Explore more resources
Altera® Design Hub

altera
An Intel Company

FPGA AI Suite

SoC Design Example User Guide

Updated for FPGA AI Suite: **2024.3**

Online Version
Send Feedback

768979
2025.03.28

Contents

1. Publication Deprecation Notice

This publication is deprecated and will not be updated in the future.

For information about design examples for FPGA AI Suite Version 2024.3 and later, refer to the *FPGA AI Suite Design Examples User Guide*.

1.1. FPGA AI Suite SoC Design Example User Guide

The *FPGA AI Suite SoC Design Example User Guide* describes the design and implementation for accelerating AI inference using the FPGA AI Suite, Intel® Distribution of OpenVINO™ toolkit, and the following *Intel* FPGA development kits:

- Agilex™ 7 FPGA I-Series Transceiver-SoC Development Kit (DK-SI-AGI027FC)
- Arria® 10 SX SoC FPGA Development Kit (DK-SOC-10AS066S)

The following sections in this document describe the steps to build and execute the design:

- FPGA AI Suite SoC Design Example Quick Start Tutorial on page 9
- FPGA AI Suite SoC Design Example Run Process on page 31

The following sections in this document describe design decisions and architectural details about the design:

- FPGA AI Suite SoC Design Example Build Process on page 33
- FPGA AI Suite SoC Design Example Quartus Prime System Architecture on page 42
- FPGA AI Suite Soc Design Example Software Components on page 56
- Streaming-to-Memory (S2M) Streaming Demonstration on page 61

Use this document to help you understand how to create a SoC example design with the targeted FPGA AI Suite architecture.

About the FPGA AI Suite Documentation Library

Documentation for the FPGA AI Suite is split across a few publications. Use the following table to find the publication that contains the FPGA AI Suite information that you are looking for:

Table 1. FPGA AI Suite Documentation Library

Title and Description	
Release Notes Provides late-breaking information about the FPGA AI Suite including new features, important bug fixes, and known issues.	Link
Getting Started Guide Get up and running with the FPGA AI Suite by learning how to initialize your compiler environment and reviewing the various design examples and tutorials provided with the FPGA AI Suite	Link
IP Reference Manual Provides an overview of the FPGA AI Suite IP and the parameters you can set to customize it. This document also covers the FPGA AI Suite IP generation utility.	Link
Compiler Reference Manual Describes the use modes of the graph compiler (`dla_compiler`). It also provides details about the compiler command options and the format of compilation inputs and outputs.	Link
PCIe-based Design Example User Guide Describes the design and implementation for accelerating AI inference using the FPGA AI Suite, Intel Distribution of OpenVINO toolkit, and a Terasic* DE10-Agilex Development Board.	Link
SoC-based Design Example User Guide Describes the design and implementation for accelerating AI inference using the FPGA AI Suite, Intel Distribution of OpenVINO toolkit, and an Arria 10 SX SoC FPGA Development Kit (DK-SOC-10AS066S) or Agilex 7 FPGA I-Series Transceiver-SoC Development Kit.	Link

Intel Distribution of OpenVINO toolkit Requirement

To use the FPGA AI Suite, you must be familiar with the Intel Distribution of OpenVINO toolkit.

FPGA AI Suite Version 2024.3 requires the Intel Distribution of OpenVINO toolkit Version 2023.3 LTS. For OpenVINO documentation, refer to https://docs.openvino.ai/2023.3/documentation.html.

2. About the SoC Design Example

The FPGA AI Suite SoC design example shows how the Intel Distribution of OpenVINO toolkit and the FPGA AI Suite support the CPU-offload deep learning acceleration model in an embedded system

The SoC design examples are implemented with the following components:

- FPGA AI Suite IP
- Intel Distribution of OpenVINO toolkit
- The community-supported OpenVINO ARM plugin
- Sample hardware and software systems that illustrate the use of these components
- Arm*-Linux build scripts for the Arria 10 SX SoC and Agilex 7 I-Series SoC FPGA hard processor system (HPS) built using Yocto frameworks

For an easier initial experience, these design examples include prebuilt FPGA bitstreams and a Linux-compiled system image that correspond to pre-optimized FPGA AI Suite architecture files.

You can copy this disk-image to an SD card and insert the card into a supported FPGA development kit. Additionally, you can use the design example scripts to choose from a variety of architecture files and build (or rebuild) your own bitstreams, subject to IP licensing limitations.

SoC Design Example Execution Models

The SoC design example has two execution models:

- Memory-to-memory (M2M) execution model, which provides a `dla_benchmark` interface to the inference engine, similar to the PCIe-based design examples.

  For design details, refer to Memory-to-Memory (M2M) Variant Design on page 43.
- Streaming-to-memory (S2M) execution model that demonstrates a streaming data source

  For simplicity in this design example, the streaming data source is the SoC ARM CPU itself, which streams to a layout transform on the FPGA, but this design illustrates one of the suggested system architectures for any streaming source.

  For design details, refer to Streaming-to-Memory (S2M) Variant Design on page 45.

The design example is typically compiled for the S2M execution model, which supports both the M2M and S2M modes. A reduced functionality bitstream is also included as a compilation option, which supports only the M2M execution model.

The SoC design example has been optimized for simplicity, to create a flexible foundation that you can use to build more complex SoC designs.

2.1. FPGA AI Suite SoC Design Example Prerequisites

The SoC design example requires one of the following development kits:

- Agilex 7 FPGA I-Series Transceiver-SoC Development Kit (DK-SI-AGI027FC) (Ordering code DK-SI-AGI027FA or DK-SI-AGI027FC)

  This development kit features an Agilex 7 I-Series SoC device with 4 F-Tiles (OPN: AGIB027R31B1E1V).

  Important: The FPGA AI Suite requires DDR4 memory with an x8 or higher component data width. The RAM device provided with the Agilex 7 FPGA I-Series Transceiver-SoC Development Kit provides only a x4 component data width. For more details and recommended RAM modules, refer to Agilex 7 FPGA I-Series Transceiver-SoC Development Kit Hardware Requirements on page 7.

  For more details about this development kit, refer to the following URL: https://www.intel.com/content/www/us/en/products/details/fpga/development-kits/agilex/si-agi027.html
- Arria 10 SX SoC FPGA Development Kit (DK-SOC-10AS066S)

  This development kit features an Arria 10 SX 660 device (OPN: 10AS066N3F40E2SG) with a " -2" speed grade with the included DDR4 HILO memory cards.

  For more details about this development kit, refer to the following URL:

  https://www.intel.com/content/www/us/en/products/details/fpga/development-kits/arria/10-sx.html

In addition, the following hardware components are required:

- SDHC flash card, class 10 speed or faster (minimum 2 GB but 4 GB or more is recommended)
- Mini-USB cable suitable for connecting the development board to a host PC
- Ethernet cable suitable for connecting the development board to a network to provide access from a host PC on the same network

The host PC must use a supported operating system (Red Hat* Enterprise Linux* 8, Ubuntu* 20.04, or Ubuntu 22.04), and must have an internet connection to install the software dependencies.

To build bitstreams, Quartus® Prime Pro Edition Version 24.3 must be installed on the host system.

Although the development host system does not need to be the same as the system used to build packages and bitstreams, this guide does not explicitly cover the scenario where they are distinct.

2.1.1. Agilex 7 FPGA I-Series Transceiver-SoC Development Kit Hardware Requirements

The FPGA AI Suite SoC design example requires x8 (or wider) DDR4 memory.

The RAM module provided with the Agilex 7 FPGA I-Series Transceiver-SoC Development Kit does not support the FPGA AI Suite SoC design example because the included RAM module provides only an x4 width.

The design example has been verified on a development kit fitted with a Kingston* x8 RDIMM (KSM32RS8/16MFR). Intel recommends using this memory module to help you successfully use the design example.

3. FPGA AI Suite SoC Design Example Quick Start Tutorial

The SoC design example quick start tutorial provides instructions to do the following tasks:

- Build a bitstream and flash card image for the FPGA development kit.
- Run the `dla_benchmark` utility from the example runtime on the SoC FPGA HPS (Arm processor) host. This example runtime uses the memory-to-memory (M2M) model.
- Run the streaming image application that streams data from the HPS Arm processor host to the FPGA device in a way that mimics how data is streamed from any other input source (such as Ethernet, HDMI, or MIPI). This streaming image application uses the steaming-to-memory (S2M) model.

This quick start tutorial assumes that you have reviewed the following sections in the *FPGA AI Suite Getting Started Guide*:

- About the FPGA AI Suite
- Installing the FPGA AI Suite

SoC Design Example Quick Start Tutorial Prerequisites

Before you start the tutorial ensure that you have successfully completed the installation tasks outlined in "Installing the FPGA AI Suite Compiler and IP Generation Tools" in the *FPGA AI Suite Getting Started Guide*.

The remaining sections of the *FPGA AI Suite Getting Started Guide* can help you understand the overall flow of using the FPGA AI Suite, but they are not required to complete this quick start tutorial

3.1. Initial Setup

The quick start tutorial instructions assume that you have initialized your environment for the FPGA AI Suite with the `init_env.sh` script from a shell that is compatible with the Bourne shell (`sh`).

The FPGA AI Suite `init_env.sh` script might already be part of your shell login script. If not, then use the following command to initialize your shell environment:

```
source /opt/intel/fpga_ai_suite_2024.3/dla/bin/init_env.sh
```

This command assumes that the FPGA AI Suite is installed in the default location. If you are using an FPGA AI Suite version other than 2024.3, adjust the path to script accordingly.

3.2. Initializing a Work Directory

While you can build the design example directly in the $COREDLA_ROOT location, it is better to use a work directory. You can create a work directory as follows:

```
mkdir ~/coredla_work
cd ~/coredla_work
source dla_init_local_directory.sh
```

If you created a work directory while following the instructions in the *FPGA AI Suite Getting Started Guide*, the `dla_init_local_directory.sh` script prompts you to use the `coredla_work.sh` script instead to set the $COREDLA_WORK environment variable.

3.3. (Optional) Create an SD Card Image (`.wic`)

An SD card provides the FPGA bitstream and HPS disk image to the SoC FPGA development kit. You can build your own SD card image or use the prebuilt image provided by the FPGA AI Suite SoC design example.

If you want to use the prebuilt image, skip this section and go to Writing the SD Card Image (.wic) to an SD Card on page 15.

