

DMA Linux Lab

Overview

This lab will provide a base design to help a designer get AXI Streaming data in and out of PL or PS DDR. A user should be able to take this design and modify it to something more relevant for their application. This example works with the movement of Memory Mapped data as well as Streaming data.

Lab Instructions

The following instructions refer to a zip file containing all of the source files to execute the entire lab. This zip file is labeled 182mover-6-26-2018.zip. The contents of this zip file should be extracted into a clean directory and preferably within Linux. The lab will make use of the primary Xilinx Tools:

- 2018.2 Vivado
- 2018.2 PetaLinux
- 2018.2 SDSoC

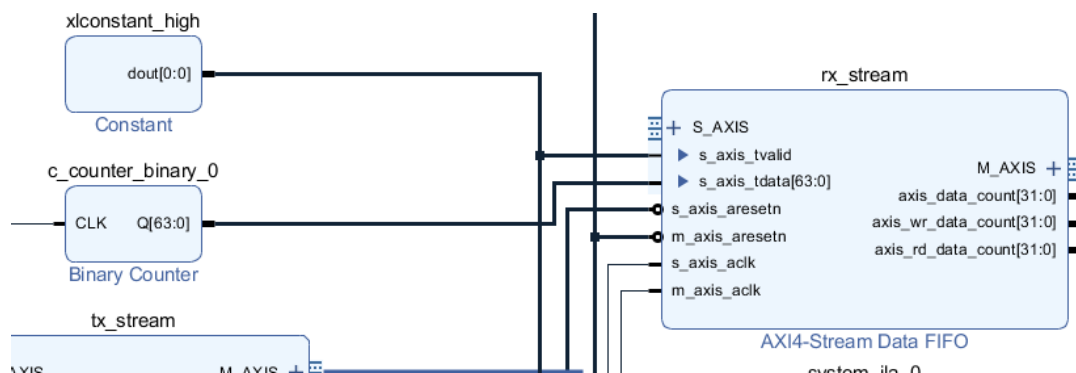
Please ensure to have access to these tools before attempting to start.

Step 1: Vivado Project

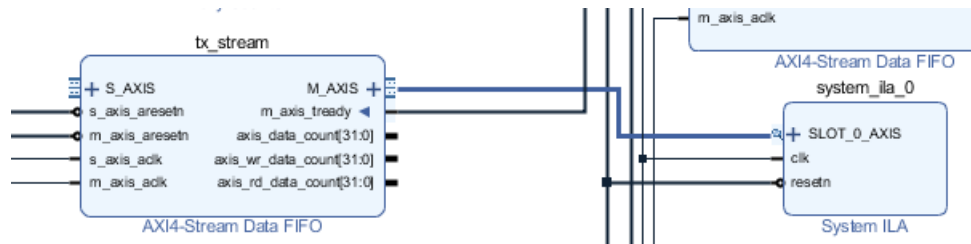
This lab is designed for a user who is already familiar with Vivado and for that reason a Tcl script is used to create the Vivado design. Ensure the zip file is extracted in a clean directory. Then change directory into the “viv” directory at the base directory where the zip is extracted. If it’s not there, create the directory. Within this directory, ensure the vivado tools are sourced and type:

```
>>vivado -source ../scripts/vivado_prj_create.tcl
```

This will create a Vivado project and have two streaming IPs the software from SDSoC will interface with. This is a good time to look at the Vivado IPI design to analyze the rx_stream and tx_stream IPs; which are AXI4-Stream Data FIFOs. Notice the rx_stream FIFO has a completely unconnected M_AXIS interface while it’s constantly receiving counting data as an input:

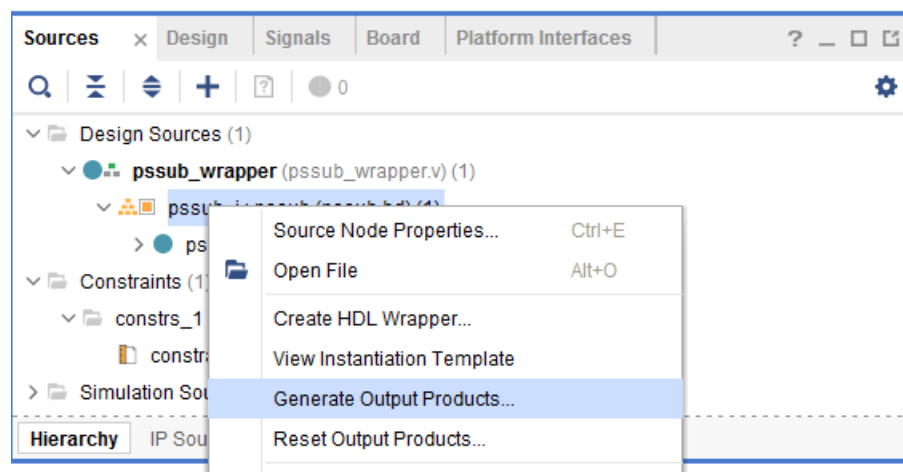


Next, take a look at the tx_stream IP and notice how the S_AXIS port is completely unconnected. This port will be interfacing with SDSoC tools/framework. There’s a system ILA on the other end to monitor the data coming out.



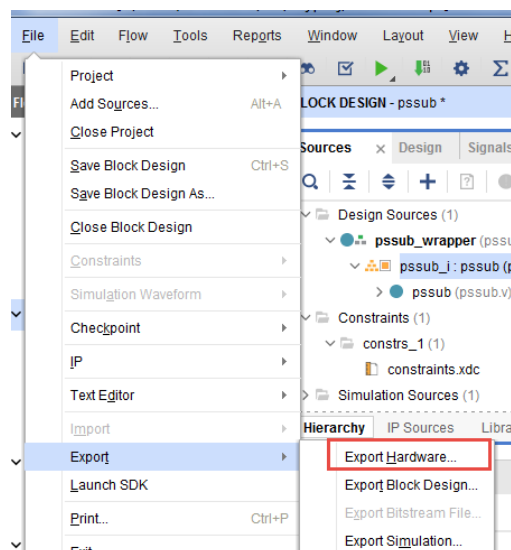
Lastly, notice the MIG core and how we've placed a System ILA on the MIG AXI interface. This system ILA can be looked at to see the traffic to the Memory Controller (MIG).

Generate the output products by right-clicking the BD in the sources window and selecting "Generate Output Products..."



Keep the defaults and select "Generate"

While the Design Runs are running, you should be able to Export the HDF. The generation of the design runs should take a couple minutes, but you shouldn't have to wait till each IP synthesized. Below is how to export the HDF:



Then select the Defaults and export the HDF local to the project. This will export the HDF into the moverlab.sdk directory.

