outside the SDSoC IDE (user defined make files) and debugged either on the command line or using the SDSoC IDE.

Debugging SDSoC Linux Applications

If you create a Linux application project using the SDSoC IDE, you can debug your application as follows:

1. Select the Debug build-configuration as the active build configuration and build your project
2. Copy the sd_card image generated in the previous step to an SD card, and boot your board with it.
3. Make sure your board is connected to the network, and note its IP address
4. Select the Debug as option to create a new debug-configuration, and enter the IP address for the board
5. You will now switch to the SDSoC debug perspective which allows you to start, stop, step, set breakpoints, examine variables and memory, and perform various other debug operations.

Debugging SDSoC Standalone Applications

If you create a Standalone (Bare-metal) application project using the SDSoC IDE, you can debug your application as follows:

1. Select the Debug build-configuration as the active build configuration and build your project
2. Make sure your board is connected to your host using the Debug connector.
3. Select the Debug as option to create a new debug-configuration
4. You will now switch to the SDSoC debug perspective which allows you to start, stop, step, set breakpoints, examine variables and memory, and perform various other debug operations.

Debugging SDSoC FreeRTOS Applications

If you create a FreeRTOS application project using the SDSoC IDE, you can debug your application using the same steps as a Standalone (Bare-metal) application project.

Debugging user-makefile SDSoC linux applications from the command line

Software applications running on a Linux-ARM platform, and created outside the SDSoC IDE can be debugged using gdbserver running on the ARM platform, and arm-xilinx-linux-gnueabi-gdb running on the host platform, with an Ethernet link to the ARM platform. Details are provided in the SDSoC documentation but a quick summary is shown below.

Modify the make files to compile and link with the `-g` option for debugging.

1. Build the application using the modified make files.
2. Make sure the target board can be reached from the host over Ethernet/TCP/IP.
3. Start the application on the target board by running.

   ```
   gdbserver :1234 <application name>
   ```

4. On the host side, run arm-xilinx-linux-gnueabi-gdb and at the `gdb` prompt, type the following to get started
   a. Target remote <IP address of target board>:1234
   b. Break main
   c. Continue
   d. <set other breakpoints, continue, step, examine values, etc.>

Debugging User-Makefile SDSoC Linux Applications Using the SDSoC IDE

You can use the SDSoC IDE to debug linux and standalone applications even if they are not built as SDSoC IDE managed-make projects. For debugging SDSoC Linux applications, with SDSoC, do the following:

Modify the make files to compile and link with the –g option for debugging.

1. Build the application using the modified make files.
2. Make sure the target board can be reached from the host over Ethernet/TCP/IP.
3. Open the SDSoC IDE and select File->New->Project->Xilinx->Hardware Platform Specification and click next
4. Type in a project name and then click on the browse button for the target hardware specification, and locate the exported hw platform for this design (typically found in _sds/ipi/*.sdk/SDK/SDK_Export/hw/*.xml)
5. Select File->New->Application Project and type in a dummy project name targeting the hardware platform from Steps 3,4 and select Finish
   - The dummy project should be an empty C/C++ linux app
6. Select Run->Debug Configurations from the SDSoC menu
7. Double click on the Xilinx C/C++ application (System Debugger) entry on the left hand panel to create a new debug configuration
8. In the "Target Setup" tab of the right hand panel, select Debug Type = Linux Application Debug, and then enter the IP address of the linux target in the "Host Name" textbox (leave the Port textbox at the default value of 1534)
9. In the "Application" tab of the right hand panel, go down to the "Project" line and select the dummy project created in step 7 for the application project
10. Then got to the browse to the "Application: Local File Path" line and click on "Browse" to locate the elf file for the SDSoC application
11. For the Remote File Path: type in /mnt/<app_name>.elf which is the name of the application elf file in the target linux system
12. Un-check the Auto-attach Process Children box
13. Then click on the "Source" tab to select the source directories for the application
    - First click on the "add" button and select "File System Directory" and click OK
    - Browse to the source directory for the application
    - If there are additional source directories click the "add" button as before, and add each directory
    - Finally click the "add" button to add the sources generated by SDSoC -> this is the _sds/swstubs directory

    Note that the generated sources directory is added last to make it the first directory in the source search path

14. If the compilation was done on a different machine than the one running SDSoC, go to the Path Map tab, and click the Add button to map directory prefixes of file names at compile time to the prefixes for the same directories as seen by SDSoC
15. Click on Apply, and then the Debug button

    The debug perspective opens up, allowing you to debug with SDSoC, as you would debug a standard C/C++ application. You may have to click the Resume button a couple of times before you see the program stopped at main().