Important: You cannot build the SD card as `root` due to security checks in the BitBake tool used when creating an SD card image.

3.3.1. Installing Prerequisite Software for Building an SD Card Image

Building the SD card image requires the following additional software:

- Quartus Prime Pro Edition Version 24.3
- Ashling* RiscFree* IDE for Intel FPGAs
- (Ubuntu only) Ubuntu package `libncurses5`

If you did not install Quartus Prime Pro Edition Version 24.3 when following the instructions in the *FPGA AI Suite Getting Started Guide*, you must install it now.

Building the SD card image also requires tools provided by Ashling RiscFree IDE for Intel FPGAs. You can install Ashling RiscFree IDE from a separate installation package or part of your Quartus Prime bundled installation package.

You can download the required software from the following URL: https://www.intel.com/content/www/us/en/software-kit/790039/intel-quartus-prime-pro-edition-design-software-version-23-3-for-linux.html.

To install the prerequisite software for building an SD card image:

1. Install Quartus Prime Pro Edition and Ashling RiscFree IDE for Intel FPGAs
2. (Ubuntu only) Install Ubuntu package `libcurses` with the following command:

   ```
   sudo apt install libncurses5
   ```

3. Ensure that the QUARTUS_ROOTDIR environment variable is set properly:

   ```
   echo $QUARTUS_ROOTDIR
   ```

If the QUARTUS_ROODIR is not set, run the following command:

```
export QUARTUS_ROOTDIR=/opt/intel/intelFPGA_pro/24.3/quartus
```

If you chose install Quartus Prime in a location other than the default location, adjust the path in `export` command to match your Quartus Prime installation location

4. Ensure your $PATH environment variable includes paths to the installed Quartus Prime and Ashling RiscFree IDE binaries. Adjust the following commands appropriately if you did not install into the default location:

```
export PATH=$PATH:/opt/intel/intelFPGA_pro/24.3/niosv/bin
export PATH=$PATH:/opt/intel/intelFPGA_pro/24.3/nios2eds/bin
export PATH=$PATH:/opt/intel/intelFPGA_pro/24.3/riscfree/toolchain/riscv32-unknown-elf/bin
```

5. Confirm that Quartus Prime Pro Edition Version 24.3 is installed by running the following command:

```
quartus_cmd -v
```

Related Information

- Ashling RiscFree IDE for Intel FPGAs User Guide
- Intel FPGA Software Installation and Licensing

3.3.2. Building the FPGA Bitstreams

The FPGA AI Suite SoC design example also includes prebuilt demonstration FPGA bitstreams. If you want to use the prebuilt demonstration bitstreams in your SD card image, skip ahead to Installing HPS Disk Image Build Prerequisites on page 12.

If you build your own bitstreams and do not have an FPGA AI Suite IP license, then your bitstream have a limit of 10000 inferences. After 10000 inferences, the unlicensed IP refuses to perform any additional inference. To reset the limit, reprogram the FPGA device.

Building the FPGA Bitstreams for the Agilex 7 FPGA I-Series Transceiver-SoC Development Kit

To build the FPGA bitstreams for the Agilex 7 FPGA I-Series Transceiver-SoC Development Kit, run the following commands:

```
dla_build_example_design.py \
 -ed 4_AGX7_S2M \
 -n 1 \
 -a $COREDLA_ROOT/example_architectures/AGX7_Performance.arch \
 --build \
 --build-dir $COREDLA_WORK/agx7_perf_bitstream \
 --output-dir $COREDLA_WORK/agx7_perf_bitstream
```

The bitstreams built by these commands support both the M2M execution model and the S2M execution model.

Building the FPGA Bitstreams for the Arria 10 SX SoC FPGA Development Kit

To build the FPGA bitstreams for the Arria 10 SX SoC FPGA Development Kit, run the following commands:

```
dla_build_example_design.py \
 -ed 4_A10_S2M \
 -n 1 \
 -a $COREDLA_ROOT/example_architectures/A10_Performance.arch \
 --build \
 --build-dir $COREDLA_WORK/a10_perf_bitstream \
 --output-dir $COREDLA_WORK/a10_perf_bitstream
```

The bitstreams built by these commands support both the M2M execution model and the S2M execution model.

3.3.3. Installing HPS Disk Image Build Prerequisites

The process to build the HPS disk image has additional prerequisites. To install these prerequisites, follow the instructions for your operating system in the following sections:

- Red Hat Enterprise Linux 8 Prerequisites on page 12
- Ubuntu 20 Prerequisites on page 13
- Ubuntu 22 Prerequisites on page 13

Red Hat Enterprise Linux 8 Prerequisites

To install the prerequisites for Red Hat Enterprise Linux 8:

1. Enable additional Red Hat Enterprise Linux 8 repository and package manager:

```
sudo subscription-manager repos \
  --enable codeready-builder-for-rhel-8-x86_64-rpms
wget https://dl.fedoraproject.org/pub/epel/epel-release-latest-8.noarch.rpm
sudo dnf install ./epel-release-latest-8.noarch.rpm epel-release
sudo dnf upgrade
```

2. Install the dependency packages:

```
sudo dnf install gawk wget git diffstat unzip texinfo gcc gcc-c++ make \
chrpath socat cpio python3 python3-pexpect xz iputils python3-jinja2 \
mesa-libEGL SDL xterm python3-subunit rpcgen zstd lz4 perl-open.noarch \
perl-Thread-Queue
```

3. Install packages required to create the flash card image and FPGA AI Suite runtime and dependencies:

```
cd /tmp
mkdir uboot_tools && cd uboot_tools
wget https://kojipkgs.fedoraproject.org/\
vol/fedora_koji_archive02/packages/uboot-tools/2018.03/3.fc28/x86_64/\
uboot-tools-2018.03-3.fc28.x86_64.rpm
sudo dnf install ./uboot-tools-2018.03-3.fc28.x86_64.rpm
sudo dnf install ninja-build fakeroot
sudo python3 -m pip install pylint passlib scons
```

4. Install CMake Version 3.16.3 or later:

```
sudo dnf install openssl-devel
cd /tmp
mkdir cmake && cd cmake
wget https://github.com/Kitware/CMake/releases/\
download/v3.24.3/cmake-3.24.3.tar.gz
```

```
tar zxf cmake-3*tar.gz
cd cmake-3.24.3
./bootstrap --prefix=/usr
make
sudo make install
```

5. Install Make Version 4.3 or later:

```
cd /tmp
mkdir make && cd make
wget https://ftp.gnu.org/gnu/make/make-4.3.tar.gz
tar xvf make-4.3.tar.gz
cd make-4.3
./configure
make
sudo make install
```

6. Add the `/sbin` directory to your $PATH environment variable:

```
export PATH="/sbin:$PATH"
```

Ubuntu 20 Prerequisites

To install the prerequisites for Ubuntu 20:

1. Install the dependency packages:

```
sudo apt install gawk wget git diffstat unzip texinfo gcc build-essential \
chrpath socat cpio python3 python3-pip python3-pexpect xz-utils debianutils \
iputils-ping python3-git python3-jinja2 libegl1-mesa libsdl1.2-dev pylint3 \
xterm python3-subunit mesa-common-dev zstd liblz4-tool device-tree-compiler \
mtools
```

2. Install packages required to create the flash card image and FPGA AI Suite runtime and dependencies:

```
sudo apt install ninja-build u-boot-tools scons fakeroot
```

3. Add the `/sbin` directory to your $PATH environment variable:

```
export PATH="/sbin:$PATH"
```

Ubuntu 22 Prerequisites

To install the prerequisites for Ubuntu 22:

1. Install the dependency packages:

```
sudo apt install gawk wget git diffstat unzip texinfo gcc build-essential \
chrpath socat cpio python3 python3-pip python3-pexpect xz-utils debianutils \
iputils-ping python3-git python3-jinja2 libegl1-mesa libsdl1.2-dev xterm \
python3-subunit mesa-common-dev zstd liblz4-tool device-tree-compiler mtools
```

2. Install packages required to create the flash card image and FPGA AI Suite runtime and dependencies:

```
sudo apt install ninja-build u-boot-tools scons fakeroot
```

3. Add the `/sbin` directory to your $PATH environment variable:

```
export PATH="/sbin:$PATH"
```

3.3.4. (Optional) Downloading the ImageNet Categories

By default, the S2M streaming app prints the category associated with each image after inference.

Optionally, you can use human-readable category names as follows:

1. Download the list of ImageNet categories from a source such as the following URL:

```
https://github.com/xmartlabs/caffeflow/blob/master/examples/imagenet/
imagenet-classes.txt
```

2. Place the contents into the following file:

```
$COREDLA_WORK/runtime/streaming/streaming_inference_app/categories.txt
```

3.3.5. Building the SD Card Image

The SD card image contains a Yocto Project embedded Linux system, HPS packages, and the FPGA AI Suite runtime.

Building the SD card image requires a minimum of 100GB of free disk space.

The SD card image is build with the `create_hps_image.sh` command, which does the following steps for you:

- Build a Yocto Project embedded Linux system.
- Build additional packages required by the SoC design example runtime, including the OpenVINO and OpenCV runtimes.
- Build the design example runtime.
- Combine all of these items and FPGA bitstreams into an SD card image using `wic`.
- Place the SD card image in the specified output directory.

For more details about the `create_hps_image.sh` command, refer to Building the Bootable SD Card Image (.wic) on page 37.

To build the SD card image, run one of the following commands:

- Agilex 7 FPGA I-Series Transceiver-SoC Development Kit

```
cd $COREDLA_WORK/runtime
./create_hps_image.sh \
   -f $COREDLA_WORK/agx7_perf_bitstream/hw/output_files \
   -o <output_dir> -u \
   -m agilex7_dk_si_agi027fa
```

- Arria 10 SX SoC FPGA Development Kit

```
cd $COREDLA_WORK/runtime
./create_hps_image.sh \
   -f $COREDLA_WORK/a10_perf_bitstream/hw/output_files \
   -o <output_dir> -u \
   -m arria10
```

If the command returns errors such as "`bitbake: command not found`", try deleting the `$COREDLA_WORK/runtime/build_Yocto/` directory before rerunning the `create_hps_image.sh` command.

3.4. Writing the SD Card Image (.wic) to an SD Card

Before running the demonstration, you must create a bootable SD card for the FPGA development kit. You can use either the precompiled SD card image or an SD card image that you created.