After the HDF is exported, we're ready for PetaLinux.

Step 2: PetaLinux

The HDF will be imported into the PetaLinux project utilizing the Template flow. First create the PetaLinux project within the "peta" directory of the lab files. We'll assign the project name of "petamover," although this is arbitrary.

```
>>petalinux-create -t project -n petamover --template zynqMP
```

After the project is created, change directory into petamover:

```
>>cd petamover
```

Now it's time to import the hdf created from Vivado:

```
>>petalinux-config --get-hw-description=../viv/myproj/moverlab.sdk/
```

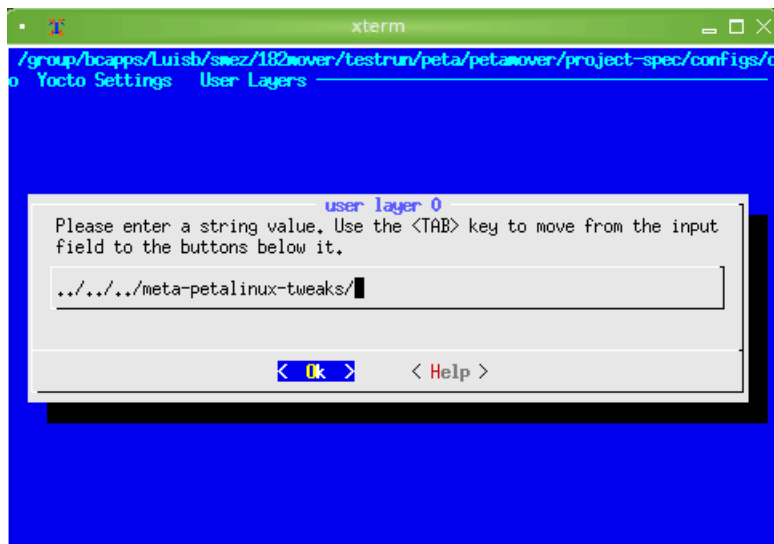
This command inherently imports the HDF. The menuconfig that pops up after this command will allow us to add a meta layer. This meta layer will add the following settings:

1. The SD card is automatically mounted to the /media/card when it's plugged into the ZCU102
2. The output from the 'ls' command is colorized
3. Some additional utility aliases, such as 'lh' which produced a detailed ls listing with human-readable file sizes are defined.

Technically, the meta layer added above is not required, however it's helpful to have the SD card automatically mounted.

Set the user layer 0 to the meta-petalinux-tweaks located in the directory where we extracted files. The setting is located here:

Yocto Settings-> User Layers -> user layer 0



(Make sure to go back three directories since everything is relative to a build directory within the PetaLinux project)

Next, copy the device tree updates into the project directory:

```
>>cp ../../device_tree/system-user.dtsi ./project-spec/meta-user/recipes-bsp/device-tree/files/
```

Take a look at the device tree modifications and notice the entry below added for the use of the APF drivers used within SDSoC:

```
2 / {
3     xlnk {
4         compatible = "xlnx,xlnk-1.0";
5     };
6 };
```

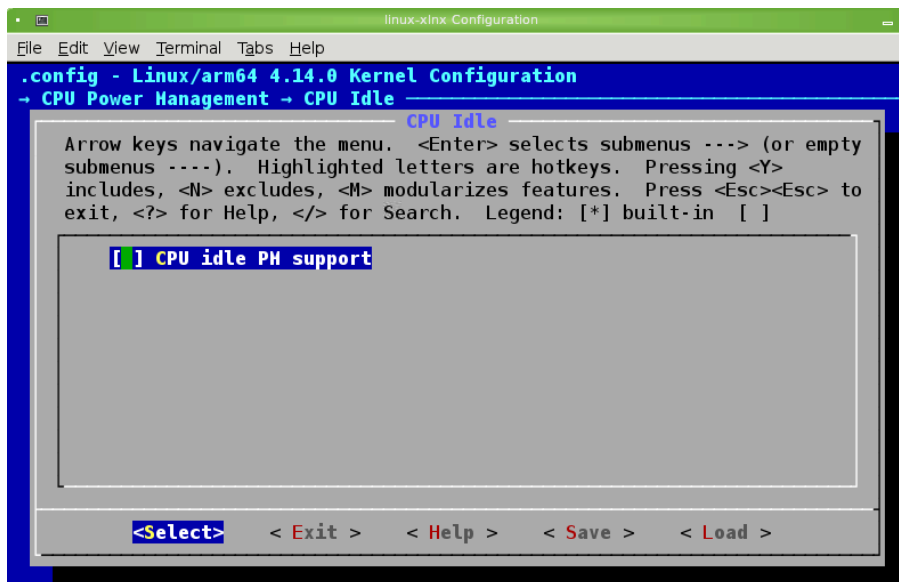
Now it's time to modify the kernel by disabling some of the power management features and enable the APF drivers. We're disabling the power management features because we want to debug with the ILA cores. If we keep these Power Management features, the monitoring with the ILA cores will cause a hang on our embedded Linux session. The APF drivers need to be installed so we can use the framework from used SDSoC.

To modify the kernel type the following:

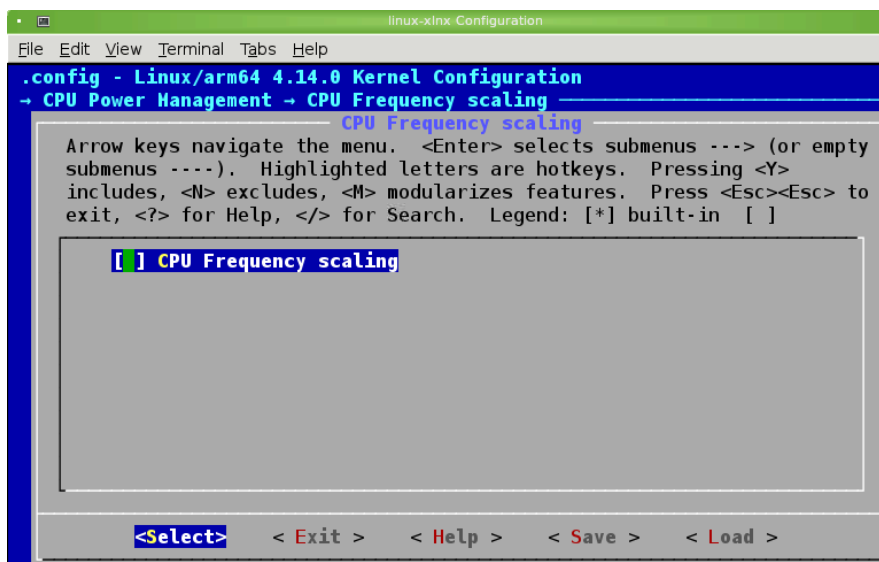
```
>>petalinux-config -c kernel
```

To disable the power management feature go to:

