SDSoC User Guide

Platforms and Libraries

UG1146 (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

Introduction

The SDSoC Design Environment is an integrated development environment for building heterogeneous embedded systems using the Zynq®-7000 All Programmable SoC. The SDSoC compiler and linker transform programs written in C/C++ into complete hardware/software systems based on command line options that specify target platform and functions within the program to compile into programmable hardware.

RECOMMENDED: *For additional information on using the SDSoC design environment, see SDSoC User Guide: Introduction to SDSoC (UG1027) [2].*

SDSoC encourages a layered design methodology, separating application-specific SoC creation from platform building. Of course, iterations can occur between platform and tailored SoC design, but in general, the skill sets and specific concerns will differ between application programmers who use SDSoC, and platform builders who use the Vivado Design Suite for hardware and PetaLinux or other development environment for building boot and OS software.

An SDSoC platform has a hardware component that provides a hardware context into which the `sdscc` compiler and linker generate hardware functions and IP-based data motion networks for communication between hardware functions and the platform. A hardware platform includes the Zynq PS, which contains dual-A9 ARM CPUs and other hard IP including a memory controller that provides access to external DDR. A platform will often include peripherals implemented in the Zynq PL, e.g., to interface with DACs, ADCs, or HDMI I/O, or to access external DDR through a MIG memory controller. A platform can also include additional functionality that is entirely independent of the logic generated by `sdscc`.

The topic of platform design is beyond the scope of this document; suffice it to say that essentially any Zynq hardware design can be used as the basis for an SDSoC platform. To the SDSoC compiler and system linker, a hardware platform provides a set of hardware interfaces (AXI, AXI-S, clocks, resets, interrupts) through which `sdscc` can establish communication channels to and between hardware functions.

An SDSoC platform also contains a software component that includes boot and run time environment, operating system and optionally, software libraries that can be linked from `sdscc` (or arm-gcc) applications. While SDSoC manages all software for hardware functions and datamovers (including drivers as needed), configuring an OS for platform hardware peripherals is the responsibility of the platform builder.

This document describes how to capture platform hardware and software interfaces and the process of creating an SDSoC compatible software library.

Platform Hardware Requirements

A Zynq hardware system must be built using the Vivado Design Suite, and have a representation as an IP Integrator block diagram. There are several rules the Vivado design must observe:

- Every platform must contain a Zynq® `processing_system7` block from the Vivado IP catalog. Every *unused* AXI port on the processing_system7 automatically becomes part of the exported SDSoC platform interface. If you wish to share a PS AXI port between SDSoC and the platform logic, you must export an unused AXI master or slave off of the AXI Interconnect IP block connected to the corresponding `processing_system7` port. If a platform exports an unterminated AXI port on an interconnect, every SDSoC application will be required to use this port; SDSoC will automatically terminate AXI ports only on the `processing_system7` IP.
- An SDSoC hardware port interface consists of AXI, AXI stream (AXIS), clock, reset, and interrupt interfaces only. AXIS interfaces require TLAST, TKEEP sideband signals to comply with the Vivado IP employed by SDSoC datamovers.
- SDSoC exported clock signals must be driven by one of the `FCLK_CLKn` ports (n = 0, 1, 2, 3) of the `processing_system7` IP. A platform can contain other clock sources, but they cannot be exported as part of the SDSoC platform interface.
- SDSoC exported resets signals must be driven by a `proc_sys_reset` block from the Vivado IP catalog, synchronized to an exported SDSoC clock signal. Each clock in the platform interface must have the associated proc_sys_reset `peripheral_reset, peripheral_aresetn, and interconnect_reset` ports also in the platform interface.
- PL to PS interrupts must be exported by a Concat (`xlconcat`) block connected to the `processing_system7 IRQ_F2P` port. IP within a platform may use some of the 16 fabric interrupts, but must use the least significant bits of the IRQ_F2P port without gaps. The SDSoC platform interrupt interface consists of any remaining unused interrupts. If a platform does not use any interrupts, the xlconcat block has an unterminated input; SDSCC will automatically terminate the interrupt concat block if necessary.
- All custom IP employed within the platform must be contained within the Vivado project. References to external IP repository paths are not allowed.

Platform Software Requirements

SDSoC currently supports Linux, standalone (bare metal), and FreeRTOS operating systems running on the Zynq target. If your platform contains peripherals that require Linux kernel drivers, it is your responsibility to configure the kernel to include several SDSoC drivers.

The default target for the `sdscc` linker is an SD card image for booting the platform.

Platform Files

An SDSoC platform consists of a Vivado hardware project, an operating system, boot files, and optional software libraries. In addition, an SDSoC platform includes XML meta-data files that describe the hardware and software interfaces. A platform provider uses Vivado Design Suite and IP Integrator to create the platform hardware, and tcl commands within Vivado to tag SDSoC hardware interfaces. The SDSoC HSI utility generates the hardware meta-data file. The platform creator must also provide boot loaders and operating system required to boot the platform, and if desired, optional software libraries that can be linked by SDSoC applications. Currently, the software platform meta-data file must be written by hand.

An SDSoC platform consists of the following elements:

- Metadata Files
  - Platform hardware description file (`<platform>_hw.pfm`)
  - Platform software description file (`<platform>_sw.pfm`)
- Vivado Project
  - Sources
  - Constraints
  - IP blocks
- Software Files
  - Library header files – optional
  - Static libraries – optional
  - Linux related objects – device-tree, u-boot, Linux-kernel, ramdisk
- Pre-Built Hardware Files - optional
  - Bitstream
  - Exported hardware files for SDK
  - Pre-generated device registration and port information software files
  - Pre-generated hardware and software interface files

Platform Hardware Description File

A platform hardware description file `<platform>_hw.pfm` is an XML meta-data file that describes the hardware system to SDSoC, including available clock frequencies, interrupts, and the hardware interfaces SDSoC can use to communicate with hardware functions. You create this file by building the base platform design in Vivado IP Integrator, setting attributes on select ports in the IP Integrator block diagram using tcl APIs in the Vivado Tcl Console (or a script), exporting the design, and then invoking the SDSoC HSI utility to generate the SDSoC platform hardware description.

The SDSoC platform port interface consists of a set of 'available' unused AXI or AXI-S bus interfaces on platform IP within an IP Integrator block diagram, a set of unused interrupts on the processing_system7 IP block, clock ports, and synchronized resets provided by `proc_sys_reset` IP from the Vivado catalog.

The Zynq `processing_system7` IP block in the Vivado catalog receives special treatment; every unused AXI interface becomes part of the SDSoC interface. For all other IP in the IP Integrator system, the platform builder must explicitly set a tcl property on an AXI or AXI-S interface in order for the interface to be included in the SDSoC platform interface.

The following command is required to enable IP Integrator export of SDSoC meta-data to SDK to create an SDSoC hardware platform description.

```
set_param project.enablePlatformHandoff true
```

IMPORTANT: *SDSoC includes a Zynq version of the Vivado Design Suite 2014.4. You are free to use any version of Vivado 2014.4 to create a hardware system, but you must use the SDSoC version of Vivado.to export an SDSoC hardware platform description,*

File name and location:

```
<directory>/<platform>/<platform>_hw.pfm
```

Example:

```
platforms/zc702/zc702_hw.pfm
```

Vivado IP Integrator Tcl Commands to Specify SDSoC Hardware Platform

The following attributes can be applied within a block diagram using Vivado tcl APIs.

- SDI_PFM_CLOCK = *TRUE* | *FALSE* set to TRUE for the platform default clock port master
- SDI_PFM_CLOCK_ID = *non-negative integer* defines the clock ID for a clock port master
- SDI_PFM_UIO = *TRUE* | *FALSE* set to TRUE if the IP instance is a Linux UIO platform device
- SDI_PFM_VERSION = *string* set to desired version string
- SDI_PFM_NAME = *string* set to desired platform name
- SDI_PFM_DESCRIPTION = *string* set to desired platform description
- SDI_PFM_FCNMAPS = *string* set to a list of XML fcnmap elements for hardware functions
- *MARK_SDI = TRUE | FALSE set to TRUE for any exported platform port*

IMPORTANT: *In this release of SDSoC, the exported hardware description file* `<platform>_hw.pfm` *may be missing some required meta-data, which you must add by hand. Inspect the vivado.log file or the output of the Vivado Tcl Console for "unknown attribute" warnings, which indicate that a particular attribute is not yet supported.*

Known issues include:

- *Platform AXI stream bus interfaces containing TLAST, TKEEP sideband signals must have the* `xd:hasTlast` *attribute (with value "true") in the corresponding* `xd:busInterface` *element, but these are not automatically detected. You must add these attributes by hand. (see Chapter 5, Example 2)*

- *Platform AXI interfaces on axi_interconnects as missing the following attributes, which must be added by hand (see Chapter 5, Example 3)*
    - `xd:resetRef` *– reference to the platform reset bus interface name for the AXI interface*
    - `xd:coherent` *– required for an interconnect interface that is connected to the S_AXI_ACP, optional otherwise*
    - `xd:numIds` *– number of available IDs for this interface. Each AXI interface has a fixed number of ID bits, and each interconnect will reserve ceil(log2(#interfaces)) bits. Since platform interfaces are shared with SDSoC by cascading axi_interconnects, SDSoC requires this number to avoid overloading an interface.*
    - `xd:memport="IC"` *– tags an interface as a channel to external memory*

These issues will be addressed in a future Vivado release.