For more info on how to debug with SDSoC refer to its user manual.

For debugging standalone(bare-metal) SDSoC applications using the SDSoC IDE, follow the following steps:

1. Compile and build the SDSoC application with the –g flag turned on (instead of –O3) using the standard makefile to produce an sd-card image
2. Copy the BOOT.BIN file from the previous step onto an SD card, and boot the board with this sd card. Make sure the board is in sd-boot mode and connected to the host
3. Open the SDSoC IDE and select File->New->Project->Xilinx->Hardware Platform Specification and click next
4. Type in a project name and then click on the browse button for the target hardware specification, and locate the exported hw platform for this design (typically found in _sds/ipi/*.sdk/SDK/SDK_Export/hw/*.xml)
5. Select File->New->Application Project and type in a dummy project name targeting the hardware platform from Step 1,2 and select Finish
   - The dummy project should be an empty C++ standalone app
6. Select Run->Debug Configurations from the IDE menu
7. Double click on the Xilinx C/C++ application (GDB) entry on the left hand panel to create a new debug configuration
8. In the "Main" tab of the right hand panel, go down to the "Project" line and select the dummy project created in step 7 for the application project
9. Go to the "C/C++ Application" line and click on "Browse" to locate the elf file for the SDSoC application
10. Click/select the "Disable Auto build" button
11. Then click on the "Source" tab to select the source directories for the application
    a. First click on the "add" button and select "File System Directory" and click OK
    b. Browse to the source directory for the application
    c. If there are additional source directories click the "add" button as before, and add each directory

d. Finally click the "add" button to add the sources generated by SDSoC -> this is the _sds/swstubs directory

Note that the generated sources directory is added last to make it the first directory in the source search path

12. Go to STDIO Connection and select "Connect STDIO to Console", and then select the correct COM port and set the baud rate to 115200
13. Click Apply
14. Click on the Device Initialization tab and make sure the Reset Type is "Reset Processor Only", the "Do not download" and "verify ELF" boxes are clear, and the path to the initialization file is set to the ps_init.tcl file in the hardware platform
15. Click on Debug button

    The debug perspective opens up, allowing you to debug with SDSoC, as you would debug a standard C/C++ application. You may have to click the Resume button a couple of times before you see the program stopped at main().

For more info on how to debug with the SDSoC IDE refer to the chapter SDSoC GUI in the SDSoC user manual.

Peeking and Poking IP Registers in the Physical Address Space

Two small executables called `mrd` and `mwr` are available for users to peek and poke registers in the PL hardware. These executables are invoked with the physical address to be accessed.

For example:

```
mrd 0x80000000 10
```

Reads ten 4-byte values starting at physical address 0x80000000 and prints them on the screen.

```
mwr 0x80000000 20
```

Writes the value 20 to the address 0x8000000.

These executables can be used to monitor and change the state of addressable registers in the accelerator, and in other IP generated by SDSoC.

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

Debugging Performance Issues

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

Estimate the actual hardware acceleration time by looking at the latency numbers in the Vivado HLS report files (`_sds/vhls/…/*.rpt`). Latency of X accelerator clock cycles = X * (processor_clock_freq/accelerator_clock_freq) processor clock cycles. Compare this with the time spent on the actual function call to determine the data transfer overhead.

For best performance improvement, the time required for executing the accelerated function must be much smaller than the time required for executing the original software function. If this is not true, try to run the accelerator at a higher frequency by selecting a different `clkid` on the sdscc/sds++ command line. If that does not work, try to determine whether the data transfer overhead is a significant part of the accelerated function execution time, and reduce the data transfer overhead. Note that the default clkid is 100 Mhz for all platforms. More details about the clkid values for the given platform can be obtained by running "sdscc –sds-pf-info <platform name>"

If the data transfer overhead is large, the following changes might help:

- Move more code into the accelerated function so that the computation time increases, and the ratio of computation to data transfer time is improved.
- Reduce the amount of data to be transferred by modifying the code or using pragmas to transfer only the required data.

Overview

This chapter describes the pragmas by SDSoC. These pragmas can be inserted into the C/C++ source code, to help SDSoC generate the data motion network hardware and software stub files.

Asynchronous Function Execution and Multiple Accelerator Instances

The syntax of this pragma is:

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

This pragma must be specified in the caller code immediately preceding a function call.