CPU Power Management->CPU Idle->CPU idle PM support

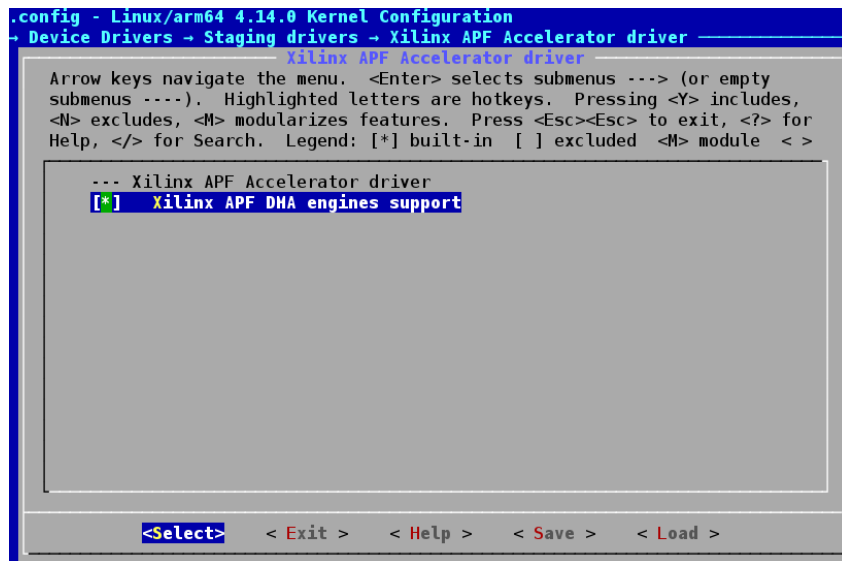


CPU Power Management->CPU Frequency scaling->CPU Frequency scaling



To enable the APF drivers go to:

Device Drivers->Staging drivers->Xilinx APF Accelerator driver->Xilinx APF DMA engines support



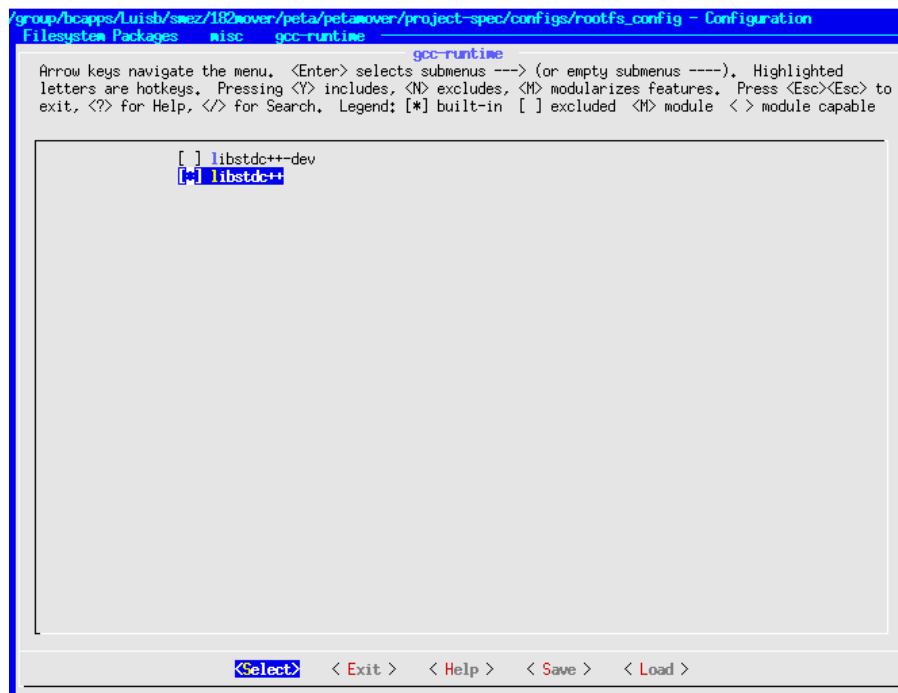
Save the changes to the kernel options.

Next, modify the root file system to add the libstdc++ gcc-runtime libraries.

>>petalinux-config -c rootfs

To add the gcc runtime libraries, go to the following:

Filesystem Package->misc->gcc-runtime->libstdc++



Now build the project:

>>petalinux-build

(((STOP HERE for the first portion!!!!)))

After the project builds, change directory to where the zip file was extracted:

```
>>cd ../../
```

Then source the move_linuximagefile.sh script to copy (and organize) the relevant files for SDSoC into the sw_files directory.

```
>>source ./scripts/move_linuximagefile.sh
```

Step 3: Create the DSA

Within the **Vivado Tcl Console** (with the Vivado project open), source the createdsa.tcl script in the 'scripts' directory; as shown below.

```
>>source ../scripts/writedsa.tcl
```

While it's building, open the writedsa.tcl script and look at how the commands correlate to the Vivado IPI project.

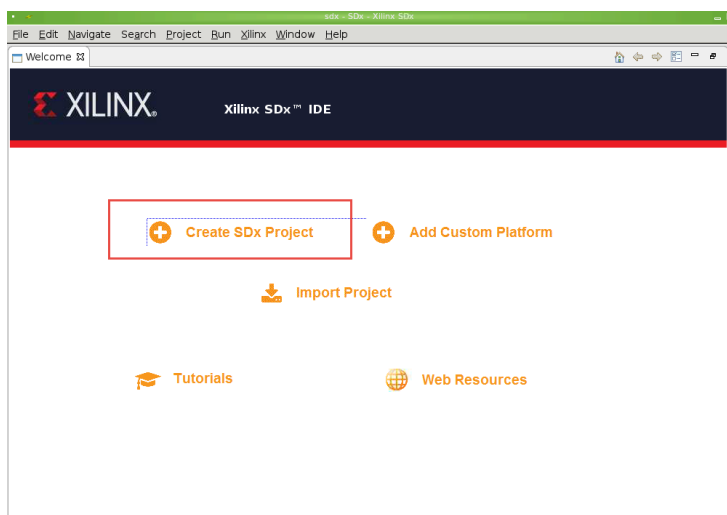
Key takeaway 1: Look at the AXIS_PORT and how they map to the AXI Stream IPs.