As stated above, every *unused* AXI interface on the Zynq `processing_system7` IP block is automatically included in the SDSoC interface. For any other port or bus interface on a platform IP, you must specify port inclusion with the following tcl commands.

```
set_property BD_ATTRIBUTE.MARK_SDI true [get_bd_pins <port>]
set_property BD_ATTRIBUTE.MARK_SDI true [get_bd_intf_pins <bus interface>]
```

Supported interface types are AXI, AXIS, clock, reset, and interrupt.

Clocks

You can use any clock source within the platform, but in the current release every exported SDSoC platform clock must be sourced by the Zynq `processing_system7` IP. Configure platform clock frequencies on the PS7 IP within Vivado and these will carry over to SDSoC.

The command:

```
set_property BD_ATTRIBUTE.MARK_SDI true [get_bd_pins /ps7/FCLK_CLK1]
```

declares the FCLK_CLK1 port on the /ps7 instance to be part of the SDSoC interface.

You must specify a non-negative integer valued ID for each clock with the following tcl command.

```
set_property BD_ATTRIBUTE.SDI_PFM_CLOCK_ID <id> [get_bd_pins <clock port>]
```

For example,

```
set_property BD_ATTRIBUTE.SDI_PFM_CLOCK_ID 1 [get_bd_pins /ps7/FCLK_CLK1]
```

sets the ID for FCLK_CLK1 to be 1.

Every platform must declare a default clock for SDSoC to use when the user does not specify an explicit clock. For example,

```
set_property BD_ATTRIBUTE.SDI_PFM_CLOCK TRUE [get_bd_pins /ps7/FCLK_CLK2]
```

declares the FCLK_CLK2 port on the /ps7 block as the default platform clock. If the clock ID for FCLK_CLK2 is 2, the XML element for that captures platform default clock is

```
<xd:systemClocks xd:defaultClock="2">
```

Resets

SDSoC requires a platform to synchronize resets to specific clocks using the Vivado `proc_sys_reset` IP, and to export the reset interfaces with the following commands.

```
set_property BD_ATTRIBUTE.MARK_SDI true [get_bd_pins <proc_sys_reset>/interconnect_aresetn]
set_property BD_ATTRIBUTE.MARK_SDI true [get_bd_pins <proc_sys_reset>/peripheral_aresetn]
set_property BD_ATTRIBUTE.MARK_SDI true [get_bd_pins <proc_sys_reset>/peripheral_reset]
```

Interrupts

Interrupts must be connected to the platform's PS7 block via an IP Integrator Concat block (xlconcat). If any IP within the platform includes interrupts, these must occupy the least significant bits of the concat block without gaps. You do not need to declare interrupts explicitly with tcl APIs; Vivado will automatically export available interrupts as part of the SDSoC platform interface.

Hardware Export

To Export the System through the Vivado GUI or via tcl:

```
write_hwdef  -file "[file join <platform>.sdk <platform>_wrapper.hdf]"
```

RECOMMENDED: *in Vivado 2014.4, the platform specification is not persistent in the IP Integrator block diagram. We recommend you reuse the HSI-generated journal tcl file when you need to regenerate the platform description.*

After capturing the platform hardware in Vivado IP Integrator, synthesizing and exporting the system hardware, from an SDSoC command shell use the HSI utility to generate the SDSoC hardware platform description.

```
$ hsi
%hsi open_hw_design <platform>.sdk/<platform>_wrapper.hdf
%hsi generate_target {SDI} [current_hw_design] -dir <target_directory>
%hsi quit
```

This will generate the hardware platform description file in <target_directory>, along with a journal tcl file that contains the tcl commands you invoked to declare the platform.

IMPORTANT: *if you have installed Vivado Design Suite or SDK in addition to SDSoC, you must invoke the SDSoC version of HSI to generate an SDSoC hardware platform description.*

`hsi` generates a file called `<platform>.pfm`, which you must rename to `<platform>_hw.pfm`

Platform Software Description File

In addition to the platform hardware description, you must provide a platform software description.

Boot Files

By default, SDSoC will create an SD card image to boot a board into a Linux prompt or execute a standalone program.

Describe the files using the following format:

```
<xd:bootFiles
  xd:os=”linux”
  xd:bif=”boot/linux.bif”
  xd:readme=”boot/generic.readme”
  xd:devicetree=”boot/devicetree.dtb”
  xd:linuxImage=”boot/uImage”
  xd:ramdisk=”boot/ramdisk.image.gz”/>
```

For standalone, where no OS is used, the description is:

```
<xd:bootFiles
  xd:os="standalone"
  xd:bif="boot/standalone.bif"
  xd:readme=”boot/generic.readme”
/>
```

Note that these elements refer to a Boot Image File (bif). The bif file must exist in the location specified.

An example Linux bif file has the following contents:

```
/* linux */
the_ROM_image:
{
  [bootloader]<boot/fsbl.elf>
  <bitstream>
```

```
  <boot/u-boot.elf>
}
```

SDSoS treats this bif file as a template that describes the boot image contents, replacing all text between '<' and '>' symbols to create the actual bif file it employs to create the boot image.

Paths to files generated by SDSoC will replace the following tags:

```
<bitstream>    Generated configuration bitstream
<elf>          Generated executable file
<path/to/file> Any path relative to the platform root directory
```

In the above example, the [bootloader] line points to the fsbl, where the path is relative to the current platform root directory. The second line refers to the bitstream generated by SDSoC. The third line refers to the u-boot executable to use.

An example standalone.bif file has the following contents:

```
/* standalone */
the_ROM_image:
{
  [bootloader]<boot/fsbl.elf>
  <bitstream>
  <elf>
}
```

In this example, the [bootloader] line points to the fsbl. The second line refers to the bitstream generated by SDSoC. The third line refers to the executable generated by SDSoC.

SDSoC outputs a README.txt for the SD card image using the template file specified by the xd:readme element. The specified file is read, and all text between '<' and '>' symbols will be replaced by SDSoC to create the final README.txt file included in the SD card image. SDSoC will replace the following tags in a line of text:

```
<elf>          Generated executable file
<platform>     Platform name
```

Library Files

A platform may optionally include libraries. If you describe the library files using the following format, SDSoC will automatically add the appropriate include and library paths (using the -I and -L switches) when calling the compiler.

```
<xd:libraryFiles
  xd:os="linux"
  xd:includeDir="arm-xilinx-linux-gnueabi/include"
  xd:libDir="arm-xilinx-linux-gnueabi/lib"/>
<xd:libraryFiles
  xd:os="standalone"
  xd:includeDir="arm-xilinx-eabi/include"
  xd:libDir="arm-xilinx-eabi/lib"/>
```

Description

The informal schema for `xd:libraryFiles` is:

```
<xd:libraryFiles
  xd:os              Operating system. Valid values: linux, standalone
  xd:includeDir      Include directory passed to compiler using -I. Only
                     a single of these elements is valid.
  xd:libDir          Library path passed to the linker using -L . Only a
                     single of these elements is valid.
  xd:libName         Library name passed to the linker using -l.  Only a
                     single of these elements is currently supported. If this
                     is specified, sdscc/sds++ automatically adds the -l option
                     when linking the ELF.
/>
```

Pre-Built Hardware Files

A platform may optionally include pre-built hardware files, which SDSoC will clone into a project when an application has no hardware functions, rather than rebuilding the bitstream and boot image. This provides an SDSoC very fast compiles to run an application purely in software. Even when a platform provides pre-built hardware files, a user can force the bitstream compile using the `sdscc -rebuild-hardware` option to force the creation of hardware files.

The example below describes pre-built hardware included in the ZC702 platform:

```
<xd:hardware
  xd:system="prebuilt"
  xd:bitstream="prebuilt/bitstream.bit"
  xd:export="prebuilt/export"
  xd:swcf="prebuilt/swcf"/>
```

Description

The informal schema for `xd:hardware` is:

```
<xd:hardware
  xd:system          Identifier associated with pre-defined hardware; when
                     SDSoC searches for a pre-built bitstream, it looks
                     for the keyword "prebuilt"
  xd:bitstream       Path to the bitstream.bit file for the pre-built
                     hardware
  xd:export          Path to the folder containing SDK-compatible files
                     created using the Vivado export_hardware command.
                     This folder contains the hardware handoff file
                     <platform>.hdf, e.g. zc702.hdf.
  xd:swcf            Path to the folder containing device registration and
                     port information files. Files found in this folder
                     are devreg.c, devreg.h, portinfo.c and
                     portinfo.h.
/>
```

The files described in this section can be created using SDSoC.

Testing the XML File

After putting the two platform XML files (`<platform>_hw.pfm, <platform>_sw.pfm`) in place, you can verify that SDSoC can read it correctly by executing the following command, which lists all the available platforms. If you see the platform you have created, then SDSoC has found it.

```
> sdscc -sds-pf-list
```

To display more information about your platform, you can use this command:

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

Vivado Project

SDSoC uses the platform Vivado project as a starting point to build an application-tailored SoC. The project must include an IPI Block Diagram and may contain any number of source files.

File name and location*:*

```
platforms/<platform>/vivado/<platform>.xpr
```

Example:

```
platforms/zc702_hdmi/vivado/zc702_hdmi.xpr
```

Note that you must place the complete project in the same directory as the xpr fie.

***IMPORTANT**: you cannot simply copy the files in a Vivado project; Vivado manages internal state in ways that may not be preserved through simple file copy. To make a project clonable, from within Vivado, File->Archive Project creates a zip archive. Unzip this archive file into the SDSoC platform directory where the hardware platform resides.*

Vivado requires Upgrade IP for every new version of the Vivado Design Suite. To migrate an SDSoC hardware platform, open the project in the new version of the tools, and then Upgrade all IP. Archive the project and then unzip this archive into the SDSoC platform hardware project.