The precompiled SD card image (.wic) is in the following location:

- Agilex 7 FPGA I-Series Transceiver-SoC Development Kit

```
$COREDLA_ROOT/demo/ed4/agx7_soc_s2m/sd-card/coredla-image-
agilex7_dk_si_agi027fa.wic
```

- Arria 10 SX SoC FPGA Development Kit

```
$COREDLA_ROOT/demo/ed4/a10_soc_s2m/sd-card/coredla-image-arria10.wic
```

If you built your own SD card image following the instructions in (Optional) Create an SD Card Image (.wic) on page 10, then your SD card image is located in the directory that you specified for the -o option of the create_hps_image.sh command.

To write the SD card image to an SD card:

1. Determine the device associated with the SD card on the host by running the following command before and after inserting the SD card:

```
cat /proc/partitions
```

   Typical locations for the SD card include /dev/sdb or /dev/sdc. The rest of these instruction use /dev/sdx as the SD card location.

2. Use the dd command to write the SD card image as follows:

```
wic_image=<path to SD (.wic) image file>
sudo umount /dev/sdx*
sudo dd if=$wic_image of=/dev/sdx bs=1M
sudo sync
sudo udisksctl power-off -b /dev/sdx
```

After the SD card image is written, insert the SD card into the development kit SD card slot.

If you want to use a Microsoft* Windows* system to write the SD card image to the SD card, refer to the GSRD manuals available at the following URL: https://www.rocketboards.org/foswiki/Documentation/GSRD.

3.5. Preparing SoC FPGA Development Kits for the FPGA AI Suite SoC Design Example

To prepare an FPGA development kit for the FPGA AI Suite SoC design example:

1. Prepare one of the supported development kits:
   - Prepare the Agilex 7 FPGA I-Series Transceiver-SoC Development Kit.
   - Prepare the Arria 10 SX SoC FPGA Development Kit (DK-SOC-10AS066S).
2. Configure the SoC FPGA development kit UART connection.
3. Determine the SoC FPGA development kit IP address.

3.5.1. Preparing the Agilex 7 FPGA I-Series Transceiver-SoC Development Kit

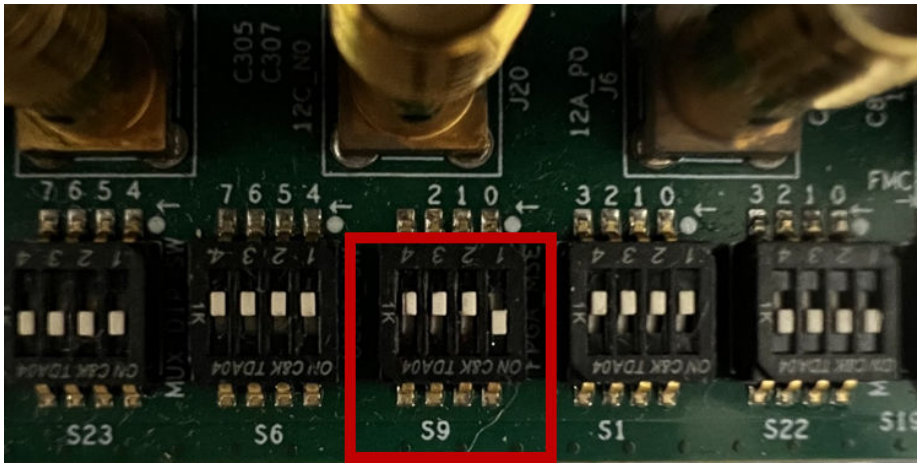
To prepare the Agilex 7 FPGA I-Series Transceiver-SoC Development Kit for the FPGA AI Suite SoC design example:

1. Confirming Agilex 7 FPGA I-Series Transceiver-SoC Development Kit Board Set Up on page 16
2. Programming the Agilex 7FPGA Device with the JTAG Indirect Configuration (.jic) File on page 17.
3. Connecting the Agilex 7 FPGA I-Series Transceiver-SoC Development Kit to the Host Development System on page 19

3.5.1.1. Confirming Agilex 7 FPGA I-Series Transceiver-SoC Development Kit Board Set Up

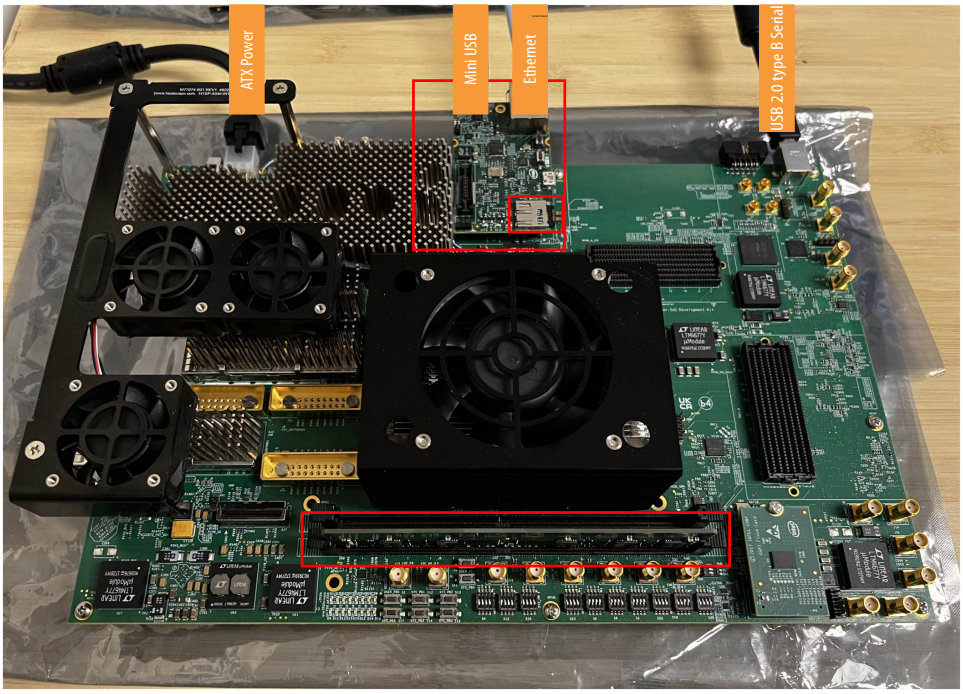
Confirm the board settings as follows:

1. Ensure that the Agilex 7 FPGA I-Series Transceiver-SoC Development Kit has the required DIP switch and jumper settings. The SoC example design requires that all DIP switches have their default settings except for S9 where switch 1 is ON and switches 2,3, and 4 are OFF:



   For more details about default DIP switch and jumper settings, refer to *Agilex 7 FPGA I-Series Transceiver-SoC Development Kit User Guide*.
2. Ensure that the HPS IO48 OOBE daughter card is installed in connector J4 on the development kit, and the SD card with the programmed Yocto image is installed in the daughter card.
3. Ensure that the DDR4 x8 RDIMM is installed in the PCIe slot furthest from the fan. For RDIMM requirements, refer to Agilex 7 FPGA I-Series Transceiver-SoC Development Kit Hardware Requirements on page 7.

When configured and connected, the Agilex 7 FPGA I-Series Transceiver-SoC Development Kit should resemble the following image:



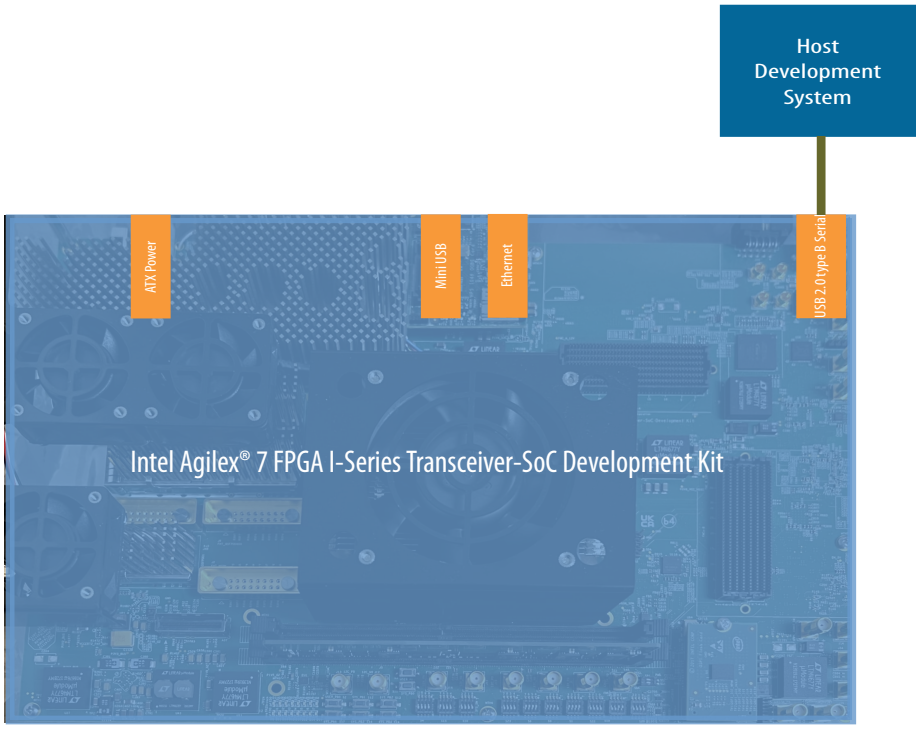
The board connections serve the following purposes:

- The USB 2.0 connector is used to program the FPGA device.
- The Ethernet connector is used for fast data transfer to the HPS.
- The micro USB connector is used to monitor the serial output from, and provide command-line input to the HPS during operation.