Key takeaway 2: Notice how the AXI_PORT property enables SDSoC to understand the memory mapped resources available.

Key takeaway 3: Look at how SDSoC knows about the clocks, resets, and interrupts.

Step 4: SDSoC - Creating the Platform

SDSoC is used to create the platform. Launch SDSoC by typing 'sdx' in the terminal. The welcome screen should show up and now select the "Create SDx Project" button:



Select the "Platform" option as shown below and select "Next":

Project Type

Choose the project type to create.

☐ Application

Create a new application targeting Embedded Processors (ARM, MicroBlaze) or x86 with or without hardware acceleration. You can start with example templates or add your own sources. You will be able to start with an installed platform, a pre-created platform or from Vivado outputs [Device Support Archive (DSA) or Hardware Definition File (HDF)]

☒ Platform

Create a platform project from output of Vivado [Device Support Archive (DSA) or Hardware Definition File (HDF)]. A platform will enable you to specify options for the kernels, BSPs as well as settings required for creating new applications. Platforms are currently supported for embedded (SDK and SDSoc) software developers.

Next, specify the DSA built from Vivado. The DSA is located in the zipExtract/viv/myproj/dsa/ directory; as shown below and select Finish:

Platform Specification

Specify the DSA/HDF and platform options

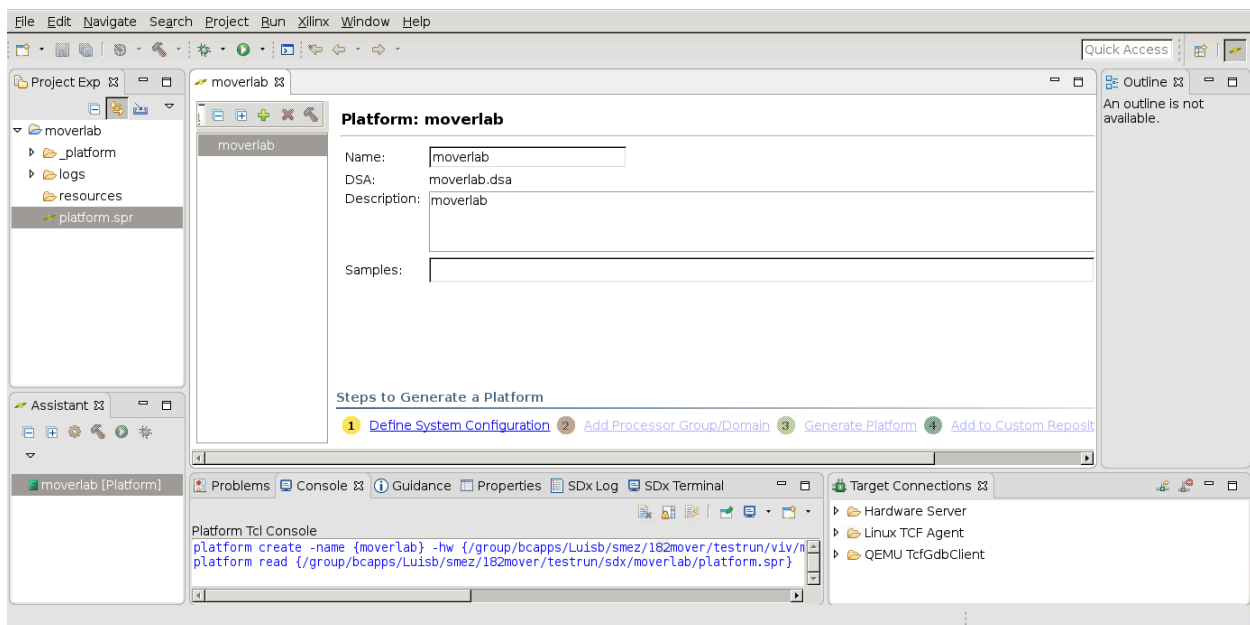
Hardware specification file (DSA/HDF):

[Browse...](#)

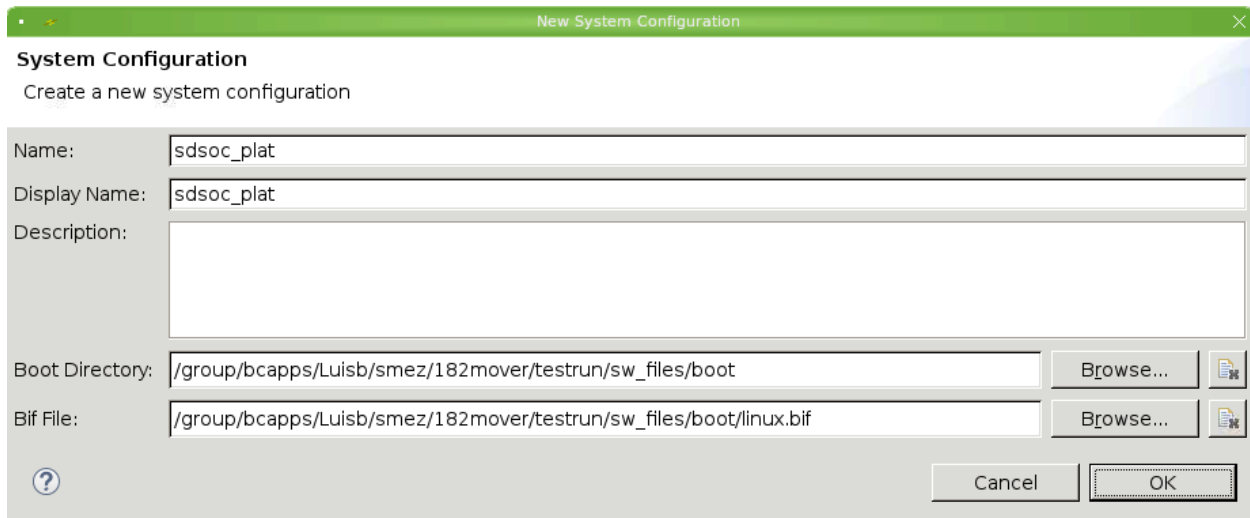
☒ Import software platform components

☐ Build software platform components

The following screen should be shown after beginning the Platform project:



Select the “Define System Configuration” step to select the boot files created from PetaLinux and also provide a Name. Then select OK.