If you encounter "IP Locked" errors when SDSoC invokes Vivado, it is a result of failing to make the platform clonable as described above.

Library Header Files

If the platform requires application code to `#include` platform-specific header files, these can reside in a subdirectory of the platform directory. The subdirectory is pointed to by the `xd:includeDir` attribute for the corresponding OS as described earlier.

For a given `xd:includeDir="<relative_include_path>"` in a platform xml, the location is:

```
platforms/<platform>/<relative_include_path>
```

Example:

For `xd:includeDir="arm-xilinx-linux-gnueabi/include"`:

```
platforms/zc702_hdmi/arm-xilinx-linux-gnueabi/include/zc702hdmi/hwi_export.h
```

To use the header file in application code, use the following line:

```
#include "zc702hdmi/hwi_export.h"
```

RECOMMENDED: *Note that if header files **are not** put in the standard area, users need to point to them using the `-I` switch in the SDSoC compile command. We recommend putting the files in the standard location as described in the platform xml file.*

Static Libraries

If the platform requires users to link against static libraries provided in the platform, these can reside in a subdirectory of the platform directory. The subdirectory is pointed to by the `xd:libDir` attribute for the corresponding OS as described earlier.

For a given `xd:libDir="<relative_lib_path>"` in a platform xml, the location is:

```
platforms/<platform>/<relative_lib_path>
```

Example:

For `xd:libDir="arm-xilinx-linux-gnueabi/lib"`:

```
platforms/zc702_hdmi/arm-xilinx-linux-gnueabi/lib/libzc702hdmi.a
```

To use the library file, use the following linker switch:

```
-lzc702hdmi
```

RECOMMENDED: *Note that if static libraries are not put in the standard area, users need to point to them using the `-L` switch in the SDSoC link command. We recommend putting the files in the standard location as described in the platform xml file.*

Pre-Built Hardware

As described above, a platform can optionally include pre-built configurations to be used directly when the user does not specify any hardware functions in an application. In this case, the user does not need to wait for a hardware compile of the platform itself to create a bitstream and other required files.

The pre-built hardware should reside in a subdirectory of the platform directory. Data in the subdirectory is pointed to by the `xd:bitstream`, `xd:export`, and `xd:swcf` attributes for the corresponding pre-built hardware as described earlier.

For a given `xd:bitsteam="<relative_lib_path>/bitstream.bit"` in a platform xml, the location is:

```
platforms/<platform>/<relative_lib_path>/bitstream.bit
```

For a given `xd:export="<relative_export_path>"` in a platform xml, the location is:

```
platforms/<platform>/<relative_export_path>
```

For a given `xd:swcf="<relative_swcf_path>"` in a platform xml, the location is:

```
platforms/<platform>/<relative_swcf_path>
```

Example:

For `xd:bitstream="prebuilt/bitstream.bit"`:

```
platforms/zc702/hardware/prebuilt/bitstream.bit
```

For `xd:export="prebuilt/export"`:

```
platforms/zc702/hardware/prebuilt/export
```

contains `zc702.hdf`

For `xd:swcf="prebuilt/swcf"`:

```
platforms/zc702/hardware/prebuilt/swcf
```

containing `devreg.c, devreg.h, portinfo.c and portinfo.h`

Prebuilt hardware files will automatically be employed by SDSoC when an application has no hardware functions using the usual flag:

```
-sds-pf zc702
```

To force a full Vivado bitstream and SD card image compile, use the following `sdscc` option:

```
-rebuild-hardware
```

Files used to populate the platforms/<platform>/hardware/prebuilt folder are found in the _sds folder after creating the application ELF and bitstream.

- bitstream.bit
  - File found in `_sds/p0/ipi/<platform>.runs/impl_1/bitstream.bit`
- export
  - Files found in `_sds/p0/ipi/<platform>.sdk (<platform>.hdf)`
- swcf
  - Files found in `_sds/swstubs (devreg.c, devreg.h, portinfo.c, portinfo.h)`

Linux Boot Files

SDSoC can create an SD card image to boot a board into a Linux prompt and execute the compiled applications. For this, SDSoC requires several objects as part of the platform including:

- First Stage Boot Loader (FSBL)
- U-boot
- Device Tree
- Linux Image

- Ramdisk Image

SDSoC uses the Xilinx `bootgen` utility program to combine the necessary files with the bitstream into a BOOT.BIN file in a folder called `sd_card`. The end-user copies the contents of this folder into the root of an SD card to boot the platform.

IMPORTANT: *This document does not describe the details on how to build the boot files. For detailed instructions, refer to the Xilinx Wiki at http://wiki.xilinx.com*

First Stage Boot Loader (FSBL)

The first stage boot loader is responsible for loading the bitstream and configuring the Zynq processor subsystem at boot time.

When the platform project is open in Vivado IP Integrator, click on the `File->Export->Export_to_SDK` menu options to export the Hardware to Xilinx SDK and then open up Xilinx SDK. Using this hardware platform, select the new project menu in Xilinx SDK to create a new Xilinx application, and then select the FSBL application from the list. This creates an FSBL executable.

For more detailed information on using the Xilinx SDK please see the SDK complete help system that describes concepts, tasks, and reference information.

Once the platform provider generates the FSBL through Xilinx SDK, they must copy it into a standard location for the SDSoC flow.

For SDSoC to use an FSBL, a bif file must point to it as described earlier.

```
/* linux */
the_ROM_image:
{
  [bootloader]<boot/fsbl.elf>
  <bitstream>
  <boot/u-boot.elf>
}
```

The file must reside in:

```
platforms/<platform>/boot/fsbl.elf
```

Example:

```
platforms/zc702_hdmi/boot/fsbl.elf
```

U-Boot

Das U-Boot is an open source boot loader. Follow the instructions at wiki.xilinx.com to download u-boot and configure it for your platform.

For SDSoC to use a U-Boot, a BIF file must point to it as described earlier.

```
/* linux */
the_ROM_image:
{
  [bootloader]<boot/fsbl.elf>
  <bitstream>
  <boot/u-boot.elf>
}
```

The file must reside in:

```
platforms/<platform>/boot/u-boot.elf
```

Example:

```
platforms/zc702_hdmi/boot/u-boot.elf
```

Device Tree

The Device Tree is a data structure for describing hardware so that the details do not have to be hard coded in the operating system. This data structure is passed to the operating system at boot time. Use Xilinx SDK to generate the device tree for the platform. Follow the device-tree related instructions at wiki.xilinx.com to download the devicetree generator support files, and install them for use with Xilinx SDK. There is one device tree per platform.

The file name and location are defined in the platform xml. Use the xd:devicetree attribute in an xd:bootFiles element.

Sample xml description:

```
xd:devicetree="boot/devicetree.dtb"
```

Location:

```
platforms/zc702_hdmi/boot/devicetree.dtb
```

Linux Image

A Linux image is required to boot. Xilinx provides single platform-independent prebuilt Linux image that works with all the SDSoC platforms supplied by Xilinx.

However, if you want to configure Linux for your own platform, please follow the instructions at wiki.xilinx.com to download and build the Linux kernel. Please make sure to enable the SDSoC APF drivers and the Contiguous Memory Allocator (CMA) when configuring Linux for your platform. Linux kernel build instructions for SDSoC platforms are described in

```
<sds_install_root>/<platform>/boot/how-to-build-this-linux-kernel.txt.
```

The file name and location are defined in the platform xml. Use the xd:linuxImage attribute in an xd:bootFiles element.

Sample xml description:

```
xd:linuxImage=”boot/uImage”
```

Location:

```
platforms/zc702_hdmi/boot/uImage
```

Ramdisk Image

A ramdisk image is required to boot. A single ramdisk image is included as part of the SDSoC install. If you need to modify it or create a new ramdisk, please follow the instructions at wiki.xilinx.com.

The file name and location are defined in the platform xml. Use the xd:ramdisk attribute in an xd:bootFiles element.

Sample xml description:

```
xd:ramdisk=”boot/uramdisk.image.gz”
```

Location:

```
platforms/zc702_hdmi/boot/uramdisk.image.gz
```

Standalone Boot Files

If no OS is required, the end-user can create a boot image that automatically executes the generated executable.

FSBL

Use the same FSBL executable as described in the Linux Boot Files.

Executable

For SDSoC to use an executable in a boot image, a bif file must point to it as described earlier.

```
/* standalone */
the_ROM_image:
{
  [bootloader]<boot/fsbl.elf>
  <bitstream>
  <elf>
}
```

SDSoC will automatically insert the generated bitstream and elf files.

Library Files

SDSoC software callable libraries consist of hardware, software and meta-data files. The library provider encapsulates the all the required elements into a library that can be used as any other C/C++ static library.

The following is the list of elements that are part of an SDSoC software callable library:

- Header file
  - Function prototype
- Static Library
  - Function definition
  - IP core
  - IP configuration parameters
  - Function argument mapping

Libraries as Part of an SDSoC Platform

A software callable library can provide SDSoC compiled applications access to IP blocks within a platform. In Vivado 2014.4 there is no way to bind this information into the IP Integrator block diagram, so in addition to creating a library (as discussed in the Creating a Library section below) you must also add the associated meta-data manually into the hardware platform description file described earlier in this document.

Header File

The function prototype in provides in a header file as it is done for any C/C++ static library. The header file defines the function interface. This header file is to be included in the user's source code.