3.5.1.2. Programming the Agilex 7FPGA Device with the JTAG Indirect Configuration (`.jic`) File

To program the Agilex 7 FPGA device with the JTAG indirect configuration (`.jic`) file:

1. Connect the Agilex 7 FPGA I-Series Transceiver-SoC Development Kit to your host development system via USB 2.0 connection as shown in the following diagram:



2. Program the QSPI with the `.jic` file by running the following commands on the host development system:

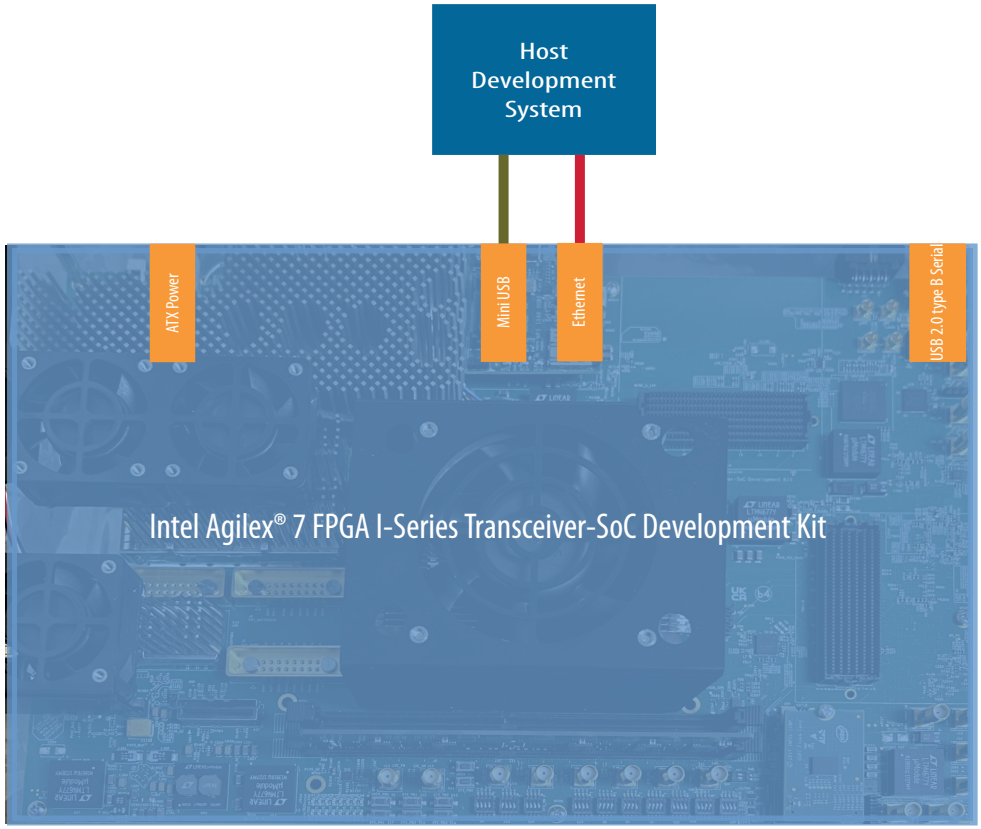
```
cd $COREDLA_ROOT/demo/ed4/agx7_soc_s2m/sd-card/
quartus_pgm -m jtag -o "pvi;u-boot-spl-dtb.hex.jic@<device_number>"
```

where *<device_number>* is 1 or 2, depending on whether the HPS is already running (that is, the prior state of the device). Use 1 if the HPS is not running, and 2 if the HPS is already running. If you do not know the state of the device, try 1. If that fails, try 2.

The Agilex 7FPGA device is configured from the QSPI flash at boot time.

3.5.1.3. Connecting the Agilex 7 FPGA I-Series Transceiver-SoC Development Kit to the Host Development System

Connect the Agilex 7 FPGA I-Series Transceiver-SoC Development Kit to your host development system via Ethernet and USB UART connections as shown in the following diagram:



3.5.2. Preparing the Arria 10 SX SoC FPGA Development Kit (DK-SOC-10AS066S)

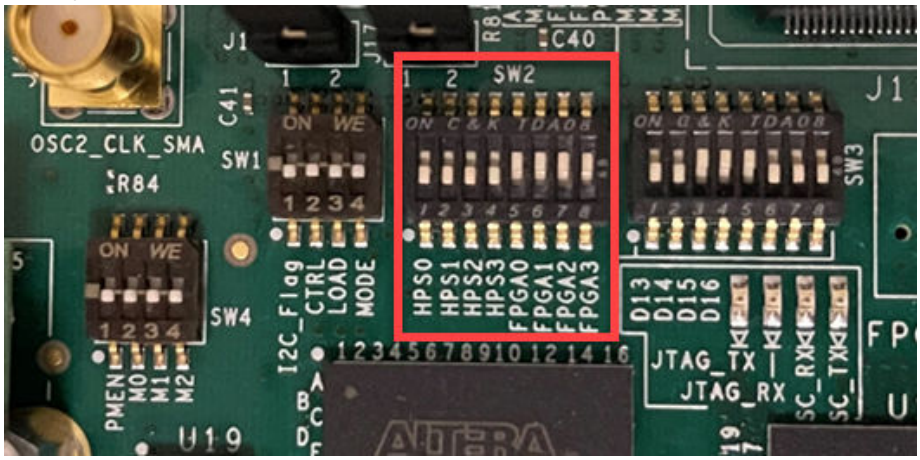
To prepare the Arria 10 SX SoC FPGA Development Kit for the FPGA AI Suite SoC design example:

1. Confirming Arria 10 SX SoC FPGA Development Kit (DK-SOC-10AS066S) Board Settings on page 19
2. Connecting the Arria 10 SX SoC FPGA Development Kit (DK-SOC-10AS066S) to the Host Development System on page 22.

3.5.2.1. Confirming Arria 10 SX SoC FPGA Development Kit (DK-SOC-10AS066S) Board Settings

Confirm the board settings as follows:

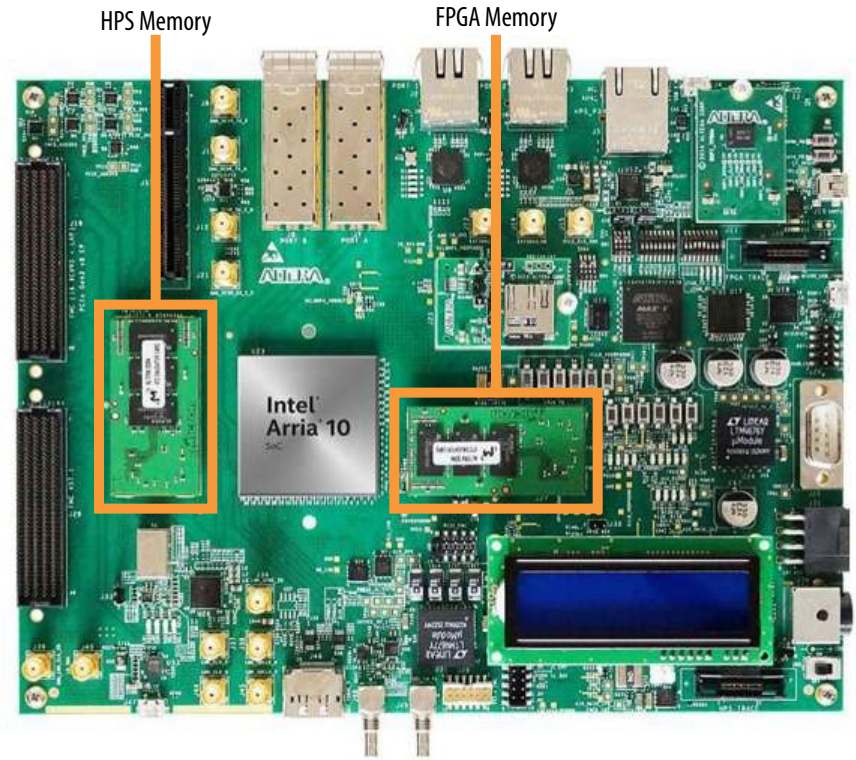
1. Ensure that the Arria 10 SX SoC FPGA Development Kit (DK-SOC-10AS066S) has the required DIP switch and jumper settings. The SoC example design requires that all DIP switches have their default settings except for SW2 switches 5, 6, 7, and 8, which should be switched ON:



   For more details about default DIP switch and jumper settings, refer to *Arria 10 SoC Development Kit User Guide*.

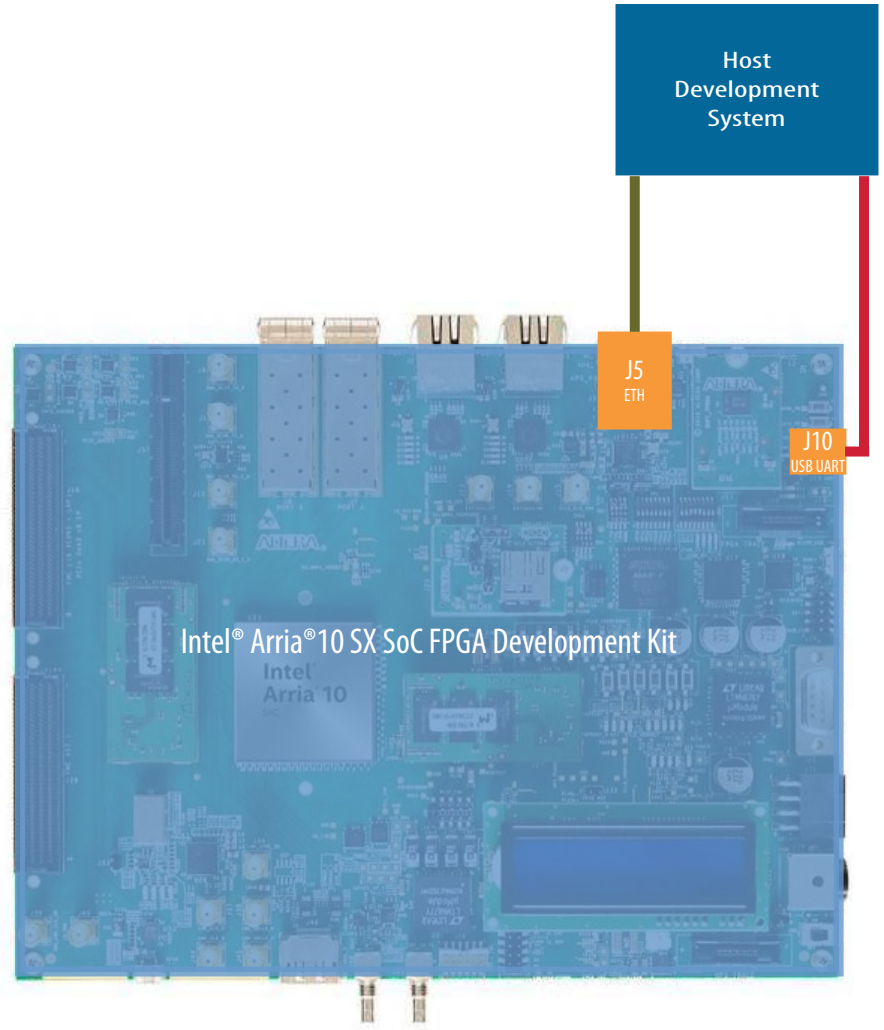
2. Ensure that the HILO cards are fitted correctly.