New System Configuration

Create a new system configuration

Name: sdsoc_plat

Display Name: sdsoc_plat

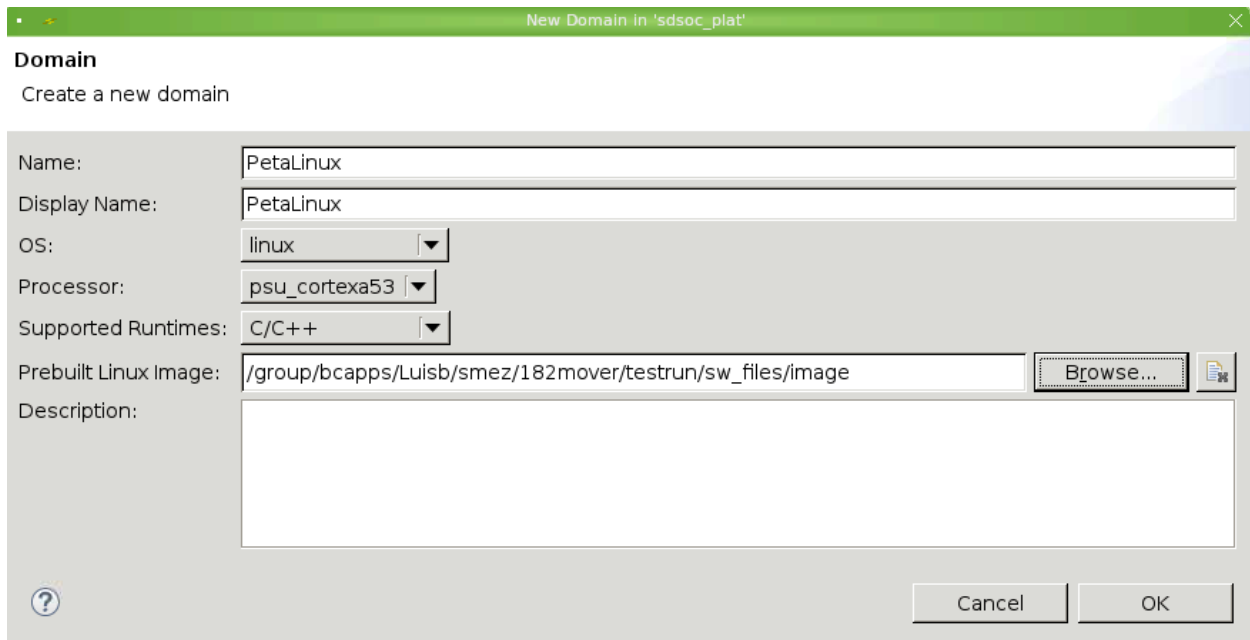
Description:

Boot Directory: /group/bcapps/Luisb/smez/182mover/testrun/sw_files/boot Browse...

Bif File: /group/bcapps/Luisb/smez/182mover/testrun/sw_files/boot/linux.bif Browse...

? Cancel OK

Now select the “Add Processor Group/Domain” to point to the fit image created from PetaLinux as shown below:



New Domain in 'sdsoc_plat'

Create a new domain

Name: PetaLinux

Display Name: PetaLinux

OS: linux

Processor: psu_cortexa53

Supported Runtimes: C/C++

Prebuilt Linux Image: /group/bcapps/Luisb/smez/182mover/testrun/sw_files/image Browse...

Description:

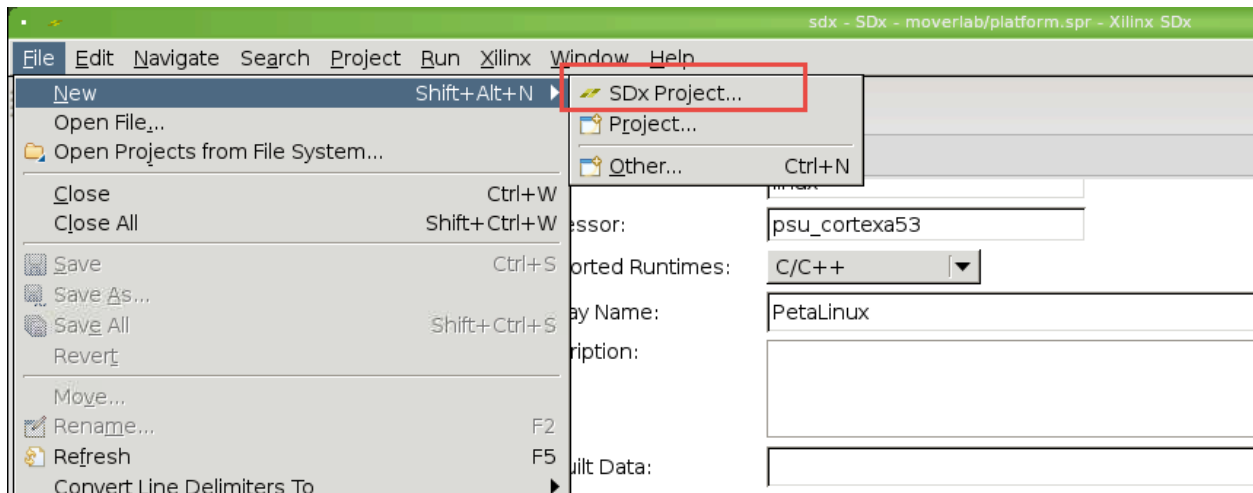
? Cancel OK

Lastly, select to “Generate Platform.” This will generate the platform and add it local to the project. Next, it’s time to create the application. Do not close SDx yet.

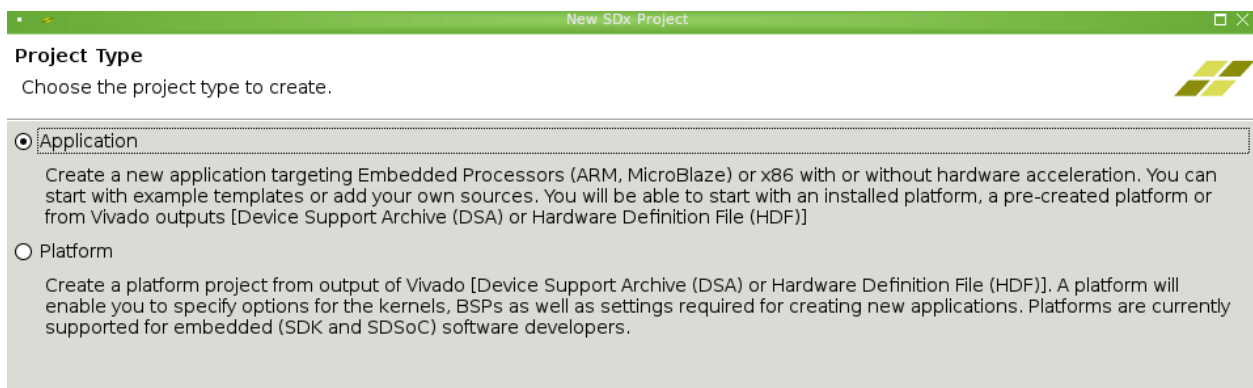
Step 5: SDSoc - Creating the Linux application

Now it’s time to create the Linux application with the ability to direct the streaming data in and out of the application. This particular application will take 64-bit input streaming data at 100MHz and send it directly to PL DDR. Then it will take that data from PL DDR into PS DDR. Lastly, it will take the data in PS DDR out to a 64-bit streaming output at 100Mhz.