Example:

```
// FILE: fir.h
#define N 256
void fir(signed char X[N], short Y[N]);
```

where array X is the filter input and Y is the filter output.

Static Library

An SDSoC static library contains several elements that allow a software function to be executed on programmable resources. These elements are described below.

Function Definition

The function interface defines an entry point into the library. This is the function (or set of functions) that can be called from the user code.

The contents of the function are not required because SDSoC will replace the function body with API calls that execute the data transfer to/from the IP block. These calls are dependent on the data motion network created by SDSoC.

Example:

```
// FILE: fir.c

#include "fir.h"
#include <stdlib.h>
#include <stdio.h>

void fir(signed char X[N], short Y[N])
{
  // SDSoC replaces function body with API calls for data transfer
}
```

Note: The API calls used by SDSoC require the use of stdlib.h and stdio.h to be included in this file.

IP Core

The IP core is a Vivado compatible HDL IP that is used as a backend for the library function or functions. This core could be located in the Vivado IP repository or in any other location. When the library is used, the corresponding IP core is instantiated in the hardware system.

If the IP core is not Vivado compatible, you must integrate it 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. In addition, the packager creates an archive zip file that contains the directory and its contents required by Vivado Design Suite.

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]

IP Configuration Parameters

Most HDL IP cores are customizable at synthesis time. This is done through IP parameters that define the IP core's behavior. SDSoC uses this information at the time the core is instantiated in a generated system. This information is captured in an XML file.

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.

Example:

```
<!-- FILE: fir.params.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<xd:component xmlns:xd="http://www.xilinx.com/xidane" 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>
```

Function Argument Map

This information describes the relationship between the C/C++ function and the HDL IP interface. SDSoC uses this information to create the code that performs the data transfers to/from the IP core and to do the appropriate connections. This mapping is captured in an XML file.

The information includes the following.

- Function name – the name of the function mapped onto a component
- Component reference – either an IP name for a C-callable library, or the platform name if the function maps onto any IP within the platform itself. Note that a platform is an interface defined for a Vivado IP Integrator system, but is not an IP. For non-platform IP, the component reference will be the IP type name, i.e., the "name" in the IP-XACT VLNV.
- 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". Currently SDSoC does not support "inout" function arguments.
- Bus interface – the name of the IP port corresponding to a function argument. For a platform component, this name is the platform interface `xd:name`, not the actual port name on the corresponding platform IP.
- 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.

IMPORTANT: *The arguments, i.e., identifiers in the XML fcnmap must be the same as they are in the function definition, and they must occur in precisely the same order.*

Example:

```
<!-- FILE: fir.fcnmap.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<xd:repository xmlns:xd="http://www.xilinx.com/xidane">
  <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:latencyEstimates xd:worst-case="20"
                         xd:average-case="20"
                         xd:best-case="20"/>
    <xd:resourceEstimates xd:BRAM="0" xd:DSP="1"
                          xd:FF="200" xd:LUT="200"/>
  </xd:fcnMap>
</xd:repository>
```

Creating a Library

Xilinx provides a utility called sdslib that allows the creation of SDSoC libraries.

```
Usage: sdslib [arguments] [options]

Arguments (mandatory):
  -lib <libname>        Library name to create or append to
  <tuple>+              A tuple is made of two items (in order):
                         1. Function
                         2. Filename
                         e.g. fir fir.c
  -vlnv <v>:<l>:<n>:<v> Use IP core specified by this vlnv
                         e.g. -vlnv xilinx.com:ip:fir_compiler:7.1
  -ip-map <file>        Use specified <file> as IP function map
  -ip-params <file>     Use specified <file> as IP parameters

Options:
  -ip-repo <path>       Add HDL IP repository search path
  -os <name>            Specify target Operating System
                         linux (default)
                         standalone (bare-metal)

  --help                Display this information

As an example, to create an SDSoC library for a fir filter IP core, call:

> sdslib -lib libfir.a \
    fir fir.c \
    fir_reload fir_reload.c \
    fir_config fir_config.c \
    -vlnv xilinx.com:ip:fir_compiler:7.1 \
    -ip-map fir_compiler.fcnmap.xml \
    -ip-params fir_compiler.params.xml
```

In the above example, sdslib will use the functions `fir` (in file `fir.c`), `fir_reload` (in file `fir_reload.c`) and `fir_config` (in file `fir_config.c`) and archive them into the `libfir.a` static library. The `fir_compiler` IP core is specified using `-vlnv` and the IP parameters and function map are specified with `-ip-params` and `-ip-map` respectively.

SDSoC Hardware Functions for Platform IP

If the IP blocks that implement a library are contained within a hardware platform, the associated function mappings must also be included in the SDSoC hardware platform description file described earlier in this document. There is no way to associate the function maps to the IP blocks in Vivado 2014.4, so you must manually add them the hardware platform description file.

In addition, you must provide a component declaration for SDSoC, derived from the IP-XACT component.xml within the IP. You create this component declaration by running the following command within an SDSoC command shell.

```
xsltproc --stringparam P_XD_AUTOESL_COMP TRUE --stringparam P_XD_COMP_TYPE
accelerator -o <IP>.xml <sds_root>/scripts/xsd/ipxact2xdcomp.xsl
<platform_root>/vivado/<platform>.ipdefs/repo/<IP_name_version>/<IP_name_version
>/component.xml
```

In this command,

- <sds_root> is the root of the SDSoC install
- <platform> is the platform name
- <platform_root> is the platform directory
- <IP> is the IP name
- <IP_name_version> is the IP directory in the Vivado project

After creating the function maps and <IP>.xml, copy the <xd:fcnMap> and <xd:component> elements into the hardware platform description file <platform>_hw.pfm at the end, following the platform component and before the </xd:repository> closing tag.

For direct I/O functions that map arguments to AXI-S bus interfaces on a platform IP, you must also add an additional –io option (with no argument) to sdslib command described above.

Testing a Library

To test a library, create a program that uses the 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 use the library libfir.a located at <path to library>.

Examples

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

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

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 multiple –ip-repo options are specified, they are searched in the order listed on the sdscc command line.

You can find an example on how to call an IP hardware function in the SDSoC tools installation under the `samples/hdl_fir_filter` directory.

Exporting Libraries for GCC

When creating an application, SDSoC uses an SDSoC platform as a starting point and builds on top of it. SDSoC creates a hardware design that includes the functions selected for hardware implementation as well as the corresponding data movers. SDSoC also creates the required functions to communicate with these accelerators.

An SDSoC user can create a design and export a library that can be uses by GCC users to develop software on it. This pure software development will not involve any change of hardware, synthesis or implementation.

The elements of a library for GCC are:

- Header files
- Shared library
- CF static library
- SD card image

Header files are typically provided to the software developer in an include directory. This directory must include all function prototype functions delivered in the library as well as any dependent header files to allow for compilation of a software program.

The shared library is created using SDSoC. Source files are compiled with the position independent code flag (-fPIC) and then linked using the –shared switched.

The connectivity of the hardware blocks is determined using a source file that includes a process function that defines how the user will call the library.

Example:

```
File: mmult_call.c
#include "mmult_accel.h"

void mmult_call (float in_A[A_NROWS*A_NCOLS],
                 float in_B[A_NCOLS*B_NCOLS],
                 float out_C[A_NROWS*B_NCOLS])
{
  mmult_accel(in_A, in_B, out_C);
}
```

This example specifies that there is a single call to the function mmult_accel that will be selected for hardware implementation. Multiple functions could be cascaded in which the output of the first call is used as input for the second call. SDSoC will determine the connectivity based on this.

To compile the library, compile the code to implement in hardware in the usual way using SDSoC and also make it position independent using -fPIC. In the following example, the function mmult_accel in the file mmult_accel.cpp is to be implemented in hardware.

```
sdscc -sds-pf zc702 -sds-hw mmult_accel mmult_accel.cpp -sds-end \
-c -fPIC mmult_accel.c -o mmult_accel.o
```

Also, compile the process function code.

```
sdscc -sds-pf zc702 -c -fPIC mmult_call.c -o mmult_call.o
```

Finally, link both and specify the shared library options.

```
sdscc -sds-pf zc702 -shared mmult_accel.o mmult_call.o -o libmmult_accel.so
```

This will create a libmmult_accel.so library that any GCC developer can use for linking.

The above command will also create an sd_card image that will contain the boot files needed to execute the program that links against the library.

You can find a complete example in the samples/mmult_shared_lib/build directory in the SDSoC install.

Compiling using GCC

When a library is exported for use with GCC, there will be a directory structure that contains several elements generated as described in the previous section.

A typical directory structure is:

```
include/
mmult_accel.h
lib/
   libmmult_accel.so
sd_card/
   BOOT.BIN
README.txt
boot.bif
devicetree.dtb
libmmult_accel.so
uImage
uramdisk.image.gz
```

The `include` directory contains all necessary header files to compile the application. Use the –I compiler switch to point to that directory.

The `lib` directory contains all files needed to link the application. Use the –L switch to point to that directory when linking and the –l switch to specify the library.

Copy the sd_card directory into an SD card and use it to boot the board.

As an example, assume you created a file called mmult.cpp that contains the main function and calls the function in the shared library. Compile the file using

```
arm-xilinx-linux-gnueabi-g++ -c -O3 mmult.cpp -o mmult.o
```

Then link the application using

```
arm-xilinx-linux-gnueabi-g++ -O3 mmult.o -L./lib -lmmult_accel -lpthread \
-o mmult.elf
```

This will create an executable called mmult.elf that you can copy into your SD card along with the boot files.