The Arria 10 SX SoC FPGA Development Kit (DK-SOC-10AS066S) includes two DDR4 HILO cards: the HPS memory (1GB) and the FPGA memory (2GB). Both the HPS Memory and FPGA Memory DDR4 HILO modules must be fitted as shown in the following image:



3.5.2.2. Connecting the Arria 10 SX SoC FPGA Development Kit (DK-SOC-10AS066S) to the Host Development System

Connect the Arria 10 SX SoC FPGA Development Kit (DK-SOC-10AS066S) to your host development system via Ethernet and USB UART connections as shown in the following diagram:



3.5.3. Configuring the SoC FPGA Development Kit UART Connection

The SoC FPGA development kit boards have USB-to-serial converters that allows the host computer to see the board as a virtual serial port:

- The Arria 10 SX SoC FPGA Development Kit (DK-SOC-10AS066S) has a FTDI USB-to-serial converter chip.
- The Agilex 7 FPGA I-Series Transceiver-SoC Development Kit has a USB-to-serial converter on the IO48 daughter card.

Ubuntu, Red Hat Enterprise Linux, and other modern Linux distributions have built-in drivers for the FTDI USB-to-serial converter chip, so no driver installation is necessary on those platforms.

On Microsoft Windows, the Windows SoC EDS installer automatically installs the necessary drivers. For details, see the SoC GSRD for your SoC FPGA development kit at the following URL: https://www.rocketboards.org/foswiki/Documentation/GSRD

The serial communication parameters are as follows:

- Baud rate: 115200
- Parity: None
- Flow control: None
- Stop bits: 1

On Windows, you can use utilities such as TeraTerm or PuTTY to connect the board. You can configure these utilities from their tool menus.

On Linux, you can use the Minicom utility. Configure the Minicom utility as follows:

1. Determine the device name associated with the virtual serial port on your host development system. The virtual serial port is typically named `/dev/ttyUSB0`.
   a. Before connecting the mini USB cable to the SoC FPGA development kit, determine which USB serial devices are installed with the following command:
      ```
      ls /dev/ttyUSB*
      ```
   b. Connect the mini USB cable from the SoC FPGA development kit to the host development system.
   c. Confirm the new device connection with the `ls` command again:
      ```
      ls /dev/ttyUSB*
      ```
2. If you do not have the Minicom application installed on the host development system, install it now.
   - OnRed Hat Enterprise Linux 8: `sudo yum install minicom`
   - On Ubuntu: `use sudo apt-get install minicom`
3. Configure Minicom as follows:
   a. Start Minicom:
      ```
      sudo minicom -s
      ```
   b. Under **Serial Port Setup** choose the following:
      - Serial Device: **/dev/ttyUSB0** (Change this value to match the system value that you found earlier, if needed)
      - Bps/Par/Bits: **115200 8N1**
      - Hardware Flow Control: **No**
      - Software Flow Control: **No**

      Press **[ESC]** to return to the main configuration menu.
   c. Select **Save Setup as dfl** to save the default setup. Then select **Exit**.

3.5.4. Determining the SoC FPGA Development Kit IP Address

To determine the FPGA development kit IP address:

1. Open a Terminal session to the FPGA development kit via the UART connection and log in using the user name `root` and no password.

```
         Starting Record Runlevel Change in UTMP…
[  OK  ] Finished Record Runlevel Change in UTMP.
     9.553286] random: crng init done
[  OK  ] Finished Load/Save Random Seed.
[   10.084845] socfpga-dwmax ff800000.ethernet eth0: Link is Up – 1Gbps/Full
– flow control off
[   10.103287] IPv6: ADDRCONF(NETDEV_CHANGE): eth0: link becomes ready

Poky (Yocto Project Reference Distro) 4.0.2 arria10-62747948036a ttyS0

arria10-62747948036a login:
```

2. Issue a `hostname` command to display the network name of the FPGA development kit board:

```
root@arria10-62747948036a:~# hostname
arria10-62747948036a
root@arria10-62747948036a:~#
```

   In this example, the network name of the board is `arria10-62747948036a`.

   Tip: You need this hostname later on to open an SSH connection to the FPGA development kit.

3. Confirm that you have a connection to the development kit from the development host with the `ping` command. Append the `.local` to the host name when you issue the `ping` command:

```
build-host:$ ping arria10-62747948036a.local -c4
PING arria10-62747948036a (192.168.0.23) 56(84) bytes of data.
64 bytes of data from arria10-62747948036a (192.168.0.23): icmp_seq=1 ttl=63
time=1.66ms
64 bytes of data from arria10-62747948036a (192.168.0.23): icmp_seq=1 ttl=63
time=1.66ms
64 bytes of data from arria10-62747948036a (192.168.0.23): icmp_seq=1 ttl=63
time=1.66ms
64 bytes of data from arria10-62747948036a (192.168.0.23): icmp_seq=1 ttl=63
time=1.66ms

--- arria10-62747948036a ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3005ms
rtt min/avg/max/mdev = 1.664/2.037/2.283/0.238 ms
```

You can use the host name when you need to transfer files to the running system by appending the `.local` to the host name. For example, for the host name `arria10-62747948036a`, you can use `arria10-62747948036a.local`.

3.6. Adding Compiled Graphs (AOT files) to the SD Card

An AOT file contains instructions for the FPGA AI Suite IP to "execute" in order to perform inference. The M2M design variant and the S2M design variant require different AOT files. The instructions in this section create both AOT files.

To add the compiled graphs to the development kit SD card:

Tip: If you completed the FPGA AI Suite Quick Start Tutorial in the *FPGA AI Suite Getting Started Guide*, you can skip steps 1-3.

1. Create the $COREDLA_WORK directory, if you have not already done so.
2. Prepare OpenVINO Model Zoo and Model Optimizer.
3. Prepare a model.

   Tip: If you completed the FPGA AI Suite Quick Start Tutorial in the *FPGA AI Suite Getting Started Guide*, you have already completed these first three steps.
4. Confirm that you have the following directory:

   ```
   $COREDLA_WORK/demo/models/public/resnet-50-tf/FP32/
   ```

   If you do not have this directory, confirm that you have completed the first three steps.
5. Compile the graphs.
6. Copy the compiled graphs to the SD card.

Related Information

FPGA AI Suite Quick Start Tutorial

3.6.1. Preparing OpenVINO Model Zoo

These instructions assume that you have a copy of OpenVINO Model Zoo 2023.3 in your `$COREDLA_WORK/demo/open_model_zoo/` directory.

To download a copy of Model Zoo, run the following commands:

```
cd $COREDLA_WORK/demo
git clone https://github.com/openvinotoolkit/open_model_zoo.git
cd open_model_zoo
git checkout 2023.3.0
```

Related Information

"OpenVINO Toolkit" in *FPGA AI Suite Getting Started Guide*

3.6.2. Preparing a Model

A model must be converted from a framework (such as TensorFlow, Caffe, or Pytorch) into a pair of `.bin` and `.xml` files before the FPGA AI Suite compiler (`dla_compiler` command) can ingest the model.

The following commands download the ResNet-50 TensorFlow model and run Model Optimizer:

```
source ~/build-openvino-dev/openvino_env/bin/activate
omz_downloader --name resnet-50-tf \
    --output_dir $COREDLA_WORK/demo/models/
omz_converter --name resnet-50-tf \
    --download_dir $COREDLA_WORK/demo/models/ \
    --output_dir $COREDLA_WORK/demo/models/
```

The `omz_downloader` command downloads the trained model to `$COREDLA_WORK/demo/models` folder. The `omz_converter` command runs model optimizer that converts the trained model into intermediate representation `.bin` and `.xml` files in the `$COREDLA_WORK/demo/models/public/resnet-50-tf/FP32/` directory.

The directory `$COREDLA_WORK/demo/open_model_zoo/models/public/resnet-50-tf/` contains two useful files that do not appear in the `$COREDLA_ROOT/demo/models/` directory tree:

- The `README.md` file describes background information about the model.
- The `model.yml` file shows the detailed command-line information given to Model Optimizer (`mo.py`) when it converts the model to a pair of `.bin` and `.xml` files

For a list OpenVINO Model Zoo models that the FPGA AI Suite supports, refer to the *FPGA AI Suite IP Reference Manual*.

3.6.3. Compiling the Graphs

The precompiled SD card image (`.wic`) provided with the FPGA AI Suite uses one of the following files as the IP architecture configuration file:

- Agilex 7 FPGA I-Series Transceiver-SoC Development Kit

  ```
  AGX7_Performance.arch
  ```

- Arria 10 SX SoC FPGA Development Kit

  ```
  A10_Performance.arch
  ```

To create the AOT file for the M2M variant (which uses the `dla_benchmark` utility), run the following command:

```
cd $COREDLA_WORK/demo/models/public/resnet-50-tf/FP32
dla_compiler \
--march $COREDLA_ROOT/example_architectures/<IP arch config file> \
--network-file ./resnet-50-tf.xml \
--foutput-format=open_vino_hetero \
--o $COREDLA_WORK/demo/RN50_Performance_b1.bin \
--batch-size=1 \
--fanalyze-performance
```

where *<IP arch config file>* is one of the IP architecture configuration files listed earlier.

To create the AOT file for the S2M variant (which uses the streaming inference app), run the following command:

```
cd $COREDLA_WORK/demo/models/public/resnet-50-tf/FP32
dla_compiler \
--march $COREDLA_ROOT/example_architectures/<IP arch config file> \
--network-file ./resnet-50-tf.xml \
--foutput-format=open_vino_hetero \
--o $COREDLA_WORK/demo/RN50_Performance_no_folding.bin \
--batch-size=1 \
--fanalyze-performance \
--ffolding-option=0
```

where *<IP arch config file>* is one of the IP architecture configuration files listed earlier.