While SDx is still open, create a new SDx Project by selecting File->New->SDx Project...



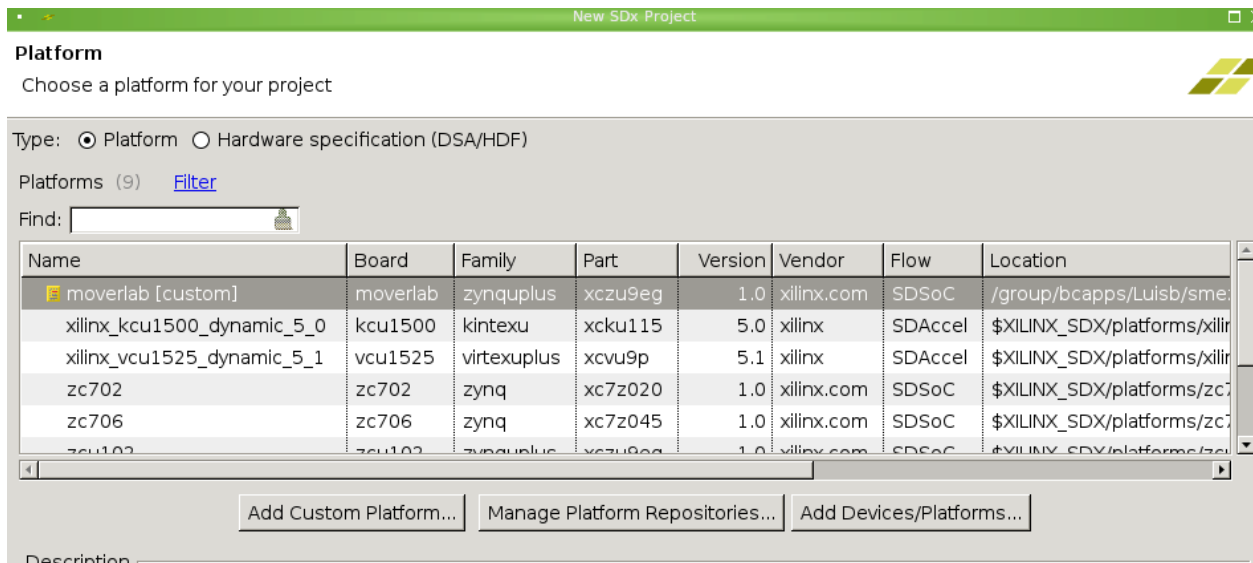
This time, instead of creating a new Platform project, create an Application:



Give the application a name. The name of the application is arbitrary and in this example, it is called "dmalinapp"

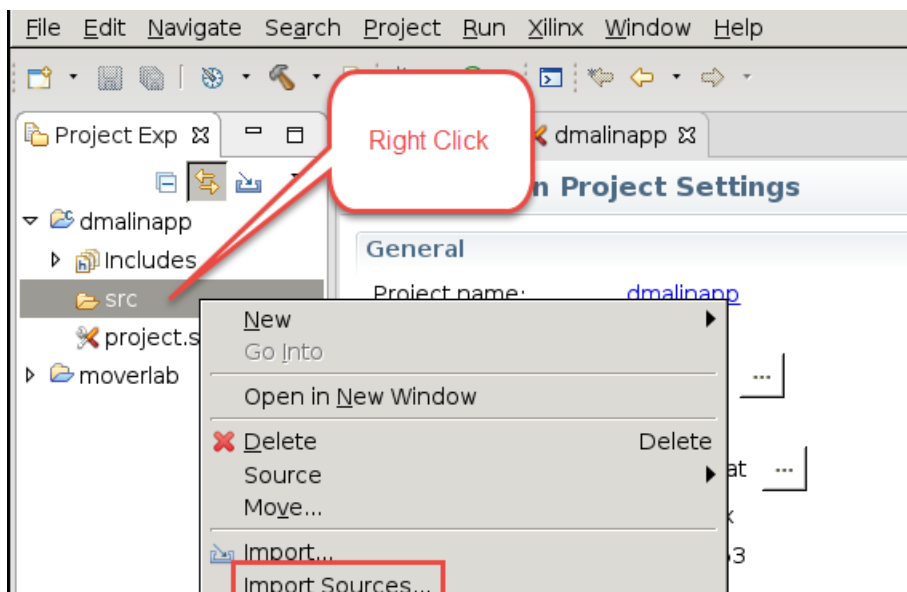


Select the custom platform that was recently create. It's is called moverlab:

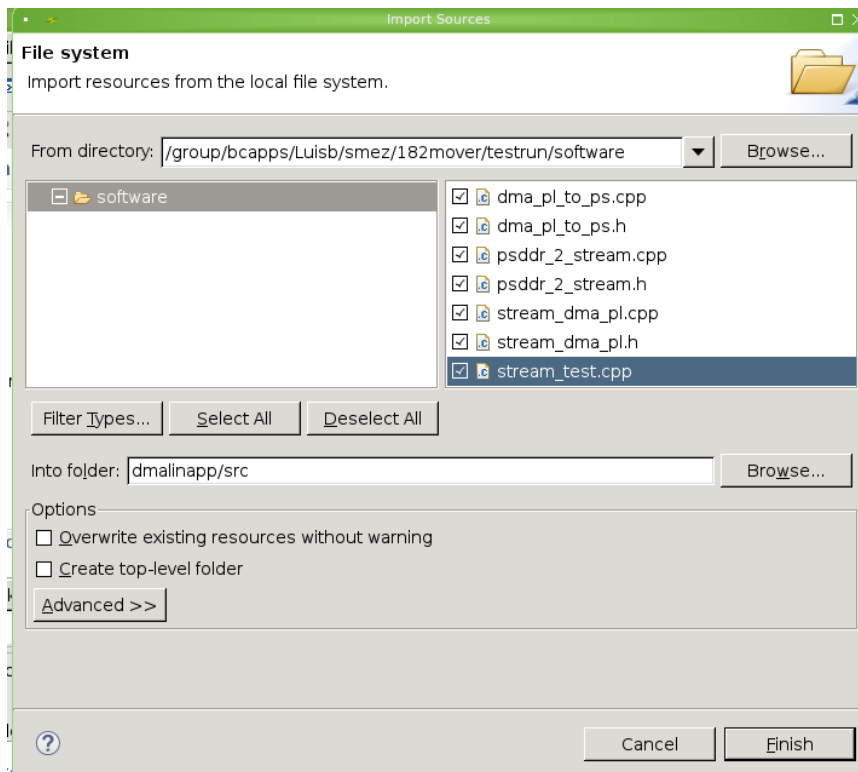


Select the defaults for “System Configuration” as well as the “Templates” and select “Finish”.