To run the program, boot the board and wait for the command prompt. Execute the following commands on the board.

```
sh-4.3# export LD_LIBRARY_PATH=/mnt
sh-4.3# /mnt/mmult.elf
```

You can find a complete example in the samples/mmult_shared_lib/use directory in the SDSoC install.

Introduction

In this Chapter, we create simple example SDSoC platforms starting from hardware systems built using the Vivado Design Suite.

Recall that an SDSoC platform hardware specification defines a connectivity interface on the Vivado design consisting of AXI and AXI stream, clock, reset, and interrupt ports to which SDSCC can connect hardware functions and data mover channels. This interface encapsulates the platform system as a single "component" seen by the SDSoC compiler that essentially corresponds to an IP Integrator block diagram (BD) for the base platform hardware. The interface is specified through attributes on the BD and IP blocks within the BD.

The process of creating an SDSoC hardware platform specification consists of specifying the platform interface within IP Integrator using tcl commands in the Vivado Tcl Console, and using the Vivado tools to export the hardware description to SDSoC. Rather than explicit "external ports", each port on the interface is a reference to a specific port on an IP within the IP Integrator block diagram.

Essentially any Vivado IP Integrator system that targets the Zynq ® family of SoCs can be the basis of an SDSoC platform. SDSoC automatically sets parameters on the Zynq `processing_system7` IP to enable AXI interfaces as needed, and adds additional DMA interrupts to the system, but otherwise does not modify any of the IP in the platform system.

Example 1: Exporting Direct I/O in an SDSoC Platform

Our first simple example is derived from the zc702 platform included with SDSoC, introducing an IP block that provides an AXIS master to proxy direct input to the Zynq PL (FPGA fabric), and another that provides an AXIS slave to proxy direct output from the PL.

In a real platform, these blocks will be replaced by platform-specific IP that deliver input and output to the PL accessible by SDSoC.

RECOMMENDED: *As you read through this tutorial, you should work through the example provided in* `<sds_root>/samples/platforms/pf_axis/`.

Refer to the `readme.txt` *file for instructions to build and test the platform.*

The Vivado design is captured as a tcl script pf_axis.bd.tcl. To build the hardware platform, open an SDSoC terminal shell, copy `<sds_root>/samples/platforms/pf_axis` into a new directory, and cd into this directory.

```
$ make vivado
```

will build the hardware platform. After the script finishes, open the resulting Vivado project.

```
$vivado pf_axis.xpr
```

Open the block diagram, which should look similar to what is shown below.

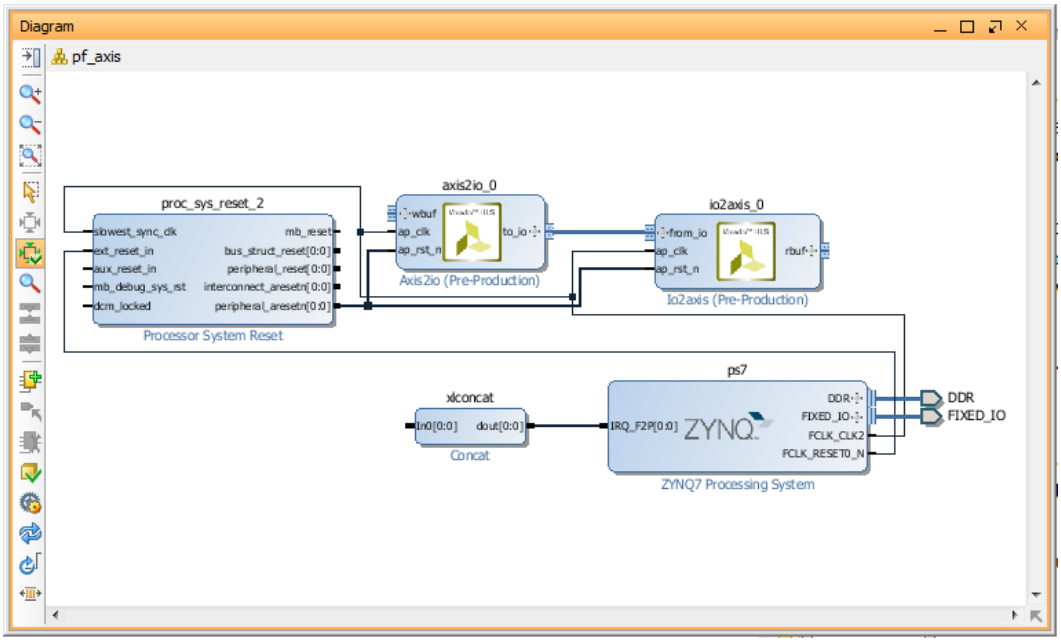

Figure 1: Block Diagram

The platform direct input (output) port is the unterminated `rbuf` (`wbuf`) port on the `io2axis_0` (`pf_write_0`) IP. We have looped back the `to_io` output from `axis2io_0` to the `from_io` input on `io2axis_0` so that we can test our platform, but clearly in a real platform, such ports will be connected to other IPs and ultimately to PL pins on the SoC.

Recall from the introduction that at the SDSoC platform interface, AXI stream interfaces *require* `TLAST`, `TKEEP` sideband signals to comply with the IP underlying SDSoC datamovers. You can confirm existence of these sideband signals within IP Integrator by expanding the ports thusly:

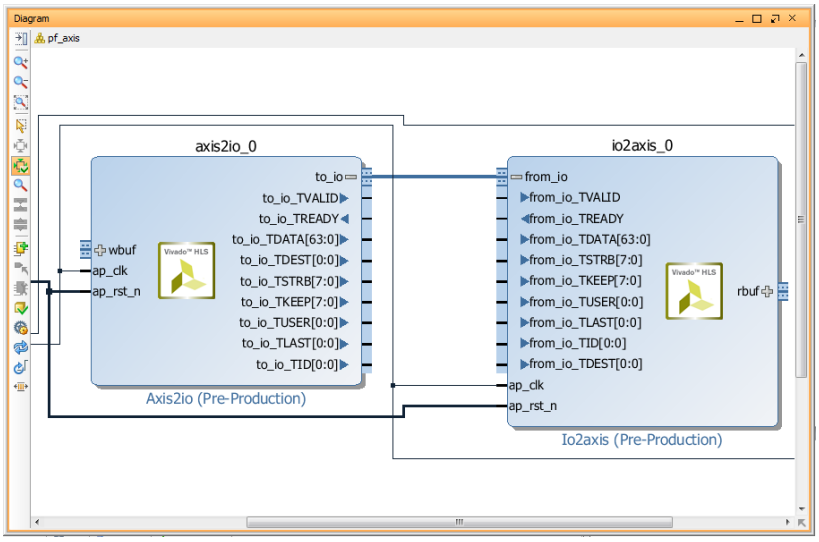

Figure 2: AXI4-Stream Bus Connections

AXI4-Stream buses entirely within the platform do not require these sideband signals.

SDSoC Platform tcl Commands in Vivado

The hardware port interface to an SDSoC platform consists of a set of *unused* AXI or AXI-S bus interfaces on platform IP (e.g., Zynq `processing_system7`), *unused* interrupts on the processing_system7 block, clock ports, and synchronized resets, e.g., provided by `proc_sys_reset` IP in the Vivado catalog.

From the SDSoC terminal, open pf_axis.bd.tcl in a text editor. Most of this file was generated by IP Integrator's write_bd_tcl command. Scroll down to near the end of the file (starting on line 428) to inspect the tcl API calls described in Chapter 2 to mark the SDSoC platform interfaces.

1. The following command enables IP integrator to export additional meta-data to SDK to create an SDSoC hardware platform description.

   ```
   set_param project.enablePlatformHandoff true
   ```

2. The platform default clock is declared thusly.

   ```
   set_property BD_ATTRIBUTE.SDI_PFM_CLOCK TRUE [get_bd_pins /ps7/FCLK_CLK2]
   ```

3. Declare platform clocks and their associated ID with the following commands.

   ```
   set_property BD_ATTRIBUTE.SDI_PFM_CLOCK_ID 2 [get_bd_pins /ps7/FCLK_CLK2]
   set_property BD_ATTRIBUTE.MARK_SDI true [get_bd_pins /ps7/FCLK_CLK2]
   ```

4. Platform resets must be synchronized to a specific clock using proc_sys_reset IP blocks. The following commands specifies one of four platform reset bundles.

   ```
   set_property BD_ATTRIBUTE.MARK_SDI true [get_bd_pins /proc_sys_reset_2/interconnect_aresetn]
   set_property BD_ATTRIBUTE.MARK_SDI true [get_bd_pins /proc_sys_reset_2/peripheral_aresetn]
   set_property BD_ATTRIBUTE.MARK_SDI true [get_bd_pins /proc_sys_reset_2/peripheral_reset]
   ```

The remaining `set_property` commands tag resets similarly.