Notes:

- This pragma applies to the immediately following call to an accelerator function.
- The ID must be a compile time constant unsigned integer, and represents a unique identifier for the hardware accelerator. i.e. Different *ID* for the same accelerator function results in different accelerator hardware instance.
- When async pragma is used, SDSoC will not generate an sds_wait() for the immediately followed caller. It is the user's responsibility to write sds_wait(ID); or #pragma SDS wait(ID) explicitly in the source code. #pragma SDS wait(ID) is recommended because the code can be compiled by either gcc or SDSoC.

Partition Specification

The syntax of this pragma is:

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

This pragma is optional. It can be used to specify which partition/bitstream the immediate following caller to an accelerator function should be implemented in.

Notes:

- By default (no pragma), the hardware accelerator will be implemented in partition 0.
- ID must be a positive integer.
- The following example shows an example of using this pragma:

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

In this example, accelerator function "foo" has no partition pragma, so it will be implemented in partition/bitstream 0, the first call to "bar" will be implemented in partition/bitstream 1 and the second "bar" will be implemented in partition/bitstream 2.

16. A complete example showing the usage of this pragma can be found in "samples/file_io_manr_sobel_partitions".

Data Transfer Size

The syntax for this pragma is:

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

Note: Future releases of SDSoC will replace this pragma by OpenACC `copy_in`, `copy_out`, or `copy_inout` pragmas to specify the data transfer direction with regard to the accelerator function.

This pragma must be specified in the header file that contains the accelerator function prototype, immediately preceding the function declaration.

Some notes about the syntax:

- For a multi-dimensional array, each dimension should be specified. For example, for a 2-dimensional array, use *ArrayName*[*offset_dim1*:*length_dim1*][*offset_dim2*:*length2_dim2*]
- Multiple arrays can be specified in the same pragma, separated by a comma(,). For example: copy(*ArrayName1*[*offset1*:*length1*], ArrayName2[*offset2*:*length2*])
- ArrayName must be one of the formal parameters of the function defintion, i.e., not from the prototype (where parameter names are optional) but from the function definition.
- *offset* is the number of elements from the first element in the corresponding dimension. It must be a compile-time constant. This is currently ignored.
- *length* is the number of elements transferred for that dimension. It can be an arbitrary expression as long as the expression can be resolved at runtime inside the function. For example:

```
#pragma SDS data copy(InData[0:num_rows + 3*num_coeffs_active + L*(P+1)])
#pragma SDS data copy(OutData[0:2*(L-M-R+2) + 4*num_coeffs_active*(1+num_rows)])
void evw_accelerator (uint8_t M, uint8_t R, uint8_t P, uint16_t L, uint8_t
num_coeffs_active, uint8_t num_rows, uint32_t InData[InDataLength], uint32_t
OutData[OutDataLength]);
```

This pragma specifies the data transfer size (number of array elements) for array arguments when calling an accelerated function and applies to all calls to a function. The length need not be a

constant. As shown in the example, it can be a C arithmetic expression involving other scalar parameters to the same function.

If this pragma is not specified for an array argument, SDSoC will first check the argument type. If the argument type has a compile-time array size, the compiler will use that as the data transfer size. Otherwise, SDSoC will try to analyze the transfer size based on the callers. If the analysis fails or there is inconsistency between callers about the transfer size, the compiler will generate an error so that the user can modify the source code.

Memory Attributes

CAUTION! *: The syntax and implementation of this pragma are both likely to be revised in the near future.*

The syntax for this pragma is:

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

This pragma must be specified in the header file that contains the accelerator function prototype, just before the function declaration.

Some notes about the syntax:

- ArrayName must be one of the formal arguments of the function.
- *cache* must be either **CACHEABLE** or **NON_CACHEABLE**. The default value will be set to be **CACHEABLE**.

  CACHEABLE means that the compiler must maintain cache coherency between the CPU and accelerator for the memory allocated to the array. To maintain the cache coherency, it may be necessary (*e.g.,* when using HP ports) to perform a cache flush before transferring the data to an accelerator and to perform a cache-invalidate when transferring the data from the accelerator to the memory.

  NON-CACHEABLE means that the compiler does not need to ensure the cache coherency of the specified memory. It is then the user's responsibility to do so when necessary. It gives compiler more freedom in allocating memory ports. A typical use case is in video applications where:

  - Cache flushing/invalidating for a large chunk of video data can significantly decrease the system performance
  - Software code does not read or write the video data so the cache coherency between processor and accelerator is not required.
- *Continuity* must be either PHYSICAL_CONTIGUOUS or NON_PHYSICAL_CONTIGUOUS. The default value is set to be NON_PHYSICAL_CONTIGUOUS.