After running either these commands, the compiled models and demonstration files are in the following locations:

Compiled Models	`$COREDLA_WORK/demo/RN50_Performance_b1.bin` `$COREDLA_WORK/demo/RN50_Performance_no_folding.bin`
Sample Images	`$COREDLA_WORK/demo/sample_images/`

Architecture File	`$COREDLA_ROOT/example_architectures/AGX7_Performance.arch`
	or
	`$COREDLA_ROOT/example_architectures/A10_Performance.arch`

3.6.4. Copying the Compiled Graphs to the SD card

To copy the required demonstration files to the `/home/root/resnet-50-tf` folder on the SD card:

1. In the serial console, create directories to receive the model data and sample images:

```
mkdir ~/resnet-50-tf
```

2. On the development host, use the secure copy (`scp`) command to copy the data to the board:

```
TARGET_IP=<Development Kit Hostname>.local
TARGET="root@$TARGET_IP:~/resnet-50-tf"
demodir=$COREDLA_WORK/demo
scp $demodir/*.bin $TARGET/.
scp -r $demodir/sample_images/ $TARGET/.
scp $COREDLA_ROOT/example_architectures/<architecture file> $TARGET/.
scp $COREDLA_ROOT/build_os.txt $TARGET/../app/
```

where *<architecture file>* is one of the following files, depending on your development kit:

- Agilex 7 FPGA I-Series Transceiver-SoC Development Kit

```
AGX7_Performance.arch
```

- Arria 10 SX SoC FPGA Development Kit

```
A10_Performance.arch
```

3. [Optional] In the serial console run the `sync` command to ensure that the data is flushed to disk.

3.7. Verifying FPGA Device Drivers

The device drivers should be loaded when the HPS boots. Verify that the device drivers are initialized by checking that `uio` files are listed in `/sys/class/uio` by running the following command:

```
ls /sys/class/uio
```

The command should show output similar to the following example:

```
uio0 uio1 uio2
```

If the drivers are not listed, refresh the modules by running the following command before checking again that the drivers are loaded:

```
uio-devices restart
```

3.8. Running the Demonstration Applications

Depending on the SoC design example mode that you want run, follow the instructions in one of the following sections:

- Running the M2M Mode Demonstration Application on page 28
- Running the S2M Mode Demonstration Application on page 29

Important: Running the demonstration applications requires the terminal session that you opened in Determining the SoC FPGA Development Kit IP Address on page 24.

3.8.1. Running the M2M Mode Demonstration Application

The M2M dataflow model uses the `dla_benchmark` demonstration application. The S2M bitstream supports both the M2M dataflow model and the S2M dataflow model.

You must know the host name of the SoC FPGA development kit. If you do not know the development kit host name, go back to Determining the SoC FPGA Development Kit IP Address on page 24 before continuing here.

To run inference on the SoC FPGA development kit:

1. Open an SSH connection to the SoC FPGA development kit:
   a. Start a new terminal session
   b. Run the following command:

      ```
      build-host:$ ssh <devkit_hostname>
      ```

      Where *<devkit_hostname>* is the host name you determined in Determining the SoC FPGA Development Kit IP Address on page 24.

      Continuing the example from Determining the SoC FPGA Development Kit IP Address on page 24, the following command would open an SSH connection:

      ```
      build-host:$ ssh arria10-62747948036a.local
      ```

2. In the SSH terminal, run the following commands:

   ```
   export compiled_model=~/resnet-50-tf/RN50_Performance_b1.bin
   export imgdir=~/resnet-50-tf/sample_images
   export archfile=~/resnet-50-tf/<architecture file>
   cd ~/app
   export COREDLA_ROOT=/home/root/app
   ./dla_benchmark \
      -b=1 \
      -cm $compiled_model \
      -d=HETERO:FPGA,CPU \
      -i $imgdir \
      -niter=8 \
      -plugin ./plugins.xml \
      -arch_file $archfile \
      -api=async \
      -groundtruth_loc $imgdir/TF_ground_truth.txt \
   ```

```
    -perf_est \
    -nireq=4 \
    -bgr
```

where <*architecture file*> is one of the following files, depending on your development kit:

- Agilex 7 FPGA I-Series Transceiver-SoC Development Kit

  ```
  AGX7_Performance.arch
  ```

- Arria 10 SX SoC FPGA Development Kit

  ```
  A10_Performance.arch
  ```

The `dla_benchmark` command generates output similar to the following example output for each step. This example output was generated using an Agilex 7 FPGA I-Series Transceiver-SoC Development Kit.

```
[Step 11/12] Dumping statistics report
count:             8 iterations
system duration:   288.8584 ms
IP duration:       121.6040 ms
latency:           136.0344 ms
system throughput: 27.6952 FPS
number of hardware instances: 1
number of network instances: 1
IP throughput per instance: 65.7873 FPS
IP throughput per fmax per instance: 0.3289 FPS/MHz
IP clock frequency: 200.0000 MHz
estimated IP throughput per instance: 149.5047 FPS (500 MHz assumed)
estimated IP throughput per fmax per instance: 0.2990 FPS/MHz
[Step 12/12] Dumping the output values
[ INFO ] Dumping result of Graph_0 to result.txt, result.bin, result_meta.json,
and result_tensor_boundaries.txt
[ INFO ] Comparing ground truth file /home/root/quartus_uplift/resnet-50-tf/
sample_images/TF_ground_truth.txt with network Graph_0
top1 accuracy: 100 %
top5 accuracy: 100 %
[ INFO ] Get top results for "Graph_0" graph passed
```

3.8.2. Running the S2M Mode Demonstration Application

To run the S2M (streaming) mode demonstration application, you need two terminal connections to the host.

You must know the host name of the SoC FPGA development kit. If you do not know the development kit host name, go back to Determining the SoC FPGA Development Kit IP Address on page 24 before continuing here.

To run the streaming demonstration application:

1. Open an SSH connection to the SoC FPGA development kit:
   a. Start a new terminal session
   b. Run the following command:

      ```
      build-host:$ ssh <devkit_hostname>
      ```

      Where <*devkit_hostname*> is the host name you determined in Determining the SoC FPGA Development Kit IP Address on page 24.

Continuing the example from Determining the SoC FPGA Development Kit IP Address on page 24, the following command would open an SSH connection:

```
build-host:$ ssh arria10-62747948036a.local
```

2. Repeat step 1 to open a second SSH connection to the SoC FPGA development kit.
3. In a terminal session, run the following commands:

```
cd /home/root/app
./run_inference_stream.sh
```

4. In the other terminal session, run the following commands:

```
cd /home/root/app
./run_image_stream.sh
```

The first terminal session (where you ran the `run_inference_stream.sh` command) then shows output similar to the following example:

```
root@arria10-ea80b8d770e7:~/app# ./run_inference_stream.sh
Runtime arch check is enabled. Check started...
Runtime arch check passed.
Runtime build version check is enabled. Check started...
Runtime build version check passed.
Ready to start image input stream.
1 - coffee mug, score = 93.9453
2 - acoustic guitar, score = 38.6963
3 - desktop computer, score = 43.9209
4 - guacamole, score = 99.9512
5 - red wine, score = 55.1758
6 - stopwatch, score = 38.8428
7 - jigsaw puzzle, score = 100
```

For more details about the streaming apps and their command line options, refer to Running the Streaming Demonstration on page 67.

3.8.3. Troubleshooting the Demonstration Applications

If you receive an error similar to the following error while running either the S2M or M2M applications, check that all the board DIMMs are securely installed:

```
altera-msgdma ff200000.msgdma: dma_sync_wait: timeout! [ 251.812846] DMA Failed
```

4. FPGA AI Suite SoC Design Example Run Process

This section describes the steps to run the demonstration application and perform accelerated inference using the SoC design example.

4.1. Exporting Trained Graphs from Source Frameworks

Before running any demonstration application, you must convert the trained model to the OpenVINO intermediate representation (IR) format (`.xml`/`.bin`) with the OpenVINO Model Optimizer.

For details on creating the `.xml`/`.bin` files, refer to Preparing a Model on page 25, which describes how to create `.xml`/`.bin`) files for ResNet50.

The rest of this guide assumes that the same file locations and file names are used as in Preparing OpenVINO Model Zoo on page 25.

The `stream_image_app` used for the S2M variant of the SoC design example assumes that images are 224x224. For details, refer to The image_streaming_app Application on page 68.

4.2. Compiling Exported Graphs Through the FPGA AI Suite

The network as described in the `.xml` and `.bin` files (created by the Model Optimizer) is compiled for a specific FPGA AI Suite architecture file by using the FPGA AI Suite compiler.

The FPGA AI Suite compiler compiles the network and exports it to a `.bin` file with the format required by the OpenVINO Inference Engine. For instructions on how to compile the `.xml` and `.bin` files into AOT file suitable for use with the FPGA AI Suite IP, refer to Compiling the Graphs on page 26

This `.bin` file created by the compiler contains the compiled network parameters for all the target devices (FPGA, CPU, or both) along with the weights and biases. The inference application imports this file at runtime.

The FPGA AI Suite compiler can also compile the graph and provide estimated area or performance metrics for a given architecture file or produce an optimized architecture file.

For the demonstration SD card, the FPGA bitstream has been built using one of the following IP architecture configuration files, so the architecture file for your development kit for compiling the OpenVINO™ Model:

- Agilex 7 FPGA I-Series Transceiver-SoC Development Kit

  ```
  AGX7_Performance.arch
  ```

- Arria 10 SX SoC FPGA Development Kit

  ```
  A10_Performance.arch
  ```

For more details about the FPGA AI Suite compiler, refer to the *FPGA AI Suite Compiler Reference Manual*.