5. Declare the io2axis master and axis2io slave AXI-S bus interfaces that proxy the direct I/O with the following commands.

```
set_property BD_ATTRIBUTE.MARK_SDI true [get_bd_intf_pins /io2axis_0/rbuf]
set_property BD_ATTRIBUTE.MARK_SDI true [get_bd_intf_pins /axis2io_0/wbuf]
```

All other commands in this script are standard Vivado commands. The following command exports a hardware handoff file from Vivado to SDK.

```
write_hwdef  -file "[file join ${pf}.sdk ${pf}_wrapper.hdf]"
```

An SDSoC utility called HSi (hardware/software interface manager) handles the system handoff from Vivado to SDSoC. From the cmd shell you have open, in the pf_axis/vivado directory, invoke HSM by executing `hsm`. From the hsm command line, execute the following two commands

```
open_hw_design "vivado/pf_axis.sdk/pf_axis_wrapper.hdf"
generate_target {sdi} [current_hw_design] -dir hsi
```

This will generate the hardware platform description file `hsi/pf_axis.pfm`. When you run the '`make pf_axis`' target, this file will be copied into the platform directory, renamed to `pf_axis_hw.pfm`. Open the platform description file in a text editor and look for the clocks, reset, and bus interfaces that were specified in `pf_axis.bd.tcl`.

IMPORTANT: *Not all required meta-data is automatically generated from Vivado via tcl APIs, so you must add this meta-data by hand to the generated XML file <platform>_hw.pfm.*

Known issues include:

- *Platform AXI4-Stream stream bus interfaces containing `TLAST`, `TKEEP` sideband signals must have the `xd:hasTlast` attribute (with value "`true`") in the corresponding `xd:busInterface` element, but these are not automatically detected.*

SDSoC Platform Software Libraries

The SDSoC compiler (`sdscc`) maps application program dataflow between hardware functions into bus-based connections between the IPs that implement the hardware functions in the platform and accelerator functions in programmable logic fabric.

Consequently, every platform IP that exports a direct I/O AXIS interface must have a C-callable library that exports functions that SDSoC will use to realize the program in hardware.

The '`all`' make target in `pf_lib/Makefile` uses the SDSoC `sdslib` utility to create a static C-callable library for the platform as described in Chapter 3 of this document. It is worth noting that you can also use `sdslib` to wrap HDL IPs as accelerator functions that can be linked by SDSoC within an application.

1. *Define the C-callable interfaces for I/O IPs.* In the SDSoC command shell, 'cd' into the pf_lib directory. There are two platform IPs that require C-callable functions, `pf_read` and `pf_write`. The hardware functions are defined in `pf_read.cpp` and `pf_write.cpp`, as follows:

```
void pf_read(u64 rbuf[N]) {}
void pf_write(u64 wbuf[N]) {}
```

The function bodies are empty; SDSoC will fill in stub function bodies as needed. Multiple functions can map to a single IP, as long as the function arguments all map onto the set of IP ports, and do so consistently (e.g., two array arguments of different sizes cannot map onto a single AXIS port on the corresponding IP).

2. *Define the mappings from the function interfaces to the respective IP ports*. Each function in the C-callable interface requires a mapping from the function arguments (and name) to the IP ports (and name). The mappings for `pf_read` and `pf_write` IPs are captured in `pf_read.fcnmap.xml` and `pf_write.fcnmap.xml`. Open `pf_read.fcnmap.xml`.

```
<xd:fcnMap xd:fcnName="pf_read" xd:componentRef="pf_axis">
    <xd:arg
      xd:name="read_buf"
      xd:direction="out"
      xd:busInterfaceRef="io2axis_0_rbuf"
      xd:portInterfaceType="axis"
      xd:arraySize="128"
      xd:dataWidth="64"
    />
    <xd:latencyEstimates xd:best-case="258" xd:worst-case="258"
xd:average-case="258"/>
    <xd:resourceEstimates xd:LUT="62" xd:FF="47" xd:BRAM="0" xd:DSP="0"/>
</xd:fcnMap>
```

This information should be fairly self-explanatory. Each function argument requires name, direction, IP bus interface name, interface type, and data width. Array arguments must specify array size. Scalar arguments must specify a register offset. SDSoC uses the latency estimates during high level scheduling.

IMPORTANT: *the `fcnMap` associates the platform function `pf_read` with the platform bus interface `pf_read_0_rbuf` on the platform component `pf_axis`, not the bus interface on the actual IP within the `pf_axis` platform that implements the function. In `pf_axis_hw.pfm` the pf_axis bus interface ("port") named "`io2axis_0_rbuf`" (`axis2io_0_wbuf`) contains the actual mapping to the IP in the `xd:instanceRef` attribute (`io2axis_0`).*

3. *Parameterize the IP*. IP customization parameters must be set at compile time in an XML file. In this lab, the platform IP has no parameters, so the file pf_read.params.xml and pf_write.params.xml are particularly simple. To see a more interesting example, open `<sdsroot>/samples/fir_lib/build/fir_compiler.{fcnmap,params}.xml` in the SDSoC install tree. This example maps multiple functions onto the Vivado FIR Filter Compiler IP core.

4. *Build the library*. The sdslib commands are as follows.

```
sdslib -lib libpf_axis.a \
  pf_read pf_read.cpp \
  -vlnv xilinx.com:sds:pf_read:1.0 \
  -ip-map pf_read.fcnmap.xml \
  -ip-repo ./xilinx_com_sds_pf_read_1_0 \
```

```
    -ip-params pf_read.params.xml

  sdslib -lib libpf_axis.a \
    pf_write pf_write.cpp \
    -vlnv xilinx.com:sds:pf_write:1.0 \
    -ip-map pf_write.fcnmap.xml \
    -ip-repo ./xilinx_com_sds_pf_write_1_0 \
    -ip-params pf_write.params.xml
```

Note that you can call `sdslib` repeatedly for a library. Each call will add additional functions into the library.

Observe the similarity to an SDSoC call to compile a function as described in SDSoC User Guide: Introduction to SDSoC (UG1027) Chapter 14, *Incorporating HDL IP into SDSoC* [1].

SDSoC Platform Software Description

As described in Chapter 2, the SDSoC platform software description is an XML file that contains information required to link against platform libraries, and create boot images to run the application on the hardware plaform. There is currently no automation for this step.

The pf_axis platform reuses all of the zc702 boot files.

1. *Open the platform software description*, `pf_axis/pf_axis/pf_axis_sw.pfm`. The following element instructs SDSoC where to find the platform software libraries created in the platform directory.

```
<xd:libraryFiles
     xd:os="linux"
     xd:includeDir="arm-xilinx-linux-gnueabi/include"
     xd:libDir="arm-xilinx-linux-gnueabi/lib"
     xd:libName=”pf_axis”
/>
```

Similarly, the boot files are specified thusly.

```
 <xd:bootFiles xd:os="linux"
               xd:bif="boot/linux.bif"
               xd:readme=”boot/generic.readme”
               xd:devicetree="boot/devicetree.dtb"
               xd:linuxImage="boot/uImage"
xd:ramdisk="boot/uramdisk.image.gz" />
```

Building the pf_axis Platform

To build the SDSoC platform from the SDSoC terminal,

```
$ make pf_axis
```

The ‘pf_axis’ build target pulls together all the components of the SDSoC platform. It creates a pf_axis root directory for the platform, expands an archive of the Vivado project into a ‘vivado’ subdirectory. It then calls make –C pf_lib to create the software library, copying the library and header into an arm-xilinx-linux-gnueabi subdirectory, and copies the pf_axis_hw.pfm and pf_axis_sw.pfm meta-data files into the platform directory.

This completes the SDSoC platform capture.

Test the SDSoC Platform

To test the platform, we have provided a very simple C++ application in `pf_axis/pf_test/pf_test.cpp`. Key points in this function are the ways in which buffers are allocated using sds_alloc,

```
u64 *wbuf = (u64 *) sds_alloc(N * sizeof(u64));
u64 *rbuf = (u64 *) sds_alloc(N * sizeof(u64));
```

and the way that the platform functions are invoked to either read from platform inputs or write to platform outputs.

```
pf_write(wbuf);   // write to platform output
pf_read(rbuf);    // read from platform input
```

`cd` into the `pf_axis/pf_test` directory and execute

```
make all
```

and after SDSoC creates an sd_card image, copy onto an SD card, boot, and run `pf_test.elf`. You should see output similar to

```
sh-4.3# ./pf_axis.elf
registering devices
generating device nodes...
Test PASSED!
sh-4.3#
```

Example 2: Software Control of Platform IP

The next example is a single clock zc702 based platform with a general purpose I/O (GPIO) IP block (axi_gpio) implemented in programmable logic and connected to the LEDs on the board. The axi_gpio IP has an associated Linux kernel driver, but here we demonstrate how to provide software control of platform IP using Linux's UIO (userspace I/O) framework to communicate to the GPIO directly from a test application. We will show how to create a platform software library outside of SDSoC, and make it available to applications within the SDSoC Design Environment.

RECOMMENDED: *As you read through this tutorial, you should work through the example provided in `<sds_root>/samples/platforms/zc702_led/`. Refer to the readme.txt file for instructions to build and test the platform.*

Build Vivado System and SDSoC Platform Hardware Description

Make a local working copy of `samples/platforms/zc702_led`, and from an SDSoC terminal shell, `cd` into this directory. There is a Makefile in this directory to build the platform. We have provided a tcl script that defines the platform.

From the terminal shell, run

```
$ make vivado
```

which invokes Vivado on the `zc702_led.bd.tcl` script to build the base system.

The IP Integrator block diagram for the platform is shown below.