PHYSICAL_CONTIGUOUS means that all memory corresponding to the associated *ArrayName* is allocated using sds_alloc, while **NON_PHYSICAL_CONTIGUOUS** means that all memory corresponding to the associated *ArrayName* is allocated using malloc. This helps the SDSoC compiler to select the optimal data mover.

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

  This pragma must be specified immediately before the function declaration. This implies that, for each formal parameter, the memory consistency of the actual argument among all the callers must be the same.

Zynq® PS-PL Interconnect

CAUTION! *: The syntax and implementation of this pragma are both likely to be revised in the near future.*

The syntax for this pragma is:

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

This pragma must be specified in the header file that contains the accelerator function prototype, just before the function declaration.

Some notes about the syntax:

- ArrayName must be one of the formal arguments of the function.
- port must be ACP or AFI or MIG. If no sys_port pragma is specified for an array argument, the PS-PL interconnect will be dependent on array's memory attribute (cacheable or non-cacheable), array size, data mover used etc. Otherwise, this pragma overrides SDSoC's PS-PL interconnect choice. MIG is valid only for the zc706_mem platform.
- Multiple arrays can be specified in one pragma, separated by a comma(,).

Hardware Buffer Depth

The syntax of this pragma is:

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

This pragma must be specified in the header file that contains the accelerator function prototype, just before the function declaration.

Some notes about the syntax:

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

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

- ArrayName must be one of the formal parameters of the function.
- BufferDepth must a compile-time constant value.
- This pragma applies only to arrays that map to BRAM or FIFO interfaces, and specifies the number of hardware buffers to allocate for the array argument, e.g., to support pipelining. For a hardware buffer the following must hold:
  - BRAM: 1 ≤ BufferDepth ≤ 4, and 2 ≤ ArraySize ≤ 16384 with data width ≤ 64
  - FIFO: 2 ≤ BufferDepth * ArraySize ≤ 16384 with data width ≤ 64

Data Mover Type

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

The syntax for this pragma is:

```
#pragma SDS data data_mover(ArrayName:DataMover)
```

This pragma must be specified in the header file that contains the accelerator function prototype, just before the function declaration.

Some notes about the syntax:

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

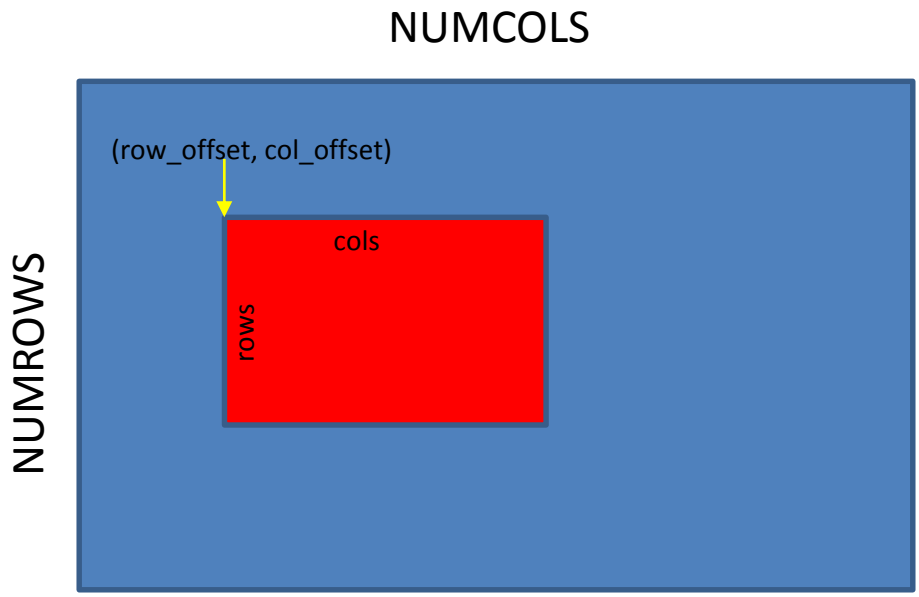
  ```
  #pragma SDS data_mover(ArrayName:DataMover, ArrayName:DataMover)
  ```