5. FPGA AI Suite SoC Design Example Build Process

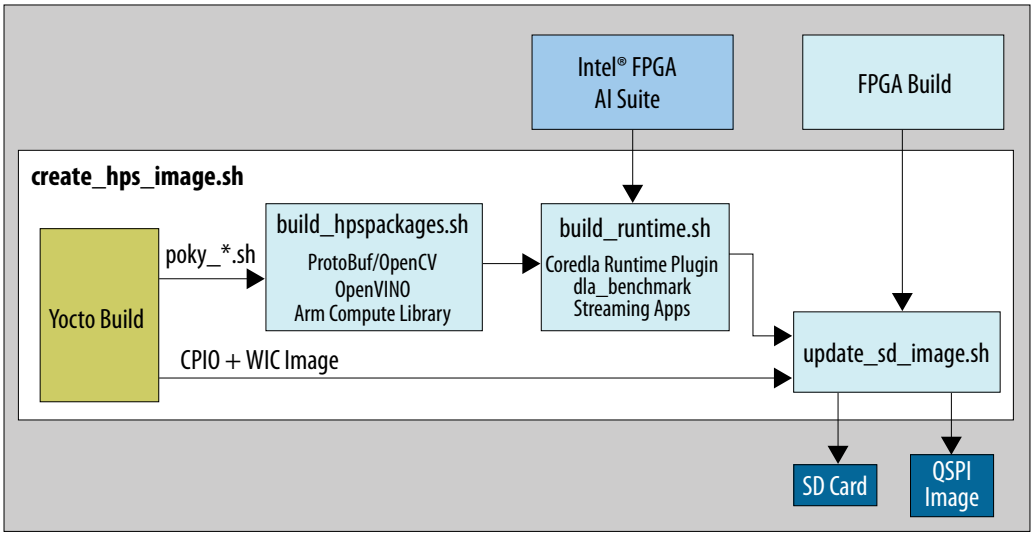
The SoC design example is built around a complete software and hardware solution.

The main building stages are:

- Build a Linux Distribution for a supported SoC FPGA development kit using a cross-compilation flow on a Linux system.
- Build the SoC design example Quartus Prime project
- Combine the FPGA and SoC Linux application and kernel onto an SD card

The following diagram illustrates the overall build process.

Figure 1. FPGA AI Suite SoC Design Example Build Process



5.1. Building the Quartus Prime Project

This design example includes prepackaged bitstreams but you can also use the build script provided to build bitstreams with custom architectures or recreate the original bitstreams, subject to IP license limitations.

The Quartus Prime project consists of the FPGA AI Suite IP as well as IP to interface with the HPS and, in the case of the S2M variant, additional flow control IP.

FPGA AI Suite SoC Design Example Quick Start Tutorial on page 9 shows how to use a specific architecture configuration file for the FPGA AI Suite IP, but you can use any other device-appropriate configuration file instead.

5.1.1. Quartus Prime Build Flow

All FPGA AI Suite design examples are launched at the command line by running the `dla_build_example_design.py` script.

After the build script is invoked, it generates an FPGA AI Suite IP from the provided architecture file, creates an Quartus Prime build directory, builds the Quartus Prime project, and produces a bitstream.

The script has several command-line options to select the SoC design example variants. For details about the build script command options, refer to Build Script Options on page 35.

Before launching the script, an architecture file is required. The FPGA AI Suite example architectures are located in directory `$COREDLA_ROOT/example_architectures/`.

Typical launch usage is as follows:

```
dla_build_example_design.py \
  -ed <variant> \
  -a <arch-file> \
  -n 1 \
  --build-dir <build directory> \
  --build \
  --output-dir <output directory>
```

The command options are defined as follows:

- The `-ed` option selects the SoC design example variant to be built. This option is case sensitive. The FPGA AI Suite SoC design examples comes as the following variants:

Table 2. SoC Design Example Variant

Variant setting	Description
`4_AGX7_M2M`	Builds a memory-to-memory (M2M) design for the Agilex 7 FPGA I-Series Transceiver-SoC Development Kit
`4_AGX7_S2M`	Builds a streaming-to-memory (S2M) design for the Agilex 7 FPGA I-Series Transceiver-SoC Development Kit
`4_A10_M2M`	Builds a memory-to-memory (M2M) design for the Arria 10 SX SoC FPGA Development Kit (DK-SOC-10AS066S)
`4_A10_S2M`	Builds a streaming-to-memory (S2M) design for the Arria 10 SX SoC FPGA Development Kit (DK-SOC-10AS066S)

- The `-a` option selects the architecture file.
- The `-n 1` option is the only legal value for this option when you build the SoC design example.
- The `--build-dir` option specifies the Quartus Prime build directory path.
- The `--build` option directs the script to call Quartus Prime and create the bitstreams. The `create_hps_image.sh` script uses these bitstreams when creating the SD card image.
- The `--output-dir` option specified the destination directory folder of the build output.

For a complete list of the build script options, refer to "Build Script Options" in *FPGA AI Suite PCIe-based Design Example User Guide*.

An example of building the Arria 10 S2M variant with the `A10_Performance` architecture in the folder `build_a10_perf` is as follows:

```
dla_build_example_design.py \
 -ed 4_A10_S2M \
 -n 1 \
 -a $COREDLA_ROOT/example_architectures/A10_Performance.arch \
 --build \
 --build-dir $COREDLA_WORK/a10_perf_bitstream \
 --output-dir $COREDLA_WORK/a10_perf_bitstream
```

After the design is built, the output products (`.sof` or `.rbf` files) must be combined with the SoC Linux system in order to be used. This is done in one of the steps in the `create_hps_image.sh` script.

Unlike non-SoC FPGA flows, the `.sof` file is not downloaded to the board via JTAG.

Do not attempt to download the `.sof` file over JTAG because downloading the file over JTAG does not result in a working system. Also, if you attempt to reprogram a running Linux system with a new `.sof` file, the Linux system crashes and the reprogramming results in an unpredictable outcome.

The FPGA device is programmed by booting the Linux system on the SoC via the SD card. For details about combining the build `.sof` file with the SD card image to create a functional solution, refer to Building the Bootable SD Card Image (.wic) on page 37.

5.1.1.1. Build Synchronization of FPGA with Software

For a system to function correctly, the release version of each FPGA AI Suite component, including the compiler, the runtime, and the FPGA AI Suite IP, must match.

In addition, the AOT file created by the FPGA AI Suite `dla_compiler` command must target the same architecture (`.arch`) file as the FPGA AI Suite IP.

When Quartus Prime compiles the FPGA AI Suite IP, it generates a build-hash that is embedded into the IP. The runtime software checks this build-hash during runtime and if the hashes do not match then the application aborts and displays a mismatch error.

The FPGA AI Suite SoC design example is always built with only one instance of the FPGA AI Suite IP.

5.1.2. Build Script Options

The options available in the `dla_build_example_design.py` script are described in "Build Script Options" in *FPGA AI Suite PCIe-based Design Example User Guide*.

The options that are specific to the *FPGA AI Suite* SoC design example are as follows:

Option	Description
`-ed`, `--example-design-id`	Specify the SoC design example variant as follows:

continued...

Option	Description
	• Agilex 7 SoC M2M variant: `4_AGX7_M2M` • Agilex 7 SoC S2M variant: `4_AGX7_S2M` • Arria 10 SoC M2M variant: `4_A10_M2M` • Arria 10 SoC S2M variant, `4_A10_S2M`
`-n, --num-instances`	Number of IP instances to build (default: 1). For SoC designs, this number must be 1.

5.1.3. Build Directory

The `dla_build_example_design.py` command creates an Quartus Prime build in the `hw` folder in the directory that you specify with the `--build-dir` command option. The project is named `top.qpf`. You can open this project in Quartus Prime software to review the build logs.

Within the build directory is a collection of command-line scripts that cover different parts of the design example build process. Use these scripts to rebuild parts of the design example if you alter the design. Otherwise, you typically do not need to run these scripts manually as the build process runs them for you.

The scripts provided are as follows:

- `create_project.bash`: This script cleans the build folder and resets the Quartus Prime project back to its default state, ready to be recompiled.
- `generate_sof.bash`: This script launches an Quartus Primer compilation from the command line
- `generate_rbf.bash`: This creates a `.rbf` file that is needed for the Arria 10 FPGA.
- `build_stream_controller.sh`: This script creates the Nios® V `.hex` file. This file holds the compiled Nios software that is embedded into the Nios subsystem.

5.1.3.1. The `create_project.bash` Script

The `create_project.bash` script has two command line options:

```
create_project.bash <arch> <ip_path>
```

The arguments must be placed in order.

- The *<arch>* option provides the architecture to be selected when building the SoC Example Design. Use the architecture file that was used as a command option in `dla_build_example_design.py` command.

  The *<arch>* value to specify is extracted from the architecture file name:

Architecture File Name	Command Line Option
`AGX7_Performance.arch`	`AGX7_Performance_AGX7`
`AGX7_FP16_SoftMax.arch`	`AGX_FP16_SoftMax_AGX7`
`A10_Performance.arch`	`A10_Performance_A10`
`A10_FP16_SoftMax.arch`	`A10_FP16_SoftMax_A10`

- The *<ip_path>* option provides a full path to the FPGA AI Suite IP

This script deletes the old output products, cleans the Platform Designer system, and resets the project.

5.1.3.2. The `generate_sof.bash` Script

The `generate_sof.bash` script launches a Quartus Prime shell and invokes the build process. Upon completion, a `.sof` file is ready for use.

5.1.3.3. The `generate_rbf.bash` Script

The `generate_rbf.bash` script converts the `.sof` file into a `.rbf` (raw binary file) file that can be used by the Linux system.

5.1.3.4. The `build_stream_controller.sh` Script

The `build_stream_controller.sh` script builds the Nios V application, and then generates a `.hex` file. The `.hex` file is embedded into the Nios V RAM by the Quartus Prime project. This file must be called `stream_controller.hex` and it must reside alongside the Quartus Prime project files (`top.qpf`).

This script has three command line options that must be entered in order:

```
build_stream_controller <quartus-project-file> <qsys system> <output-file>
```

This script is called as part of the `create_project.bash`, but you can call it manually if you modify the Nios V source code and need a new `.hex` file. The build script is typically called with the same options.

```
build_stream_controller.sh top.qpf qsys/dla.qsys stream_controller.hex
```

5.2. Building the Bootable SD Card Image (`.wic`)

To create a bootable SD Card image (`.wic` file), the following components must be built:

- Yocto Image
- Yocto SDK Toolchain
- Arm Cross-Compiled OpenVINO

- Arm Cross-Compiled FPGA AI Suite Runtime Plugin
- Arm Cross-Compiled Demonstration Applications
- FPGA `.sof/.rbf` files

The `create_hps_image.sh` script performs the complete build process and combines all the necessary components into an SD card image that can be written to an SD card. For steps required to write an SD card image to an SD card, refer to Writing the SD Card Image (.wic) to an SD Card on page 15.

The SD card image is assembled using Yocto (https://yoctoproject.org).