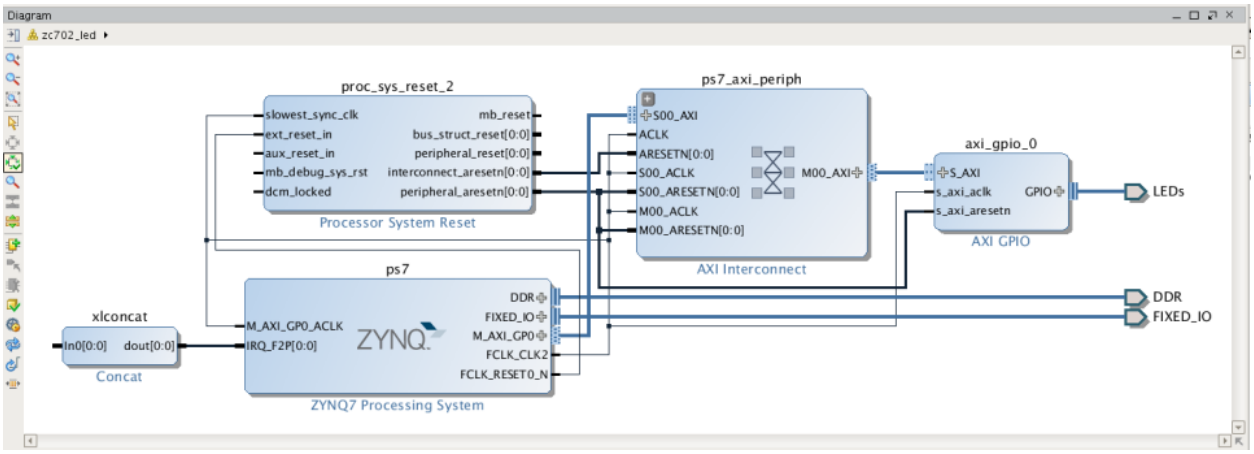

Figure 3: Block Diagram

As described in Chapter 2 of this document, this script invokes tcl APIs to specify the default platform clock, the corresponding clock ID, and the reset interfaces.

```
set_property BD_ATTRIBUTE.SDI_PFM_CLOCK TRUE [get_bd_pins /ps7/FCLK_CLK2]
set_property BD_ATTRIBUTE.SDI_PFM_CLOCK_ID 2 [get_bd_pins /ps7/FCLK_CLK2]
set_property BD_ATTRIBUTE.SDI_PFM_UIO true [get_bd_cells /axi_gpio_0]
set_property BD_ATTRIBUTE.MARK_SDI true [get_bd_pins /ps7/FCLK_CLK2]
set_property BD_ATTRIBUTE.MARK_SDI true [get_bd_pins /proc_sys_reset_2/interconnect_aresetn]
set_property BD_ATTRIBUTE.MARK_SDI true [get_bd_pins /proc_sys_reset_2/peripheral_aresetn]
set_property BD_ATTRIBUTE.MARK_SDI true [get_bd_pins /proc_sys_reset_2/peripheral_reset]
```

SDSoC employs the Linux UIO framework for hardware functions, and consequently must be informed of any UIO platform devices within the platform. For the zc702_led platform, this is accomplished by the following API call in the script:

```
set_property BD_ATTRIBUTE.SDI_PFM_UIO true [get_bd_cells /axi_gpio_0]
```

which declares the axi_gpio_0 instance to be a UIO device.

The script then builds the design and creates a hardware description that defines the project for the `hsi` utility provided as part of the SDSoC Design Environment.

After `make` completes, from the terminal shell, run

```
$ hsi
hsi% open_hw_design "vivado/zc702_led.sdk/zc702_led_wrapper.hdf"
hsi% generate_target {sdi} [current_hw_design] -dir hsi
hsi% quit
```

This will generate the platform hardware description `hsi/zc702_led.pfm`.

Build SDSoC Platform

To build the SDSoC platform, from the terminal shell run

```
$ make zc702_led
```

This build target will unzip an archived version of the Vivado project built by the previous step, and then build a platform software library to access the GPIO block driving the LEDs on the zc702 board by running

```
make -C lib
```

The boot environment for the zc702_led is identical to the zc702 platform that is provided as part of SDSoC except for the devicetree.dtb which is required to register the axi_gpio platform peripheral.

Linux UIO Framework for Userspace Control of Platform IP

The default `lib` build target creates a software library containing the UIO driver for the `axi_gpio` block consisting of the files `uio_axi_gpio.[ch]`. This driver provides a simple API for controlling the GPIO IP. Please refer to `test/pf_axis.cpp` for example usage of the API.

The devicetree provided as part of the zc702_led platform was created by hand by modifying the devicetree.dtb from the zc702 platform. First, the zc702 `devicetree.dtb` was converted to a text format (`.dts` or devicetree source) using the dtc compiler

```
dtc -I dtb -O dts -o devicetree.dts boot/devicetree.dtb
```

IMPORTANT: *the dtc compiler must be built and run on a Linux host machine or Virtual Machine (VM). You can find the source to build the dtc at https://github.com/Xilinx/linux-xlnx/tree/master/scripts/dtc.*

To register the `axi_gpio_0` platform peripheral with Linux, add the following devicetree blob to the devicetree as required by the UIO framework:

```
gpio@41200000 {
        compatible = "generic-uio";
        reg = <0x41200000 0x10000>;
};
```

This blob was created by hand and inserted into the device tree as the lexically first occurring generic-uio device within the amba record in the devicetree.

The name must be unique; we have adopted a convention using the base address for the peripheral computed by Vivado during system generation as a guarantee. The value of the `reg` member must be the base address for the peripheral and the number of byte addresses in the corresponding address segment for the IP. Both of these are visible in the Vivado IP integrator Address Editor.

To convert the device tree back to binary format required by the Linux kernel, again employ the `dtc` device tree compiler.

```
dtc -I dts -O dtb -o devicetree.dtb boot/devicetree.dts
```

The UIO driver in the `zc702_led/lib` directory provides the required hooks for the UIO framework

```
int axi_gpio_init(axi_gpio *inst, const char* instnm);
int axi_gpio_release(axi_gpio *inst);
```

Any application that accesses the peripheral must call the initialization function before accessing the peripheral and must release the resource when it's finished. The test application in `zc702_led/test` provides a sample usage.

For more information on device trees and the Linux UIO framework, we highly recommend training material available on the Web, for example

http://www.free-electrons.com/docs

Test the SDSoC Platform

To test the platform, from the terminal shell run

```
$ make -C test
```

The test application contains a simple arraycopy hardware function using the SDSoC axi_lite datamover, invoked within a loop. The LEDs on the zc702 are lit to match the binary representation of the loop index.

Although quite simple, this design demonstrates how you can employ the Linux UIO framework to control platform peripherals and provide a software library as part of your platform for use in SDSoC applications.

Example 3: Sharing a `processing_system7` AXI Port

This example demonstrates how to share a `processing_system7` AXI port between platform and SDSoC-generated IP. The AXI port must be connected to an axi_interconnect IP block within the platform, and the platform exports a corresponding master or slave port on the interconnect. During system generation, SDSoC will map data channels to the platform by cascading an interconnect connected to this exported port.

RECOMMENDED: *As you read through this tutorial, you should work through the example provided in* `<sds_root>/samples/platforms/zc702_acp/`. *Refer to the* `readme.txt` *file for instructions to build and test the platform.*

IMPORTANT: *whenever a platform exports an AXI or AXIS interface that is not part of the processing_system7 IP, every SDSoC application targeting the platform must use every such bus interface. Otherwise, the system will error out during Vivado due to unterminated interfaces. SDSoC is able to terminate unused AXI interfaces on the processing_system7 IP itself.*

Build Vivado System and SDSoC Platform Hardware Description

Make a local working copy of `samples/platforms/zc702_acp`, and from an SDSoC terminal shell, `cd` into this directory. There is a Makefile in this directory to build the platform. We have provided a tcl script that defines the platform.

From the terminal shell, run

```
$ make vivado
```

which invokes Vivado on the `zc702_acp.bd.tcl` script to build the base system.

The IP Integrator block diagram for the generated system is shown below.

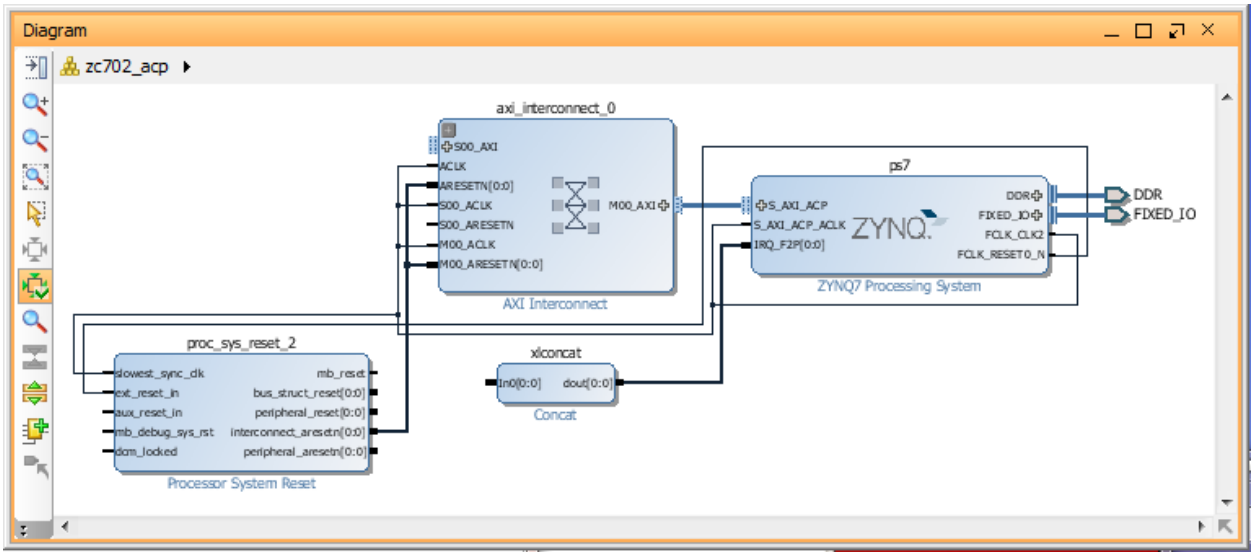

Figure 4: Block Diagram

This design is particularly simple; the platform does not actually even use the ACP, other than to export a port on an axi_interconnect. A real platform will use some of the interconnect slave ports, which must be the least significant Snm_AXI ports (no gaps) and then export the least significant indexed unused port.