- ArrayName must be one of the formal parameters of the function.
- DataMover must be either AXIFIFO or AXIDMA_SG or AXIDMA_SIMPLE or AXIDMA_2D.
- This pragma specifies the data mover HW IP type used to transfer an array argument. Typically, the compiler will choose the type of the data automatically by analyzing the code. This pragma can be used to override the compiler inference rules.
- There are some additional requirements for using AXIDMA_SIMPLE and AXIDMA_2D. The first requirement is that the corresponding array must be allocated using **sds_alloc()**.
- For AXIDMA_2D, the pragma "SDS data dim" must be present to specify the 2D array's size of each dimension. The "SDS data copy" pragma is also needed to specify a rectangular sub-region of the

2D array to be transferred. The array's second dimension size, sub-region's offset and column size must all result in addresses aligned to 64-bit boundaries (number of bytes divisible by 8).

- In the example shown below, NUMCOLS, row_offset, col_offset and cols must be multiples of 8 (each char's bitwidth is 8) for AXIDMA_2D to work properly.

```
#pragma SDS data data_mover(y_lap_in:AXIDMA_SIMPLE, y_lap_out:AXIDMA_2D)
#pragma SDS data dim(y_lap_out[NUMROWS][NUMCOLS])
```




```
#pragma SDS data copy(y_lap_out[row_offset:rows][col_offset:cols])
void laplacian_filter(unsigned char y_lap_in[NUMROWS*NUMCOLS],
                      unsigned char y_lap_out[NUMROWS*NUMCOLS],
    int rows, int cols, int row_offset, int col_offset);
```

The SDSoC API provides functions to map memory spaces, and to wait for asynchronous accelerator calls to complete. The following is a complete list of the functions:

- `void sds_wait(unsigned int)`
  - Wait for the first accelerator in the queue identified by id, to complete.
- `void *sds_alloc( size_t size)`
  - Allocate a physically contiguous array of size bytes for DMA transfers.
- `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 `size` bytes located at physical address `physical_addr`.
  - `physical_addr`: physical address to be mapped.
  - `size`: size of physical address to be mapped.
  - `virtual_addr`:
    - If a non-null value is passed in, it is considered to be the virtual-address already mapped to the `physical_addr`, and `cf_mmap` keeps track of the mapping.
    - If a null value is passed in, `cf_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 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
  - The counter counts processor clock cycles, and wraps to 0

Note: To use these APIs 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.

Name

sdscc - SDSoC C compiler
sds++ - SDSoC C++ compiler

Synopsis

sdscc [compile_options] [link_options] [grouping_options] [performance_estimation_options] [options_passed_through_to_cross_compiler] [-sds-pf platform_name] [-sds-pf-info platform_name] [-sds-pf-list] [-verbose] [- -help] [-version] [files]

sds++ [compile_options] [link_options] [grouping_options] [options_passed_through_to_cross_compiler] [-sds-pf platform_name] [-sds-pf-info platform_name] [-sds-pf-list] [-target-os *os_name*]] [-verbose] [--help] [-version] [files]

Grouping Options

[[-sds-hw function_name file [compile_options] -sds-end]* [sdscc_options]]

Performance Estimation Options

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

Compile Options

[[-files file_list] [-hls-tcl hls_tcl_directive_file] [-clkid clock_id_number] [-vlnv v:l:n:v] [-ip-map function_map_file] [-ip-params ip_parameter_file] [-ip-repo hdl_ip_repo_path]]

Link Options

[[-apm] [-dmclkid clock_id_number] [-mno-bitstream] [-mno-boot-files] [-rebuild-hardware] [-poll-mode <0|1>] [-instrument-stub]]

Description

The sdscc/sds++ compiler compiles and links C/C++ source files, including source files with functions targeted for conversion to hardware acceleration blocks. During linking, a hardware system is produced along with software API functions used to access the hardware accelerator from the user's application.

The rest of this chapter refers to sdscc, but sdscc and sds++ usage and options are identical.

In the user **`make`** file, calls to the Xilinx Sourcery CodeBench compiler for Zynq®-7000 devices are replaced with calls to sdscc, adding options to specify the accelerator function (`-sds-hw` *function_name*) and platform (`-sds-pf` *platform_name*). Source files that do not contain accelerator functions must be compiled with sdscc if they contain calls to accelerator functions. Options not recognized by sdscc are passed to the underlying gcc tool.

When building a design that has no functions mapped to hardware, sdscc uses pre-built hardware included in a platform.

During linking, sdscc creates application ELF files before hardware implementation files. Include a dependency on hardware files as well as the ELF in the **`make`** file to properly capture the "done" state of the application.