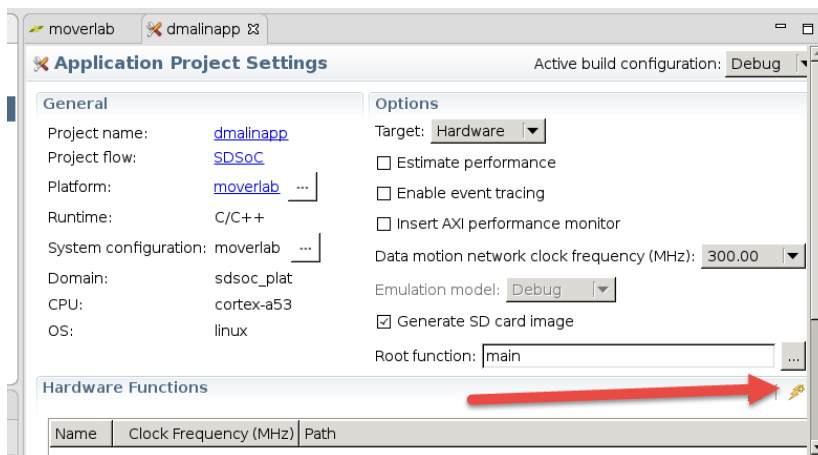
Right click the “src” directory in the Project Explorer and select to import sources:



Within the extracted zip file directory, locate the “software” directory and import all of the source files as shown below:

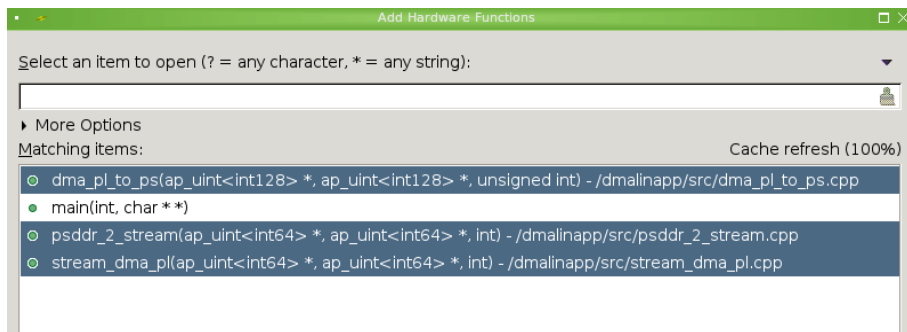


Under the Application Project Settings, select the lightning bolt button to accelerate the mover functions:

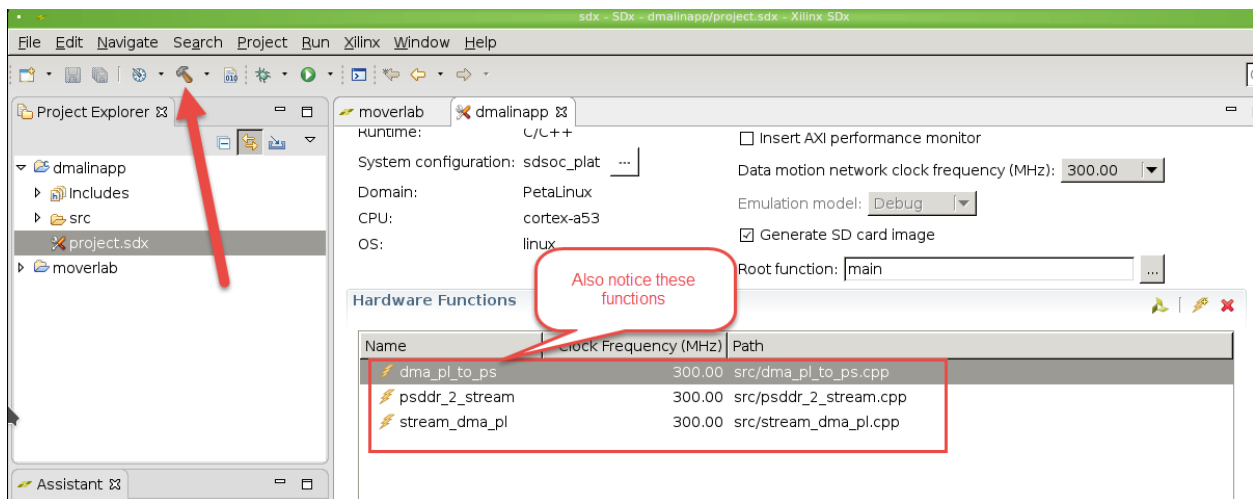


Select/Highlight the following hardware functions to be accelerated (these are the datamovers) as shown below. To select multiple functions, hold down CTRL (The GUI is slow to respond; so be patient)

- dma_pl_to_ps
- psddr_2_stream
- stream_dma_pl



Now it's time to build the project. Click on the hammer shown below and yell out "It's hammer time!"



Building this application will take some time. SDx will build new bits during this first build. After the first set up bits are created, it's very unlikely that the bits will need to be re-created as long as the hardware functions are not modified. So subsequent builds can be built quickly.