As described in Chapter 2 of this document, this script invokes tcl APIs to specify the default platform clock, the corresponding clock ID, and the reset interfaces.

```
set_property BD_ATTRIBUTE.SDI_PFM_CLOCK TRUE [get_bd_pins /ps7/FCLK_CLK2]
set_property BD_ATTRIBUTE.SDI_PFM_CLOCK_ID 2 [get_bd_pins /ps7/FCLK_CLK2]
```

```
set_property BD_ATTRIBUTE.MARK_SDI true [get_bd_pins /ps7/FCLK_CLK2]
set_property BD_ATTRIBUTE.MARK_SDI true [get_bd_pins /proc_sys_reset_2/interconnect_aresetn]
set_property BD_ATTRIBUTE.MARK_SDI true [get_bd_pins /proc_sys_reset_2/peripheral_aresetn]
set_property BD_ATTRIBUTE.MARK_SDI true [get_bd_pins /proc_sys_reset_2/peripheral_reset]
```

Every ps7 AXI interface is connected to an axi_interconnect IP. To share such an interface with SDSoC, the platform exports the least significant unused slave interconnect on the IP that masters the ps7 `S_AXI_ACP`. Because, in this example, there are no AXI masters within the platform, the platform exports `S00_AXI` with the following commands.

```
set_property BD_ATTRIBUTE.MARK_SDI true [get_bd_intf_pins /axi_interconnect_0/S00_AXI]
set_property BD_ATTRIBUTE.MARK_SDI true [get_bd_pins /axi_interconnect_0/S00_ARESETN]
set_property BD_ATTRIBUTE.MARK_SDI true [get_bd_pins /axi_interconnect_0/S00_ACLK]
```

You must mark the clock and reset pins corresponding to an AXI interface.

The script builds the design and creates a hardware description that defines the project for the `hsi` utility provided as part of the SDSoC Design Environment.

After `make` completes, from the terminal shell, run:

```
$ hsi
hsi% open_hw_design "vivado/zc702_acp.sdk/zc702_acp_wrapper.hdf"
hsi% generate_target {sdi} [current_hw_design] -dir hsi
hsi% quit
```

This will generate the platform hardware description `hsi/zc702_acp.pfm`.

In the current Vivado release, some of the necessary platform meta-data for this example is missing or incorrect, and must be added by hand.

The NUM_SI parameter for the axi_interconnect_0 has an error in its adjustment value, which should be '1' not '0'. It should be as follows, with changes highlighted here in red:

```
 <xd:parameter
    xd:instanceRef="axi_interconnect_0"
    xd:isValid="'true'"
    xd:name="NUM_SI"
    xd:value="number(count($designComponent/xd:connection/xd:busInterface[@xd:insta
nceRef=$instance and starts-with(substring(@xd:name,1,1),'S') and starts-
with(substring(@xd:name,4,4),'_AXI')])+1)"
    />
```

The S00_AXI element has an incorrect `xd:clockRef` attribute value and is missing meta-data. It should be as follows, with changes highlighted here in red:

```
<xd:busInterface
    xd:busInterfaceRef="S00_AXI"
    xd:busTypeRef="aximm"
    xd:clockRef="axi_interconnect_0_S00_ACLK"
    xd:resetRef="axi_interconnect_0_S00_ARESETN"
    xd:instanceRef="axi_interconnect_0"
    xd:mode="slave"
    xd:numIds="4"
```

```
        xd:coherent="true"
        xd:name="axi_interconnect_0_S00_AXI"
        xd:memport="IC"
        />
```

Finally, there is a missing element that you must add:

```
<xd:busInterface
    xd:busInterfaceRef="S00_ARESETN"
    xd:busTypeRef="reset"
    xd:instanceRef="axi_interconnect_0"
    xd:mode="slave"
    xd:name="axi_interconnect_0_S00_ARESETN"
    />
```

Make these changes by hand in a text editor.

Build SDSoC Platform

To build the SDSoC platform, from the terminal shell run

```
$ make zc702_acp
```

This build target will unzip an archived version of the Vivado project built by the previous step.

In general, a platform clock can be driven by a clock source other than the ps7 (e.g., a clocking wizard), and can be completely independent of the SDSoC-generated IPs. For this reason, the clock port for each AXI interconnect must be left unterminated in the platform Vivado project. SDSoC will automatically connect this to the correct clock when generating the final system.

Open the Vivado project zc702/vivado/zc702_acp.xpr and disconnect /axi_interconnect_0/S00_ACLK from the clock net. SDSoC will connect this clock port to the same clock net as the AXI mastering this bus connection. You will need to reconnect all of the other IP clock pins after deleting the connection to the axi_interconnect.

The zc702_acp platform system is shown below.

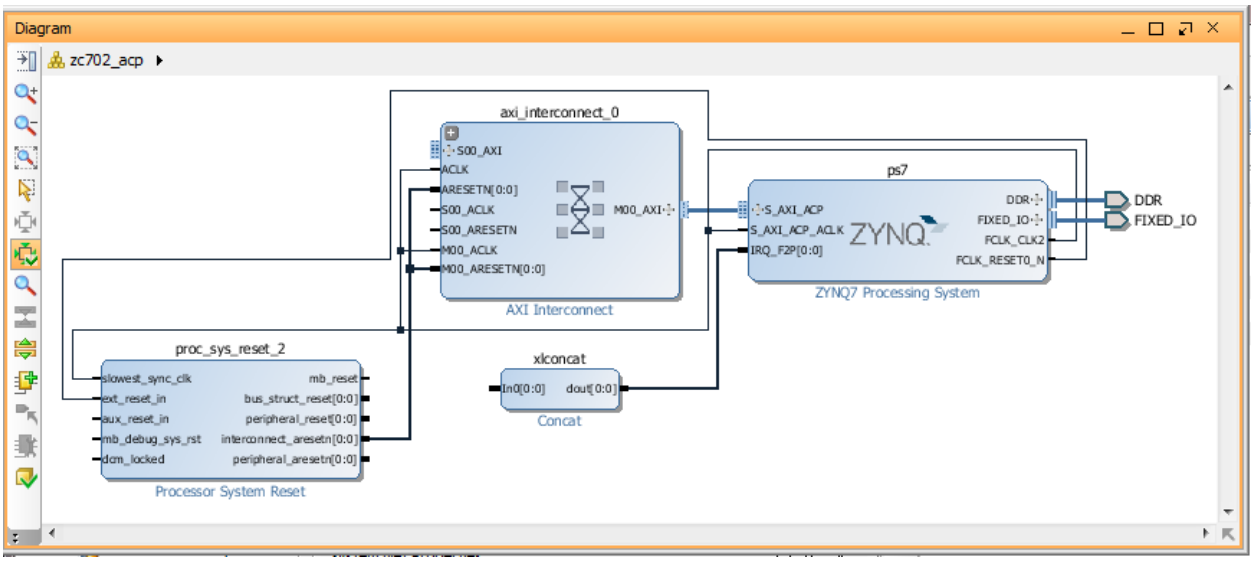

Figure 5: zc702_acp Block Diagram

IMPORTANT: *you must manually delete the connection to the axi_interconnect slave clock port so that SDSoC can connect this port to whatever platform clock will be driving this AXI slave. Otherwise, users of your platform will be unable to generate correct logic whenever SDSoC clock and the platform clock are different.*

The boot environment for the zc702_led is identical to the zc702.

Test the SDSoC Platform

To test the platform, from the terminal shell run

```
$ make -C pf_test
```

The test application contains a simple arraycopy hardware function using the SDSoC zero_copy (accelerator mastered AXI bus). Load the contents of pf_test/sd_card into an SD card and boot.

```
$ /mnt/zc702_acp_test.elf
```

This example demonstrates how ps7 ports can be shared between platform and SDSoC generated logic.

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

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 1027 – Introduction to SDSoC*, `<sdsoc_install_root>/docs/labs/ug1028-intro_to_sdsoc.pdf.`
3. *User Guide 1028 – Getting Started, Chapter 5 Lab 1 – Creating, Building, and Running an SDSoC System*, `<sdsoc_install_root>/docs/labs/ug1028-sdsoc_getting_started.pdf.`
4. *User Guide 1028 – Getting Started, Chapter 5 Lab 2 – Introduction to System Optimizations,* `<sdsoc_install_root>/docs/labs/ ug1028-sdsoc_getting_started.pdf.`
5. *User Guide 1028 – Getting Started, Chapter 5, Lab 3 – Introduction to System Debugging,* `<sdsoc_install_root>/docs/labs/ ug1028-sdsoc_getting_started.pdf.`

Please Read: Important Legal Notices

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

© Copyright 2014-2015 Xilinx, Inc. Xilinx, the Xilinx logo, Artix, ISE, Kintex, Spartan, Virtex, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. All other trademarks are the property of their respective owners.