Log files can be found in the folder `_sds/reports`.

Options

The command line options accepted by sdscc include the following. Options not used by sdscc are passed through to the underlying compiler tool.

General

-sds-pf platform_name

Specify the target platform for the system. The platform defines hardware and software system requirements and determines which boot files are required, for example, the Linux kernel and the device tree corresponding for the system. Use this option when compiling accelerator source files and when linking the ELF. Use the –sds-pf-list option to list available platforms and their features. Available platforms at this time include zc702, zc702_hdmi, zc706, microzed and zed. The *platform_name* can be a simple identifier referencing a platform in the SDSoC software installation (default usage), or a path to a folder outside of the SDSoC software installation containing platform files with the last component of the path matching the actual platform name.

-sds-pf-info *platform_name*

Display general information about a platform and exit (no other options are specified). Use the –sds-pf-list option to list available platforms.

-sds-pf-list

Display a list of available platforms and exit (no other options are specified), for example *sdscc –sds-pf-list*. At this time, supported platforms include zc702, zc702_hdmi, zc706, microzed, and zed.

-target-os <*os_name*>

The –target-os option specifies the target operating system, and the selected OS determines the compiler toolchain used, plus include file and library paths added by sdscc. The option –target-os <*os_name*> selects Linux when set to linux (default if the –target-os option is not specified on the command line), bare-metal when set to standalone, or FreeRTOS when set to freertos.

-verbose

Print verbose output to STDOUT.

-version

Print the sdscc version information to STDOUT.

--help

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

Grouping Options

Grouping options provide a means of collecting and applying sdscc options to simplify command line invocation. The simple make file fragment below illustrates the use of –sds-hw block to localize changes required for SDSoC flows:

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

CROSS_COMPILE = arm-xilinx-linux-gnueabi-
AR = ${CROSS_COMPILE}ar
LD = ${CROSS_COMPILE}ld
#CC = ${CROSS_COMPILE}g++
PLATFORM = zc702

SDSFLAGS = -sds-pf ${PLATFORM} \
        -sds-hw add add.cpp -clkid 1 -sds-end \
        -dmclkid 2
CC = sds++ ${SDSFLAGS}

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

```
LDFLAGS = -g ${LDDIRS} ${LDLIBS}

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

.PHONY: all

all: ${EXECUTABLE}

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

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

-sds-hw function_name file [compile_options] –sds-end

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

[sdscc_options]

An sdscc command line may include options outside of –sds-hw blocks. The options apply if applicable, for example the platform option –sds-pf is always applied, but the data motion clock ID option –dmclkid only applies when linking the ELF and creating the bitstream.

Performance Estimation Flow Options

Performance estimation flows enable you to obtain the estimated improvement for an application with hardware-accelerated functions versus software-only implementation.

As a pre-pass to gather data about software run time, build an instrumented application using the –perf-funcs option to specify functions to profile and the –perf-root to specify the root function encompassing calls to the profiled functions. Run the instrumented application on the target to generate a sw_perf_data.xml file containing performance data for the run.

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

In addition to these performance estimation options, specify sdscc options normally used to build an application with hardware accelerated functions.

-perf-funcs function_name_list

Specify a comma separated list of all functions to be profiled in the instrumented software application.

-perf-root function_name

Specify the root function encompassing all calls to the profiled functions. This is typically the function main.

-perf-est *data_file*

Specify the file contain run time data generated by the instrumented software application when run on the target. Estimate performance gains for hardware accelerated functions. The default name for this file is sw_perf_data.xml

-perf-est-hw-only

Run the estimation flow without running the pre-pass to collect software run data. Using this option provides hardware latency and resource estimates without providing a comparison against baseline.

CAUTION! *: After running the sd_card image on the board for collecting profile data, please run cd /; sync; umount /mnt; This ensures that the sw_perf_data.xml file is written out to the sd card*

Compile Options and Macros

-files file_list

Specify a list of one or more HLS files to compile if required, in addition to the HLS source file containing the function specified with the –hw option. The file_list contains one or more source file names separated by commas but with no white space characters. If multiple –files options are specified, this is equivalent to specifying a single –files option with all of the source file names listed. If the file contains source code that is not used by HLS and is required to link the ELF file, the file must be compiled separately to create an object file (.o) and specified as an object linked in the ELF.

-hls-tcl hls_tcl_directive_file

When using the Vivado HLS tool to synthesize the hardware accelerator, source the specified TCL file containing HLS directives (optional – for advanced users). During HLS synthesis, sdscc creates a run.tcl file used to drive the Vivado HLS tool and in this TCL file, the following commands are inserted:

```
# synthesis directives
create_clock -period <clock_period>
config_rtl -reset_level low
# end synthesis directives
```

If the –hls-tcl option is used, the commands above are replaced by the line below that sources the user TCL file

```
# user-defined synthesis directives
source <hls_tcl_directive_file>
# end synthesis directives"
```

The user must ensure the specified TCL file contains commands that result in a functionally correct run.tcl file. Note also that the –hls-tcl option is used, SDSoC will not insert interface pragmas in the C/C++ code.

-clkid *n*

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

zc702 platform

0 – 166 MHz

1 – 142 MHz

2 – 100 MHz

3 – 200 MHz

zc702_hdmi platform

1 – 142 MHz

2 – 100 MHz

3 - 166 MHz

zc706 platform

0 – 166 MHz

1 – 142 MHz

2 – 100 MHz

3 – 200 MHz

zed and microzed platforms

0 – 166 MHz

1 – 142 MHz

2 – 100 MHz

3 – 200 MHz

zybo platform

0 – 25 MHz

1 – 100 MHz

2 – 125 MHz

3 – 50 MHz

-vlnv v:l:n:v

When instantiating an IP core, specify the IP type and version using this option, for example –vlnv Xilinx.com:ip:fir_compiler:7.1

-ip-map function_map_file

When instantiating an IP core, use the specified file for the IP accelerator function map. Requires the –vlnv option.

-ip-params ip_parameter_file

When instantiating an IP core, use the specified file for IP parameter settings. Requires the –vlnv option.

-ip-repo hdl_ip_repo_path

When instantiating an IP core, repositories included with the Vivado and SDSoC tools are searched by default. The –ip-repo can be used to add a single repository path to the search path, with multiple –ip-repo options used when more than one path must be added. Requires the –vlnv option.

Pre-defined macros enable the user to guard code with `#ifdef` and `#ifndef` preprocessor statements; the macro names begin and end with two underscore characters '_'. The `__SDSCC__` macro is defined whenever sdscc or sds++ is used to compile source files, and can be used to guard code dependent on whether it is compiled by sdscc/sds++ or not, for example a GNU host compiler. When sdscc or sds++ compiles source files targeted for hardware acceleration using Vivado HLS, the `__SDSVHLS__` macro is also defined, and can be used to guard code dependent on whether high level synthesis is run or not.

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

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

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

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

Link Options

-apm

Insert AXI Performance Monitor (APM) on all generated hardware/software interfaces. Within the SDSoC GUI's Debug Perspective, you can activate the APM prior to running your application by clicking on the Start button within the Performance Counters View. For more information, please refer to the SDSoC GUI documentation.

-dmclkid <n>

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

zc702 platform

0 – 166 MHz

1 – 142 MHz

2 – 100 MHz

3 – 200 MHz

zc702_hdmi platform

1 – 142 MHz

2 – 100 MHz

3 - 166 MHz

zc706 platform

0 – 166 MHz

1 – 142 MHz

2 – 100 MHz

3 – 200 MHz

zed and microzed platforms

0 – 166 MHz

1 – 142 MHz

2 – 100 MHz

3 – 200 MHz

zybo platform

0 – 25 MHz

1 – 100 MHz

2 – 125 MHz

3 – 50 MHz

-mno-bitstream

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

-mno-boot-files

Do not generate the SD card image in the folder sd_card. This folder includes your application ELF and files required to boot Linux (BOOT.BIN with U-boot, uImage with the Linux kernel, devicetree.dtb and the filesystem uramdisk.image.gz). This option disables the creation of the sd_card folder when it contains files that shouldn't be deleted.

-rebuild-hardware

When building software-only designs where no functions are mapped to hardware, sdscc uses a pre-built bitstream found in the platform. Platform developers create pre-built hardware files using SDSoC, and if the platform in the installation area contains a pre-built bitstream already, use the –rebuild-hardware option to run Vivado implementation tools to create updated files.

-poll-mode <0|1>

The option –poll-mode <0|1> enables DMA polling mode when set to 1 or interrupt mode when set to 0 (default). For example, to specify DMA polling mode, add the sdscc option -poll-mode 1.

-instrument-stub

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

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

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