While SDSoc builds

As SDSoc builds the application, this is a good time to look at the source files. First inspect the stream_dma_pl.cpp and the stream_dma_pl.h. The stream_dma_pl.cpp shows the function being done with the stream_dma_pl call:

```

1 #include "stream_dma_pl.h"
2
3 void stream_dma_pl( ap_uint<64> *din, ap_uint<64> *doutmig, int num_elements)
4 {
5     #pragma HLS interface axis port=din
6     #pragma HLS interface m_axi port=doutmig offset=direct num_read_outstanding=8 num_write_outstanding=8 max_read_burst_length=256 max_write_burst_length=256
7     for(int i = 0; i < num_elements; i++) {
8         #pragma HLS pipeline
9         doutmig[i] = din[i];
10     }
11 }

```

This is a simple data mover that takes input data and send it right back out through another pointer. The number of beats is determined at the time the call is made. Notice the HLS pragmas; one is to indicate the port type while the other is a performance enhancement. Now look at the stream_dma_pl.h file to look at the special SDS pragmas:

```

2  #ifndef STREAM_DMA_PL_H
3  #define STREAM_DMA_PL_H
4
5  #include "ap_int.h"
6
7  #pragma SDS data sys_port(din:rx_stream_M_AXIS)
8  #pragma SDS data copy(din[0:num_elements])
9  #pragma SDS data access_pattern(din:SEQUENTIAL)
10
11 #pragma SDS data sys_port(doutmig:MIG)
12 #pragma SDS data mem_attribute(doutmig:NON_CACHEABLE|PHYSICAL_CONTIGUOUS)
13 #pragma SDS data zero_copy(doutmig[0:num_elements])
14
15 void stream_dma_pl(ap_uint<64> *din, ap_uint<64> *doutmig, int num_elements);
16
17 #endif
18

```

Notice on line 7 the sys_port pragma is mapping the din pointer to the rx_stream IP port that is the M_AXIS stream. This was determined from the script used to create the DSA.

Also notice that sys_port is used to map the dout pointer to the PL Memory controller. This is also referenced in the dsa.tcl where the memport was labeled MIG:

```

28 set mig_master []
29 for {set i 1} {$i < 16} {incr i} {
30     lappend mig_master S[format %02d $i]_AXI {memport "MIG" sptag "MIG0" memory "ddr4_0 C0_DDR4_ADDRESS_BLOCK"}
31 }
32 set_property PFM.AXI_PORT $mig_master [get_bd_cells /axi_smc]

```

Lastly notice the zero_copy pragma is used to tell SDSoC the stream_dma function will have the dout port as shared memory through an AXI master bus interface.

Next look in the stream_test.cpp file and notice how the plddr_buf and plddr_buf1 pointers are memory mapped to the physical MIG addresses:

```

52     unsigned short* plddr_buf;
53     unsigned short* plddr_buf1;
54
55     //PL Buffers
56     plddr_buf = (unsigned short *)sds_mmap((void *) (PL_DDR_ADDR ), BUF_SIZE, NULL);
57     plddr_buf1 = (unsigned short *)sds_mmap((void *) (PL_DDR_ADDR + BUF_SIZE), BUF_SIZE, NULL);
58

```

Also, look at the fifo pointer to see memory is allocated for the streaming source. In theory, this allocation isn't necessary, but I keep it there since it does no harm.

```

//Input Stream Sources
ap_uint<64> *fifo = (ap_uint<64> *)sds_alloc(BUF_SIZE);

```

Lastly, look at the call to stream_dma to see how the streaming data is captured to onto MIG:

```

82 // This will take streaming 64-bit data into PL DDR
83 #pragma SDS async(1)
84   stream_dma_pl( fifo, (ap_uint<64> *)plddr_buf, BUF_SIZE/8);
85 #pragma SDS async(1)
86   stream_dma_pl( fifo, (ap_uint<64> *)plddr_buf, BUF_SIZE/8);
87 #pragma SDS async(1)
88   stream_dma_pl( fifo, (ap_uint<64> *)plddr_buf1, BUF_SIZE/8);
89   sds_wait(1);
90   sds_wait(1);
91   sds_wait(1);
92

```

First thing to notice is that BUF_SIZE is divided by 8, this is because stream_dma_pl data types were 64-bit wide, which is 8 bytes. If you look at the loop for the data movement in the hardware function, it's based on beats. BUF_SIZE was intended for bytes counts; so that's why it's divided by 8.

Notice that there are two stream_dma_pl calls with the same plddr_buf passed. This is done to flush out the FIFO while it's overflowed since the counter has been running for a while now. The async call queues up the transaction in the adapter IP. So the first transaction is essentially thrown out.

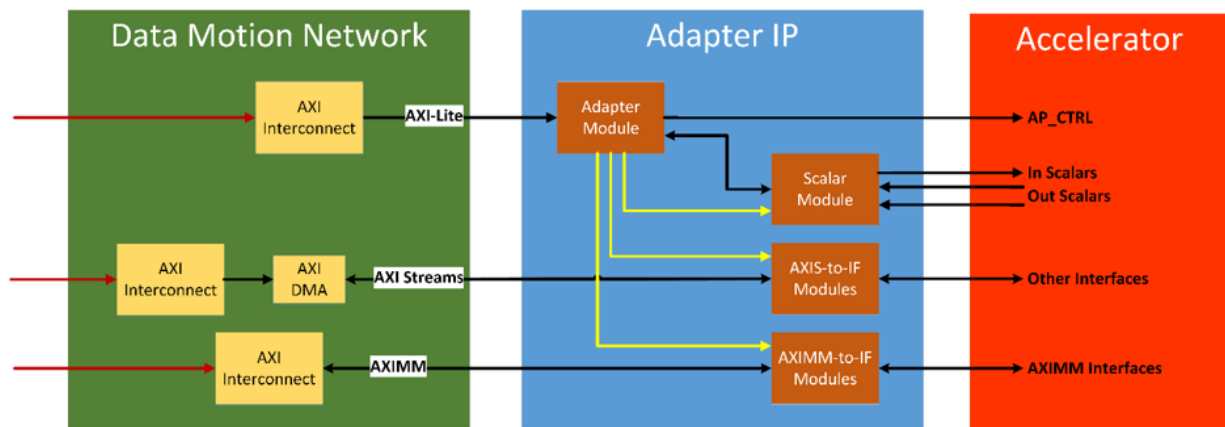


Figure 2. Adapter Configuration with Multi-Buffers

The sds_wait call simply waits for the transactions to complete.

Walk through the other accelerated functions to see how they were implemented. Reference the SDSoC User Guide for details.

Once SDSoC completes

Take the files in the /Debug/sd_card directory and copy them to your SD card. Now boot Linux and run the executable from /media/card/your_elf_name.elf and it should complete successfully. Now read the registers with devmem at address 0x80000000 to see if you have data incrementing every 8 bytes. You'll have to read them 4 bytes at a time. So you would read from address 0x80000000 and 0x80000004. You have successfully captured streaming data into Linux.

Using the ILA:

Within the SDx project, look in the following directory to find the ltx. This ltx will be used to debug.

Debug_sds\p0\vivado\prj\prj.runs\impl_1

With the ILA core, have fun and trigger on AXI Streaming transactions and compare them to what software displays. Remember to be mindful of data types. Once you see the data come out on the ILA, you have successfully transmitted streaming data from Linux to a streaming AXI master.