```
cat bin_file > /dev/xdevcfg
```

The SDSoC Eclipse/CDT-based GUI includes custom plugins to enable you to design complete hardware / software systems including Zynq-based SoCs, starting from C/C++ "application" source code. These plugins add functionality for project creation, implementation, performance estimation and debugging.

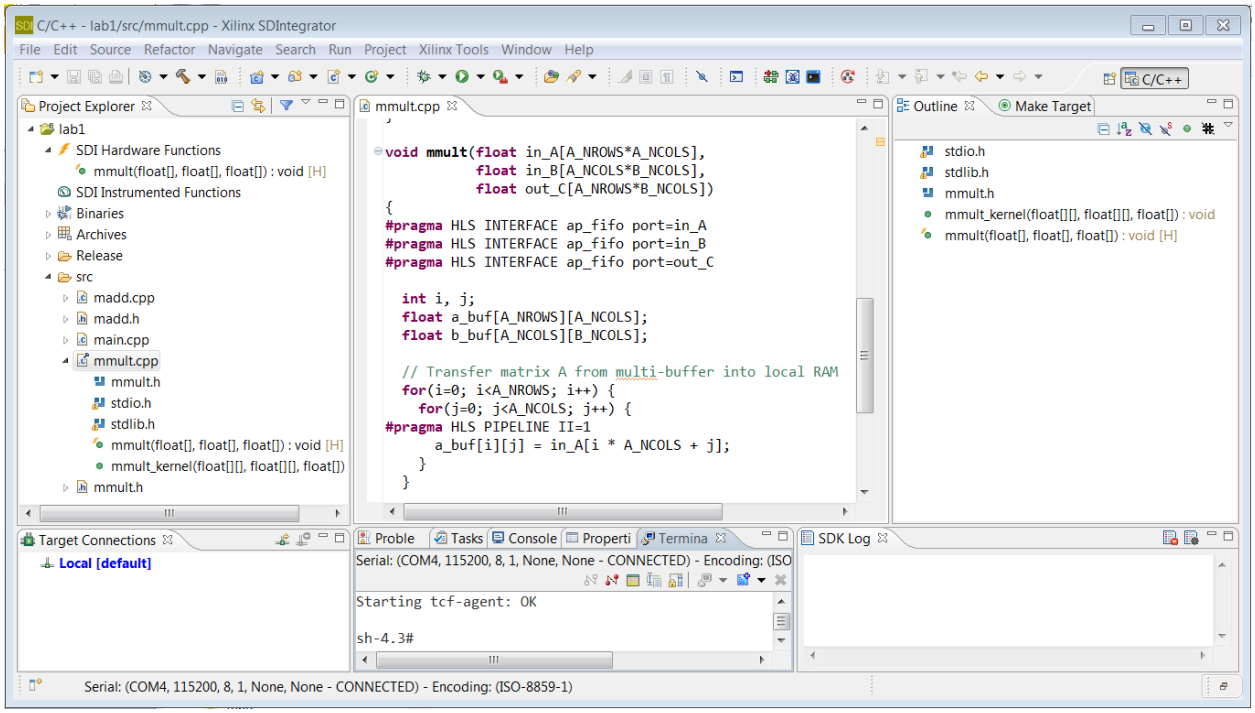


Figure 4: SDSoC GUI

To learn more about the SDSoC GUI, review the tutorials and self-guided labs [3].

Lab1 shows how to create a new SDSoC project using available templates, compile it and run the application on a board.

Lab2 demonstrates how to modify code to optimize the hardware-software system generated by SDSoC.

Lab3 shows how to retarget your design to a different platform/OS, and how to use the SDSoC GUI to download and run your hardware accelerated application on a board.

To start the gui under Linux, please run the command sdsoc.

Appendix A Additional Resources and Legal Notices

Xilinx Resources

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

Solution Centers

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

References

These documents provide supplemental material useful with this guide:

Vivado® Design Suite Documentation

References

1. *Vivado Design Suite User Guide UG896: Designing with IP, Chapter 8, Creating and Packaging IP* http://www.xilinx.com/support/documentation/sw_manuals/xilinx2013_4/ug896-vivado-ip.pdf
2. *User Guide 1146 - SDSoC Platforms and Libraries*, `<sdsoc_install_root>/docs/ug1146-sdsoc_platforms_and_libraries.pdf`.
3. *User Guide 1028 – Getting Started,* `<sdsoc_install_root>/docs/ug1028-sdsoc_getting_started.pdf`.

Please Read: Important Legal Notices

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 http://www.xilinx.com/legal.htm#tos.