You must have a build system that meets the minimum build requirements. For details, refer to https://docs.yoctoproject.org/5.0.5/ref-manual/system-requirements.html#supported-linux-distributions.

The commands to install the required packages are shown in Installing HPS Disk Image Build Prerequisites on page 12.

To perform the build, run the following commands:

```
cd $COREDLA_WORK/runtime
./create_hps_image.sh \
   -f <bitstream_directory>/hw/output_files \
   -o <output_directory> \
   -u \
   -m <FPGA_target>
```

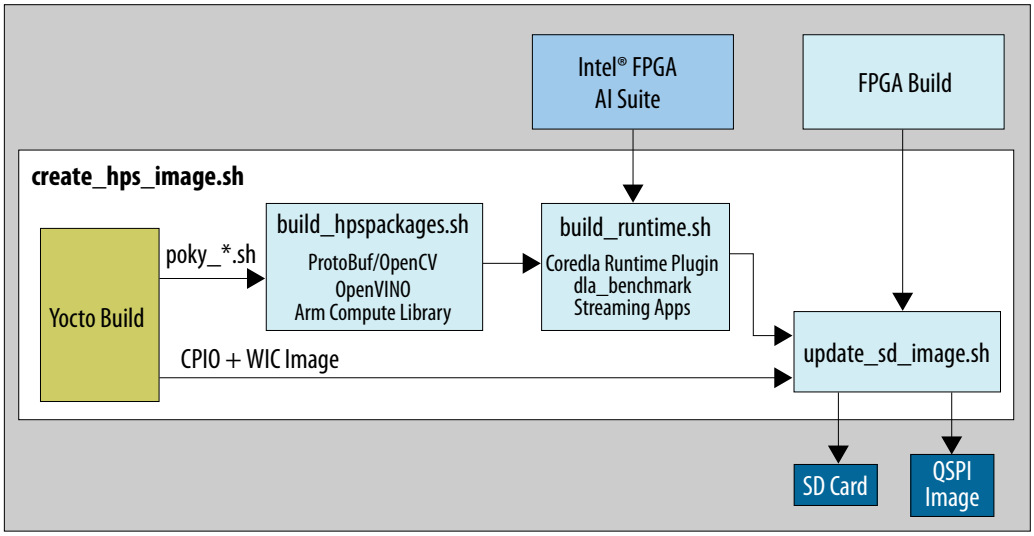
where *<FPGA_target>* is `arria10` or `agilex7_dk_si_agi027fa`

The `create_hps_image.sh` script performs the following steps:

1. Build the Yocto boot SD card image and Yocto SDK toolchain on page 39
2. Build the HPS Packages on page 40
3. Build the runtime on page 40
4. Update the SD card image on page 40

The following diagram illustrates the overall build process performed by the `create_hps_image.sh` script:

Figure 2. FPGA AI Suite SoC Design Example Build Process



Build the Yocto boot SD card image and Yocto SDK toolchain

The FPGA AI Suite Soc design example uses the Yocto Project Poky Distribution.

The Yocto images are based on Golden System Reference Designs, which you can find at the following URL: https://www.rocketboards.org/foswiki/Documentation/GSRD.

To customize the Yocto Poky distribution, modify the recipes found in layer `$COREDLA_ROOT/hps/ed4/yocto/meta-intel-coredla`.

More details can be found in Yocto Build and Runtime Linux Environment on page 57.

The defined Yocto Image recipe is `coredla-image` and can be found in `$COREDLA_ROOT/hps/ed4/yocto/meta-intel-coredla/recipes-image/coredla-image.bb`.

A Yocto SDK is also built as part of the build and this SDK is used in subsequent build steps to cross-compile the software for the Arm HPS subsystem:

- **Agilex 7**:

  ```
  $COREDLA_WORK/runtime/build_Yocto/build/tmp/deploy/sdk/poky-glibc-x86_64-coredla-image-armv8a-agilex7_dk_si_agi027fa-toolchain-4.2.3.sh
  ```

- **Arria 10**:

  ```
  $COREDLA_WORK/runtime/build_Yocto/build/tmp/deploy/sdk/poky-glibc-x86_64-lbs-image-poky-cortexa9t2hf-neon-arria10-toolchain-4.1.2.sh
  ```

The SD card image (WIC file) is in the following location:

- **Agilex 7**:

  `$COREDLA_WORK/runtime/build_Yocto/build/tmp/deploy/images/agilex7_dk_si_agi027fa/*`

- **Arria 10**:

  `$COREDLA_WORK/runtime/build_Yocto/build/tmp/deploy/images/arria10/*`

By default, the `create_hps_image.sh` script builds Yocto from scratch. However, if a prebuilt Yocto build folder is available, you can specify the prebuilt Yocto folder via the `-y` option as follows:

```
./create_hps_image.sh \
   -y <prebuilt_Yocto_directory>
   -f <bitstream_directory>/hw/output_files \
   -o <output_directory>
   -u
   -m <FPGA_target>
```

where *<FPGA_target>* is `arria10` or `agilex7_dk_si_agi027fa`

This `-y` option loads the Yocto SDK from *<PREBUILT_YOCTO_DIR>*`/build/tmp/deploy/sdk/` and the `.wic` image from *<PREBUILT_YOCTO_DIR>*`/build/tmp/deploy/images/arria10/` without rerunning a Yocto build.

Build the HPS Packages

The HPS packages are built by the `build_hpspackages.sh` script.

This script cross-compiles OpenCV, OpenVINO, and the Arm-based OpenVINO runtime plugin.

Build the runtime

The runtimes are built by the `build_runtime.sh` script.

This script cross-compiles the OpenVINO FPGA AI Suite runtime plugin and demonstration applications for the SoC devices.

Update the SD card image

The SD card image is updated by the `update_sd_image.sh` script.

This script takes the output products from the previous build steps and builds a bootable SD Card image.

The software binaries are installed to the Ext4 partition under the `/home/root/app` directory. The RTL `fit_spl_fpga.itb` file is copied to the Fat32 partition.

The SD card image is updated only if you specify the `-u` option of the `create_hps_image.sh` command along with the location of the FPGA bitstream directory through the `-f` option.

You can skip updating the SD card while building the rest of the SoC Example Design by omitting the `-f` and `-u` options:

```
./create_hps_image.sh \
   -o <output_directory> \
   -m <FPGA_target>
```

where *<FPGA_target>* is `arria10` or `agilex7_dk_si_agi027fa`

When you skip updating the SD card image, you can build bitstreams and an HPS image (Yocto, HPS packages, FPGA AI Suite runtime) concurrently. You can update the SD card image (`.wic` file) image after all the files are ready:

```
./create_hps_image.sh \
   -y ./build_Yocto
   -f <bitstream_directory>/hw/output_files \
   -o <output_directory> \
   -u \
   -m <FPGA_target>
```

where *<FPGA_target>* is `arria10` or `agilex7_dk_si_agi027fa`

6. FPGA AI Suite SoC Design Example Quartus Prime System Architecture

The FPGA AI Suite SoC design examples provide two variants for demonstrating the FPGA AI Suite operation.

All designs are Platform Designer based systems.

There is a single top-level Verilog RTL file for instantiating the Platform Designer system.

These two variants demonstrate FPGA AI Suite operations in the two most common usage scenarios. These scenarios are as follows:

- **Memory to Memory (M2M)**: In this variant, the following steps occur:
  1. The Arm processor host presents input data buffers to the FPGA AI Suite that are stored in a system memory.
  2. The FPGA AI Suite IP performs an inference on these buffers.
  3. The host system collects the inference results.

  This variant demonstrates the simplest use-case of the FPGA AI Suite.
- **Streaming to Memory (S2M)**: This variant offers a superset of the M2M functionality. The S2M variant demonstrates sending streaming input source data into the FPGA AI Suite IP and then collecting the results. An Avalon® streaming input captures live input data, stores the data into system memory, and then automatically triggers FPGA AI Suite IP inference operations.

  You can use this variant as a starting point for larger designs that stream input data to the FPGA AI Suite IP with minimal host intervention.

6.1. FPGA AI Suite SoC Design Example Inference Sequence Overview

The FPGA AI Suite IP works with system memory. To communicate with the system memory, the FPGA AI Suite has its own multichannel DMA engine.

This DMA engine pulls input commands and data from the system memory. It then writes the output data back to this memory for a Host CPU to collect.

When running inferences, the FPGA AI Suite continually reads and writes intermediate buffers to and from the system memory. The allocation of all the buffer addresses is done by the FPGA AI Suite runtime software library.

Running an inference requires minimal interaction between the host CPU and the IP Registers.

The system memory must be primed with all necessary buffers before starting a machine learning inference operation. These buffers are setup by the FPGA AI Suite runtime library and application that runs on the host CPU.

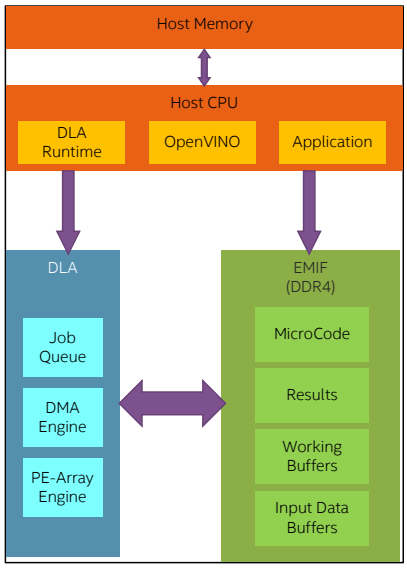
After the setup is complete, the host CPU pushes a job into the IP registers.

The FPGA AI Suite IP now performs a single inference. The job-queue registers in the IP are FIFO based, and the host application can store multiple jobs in the system memory and then prime multiple jobs inside the IP. Each job stores results in system memory and results in a CPU interrupt request.

For each inference operation in the M2M model, the host CPU (HPS) must perform an extensive data transfer from host (HPS) DDR memory to the external DDR memory that is allocated to the FPGA AI Suite IP. As this task has not been FPGA-accelerated in the design, the host operating system and FPGA AI Suite runtime library must manually transfer the data. This step consumes significant CPU resources. The M2M design uses a DMA engine to help with the data transfer from HPS DDR to the allocated DDR memory.

The FPGA AI Suite inference application and library software are responsible for keeping the FPGA AI Suite IP primed with new input data and responsible for consuming the results.

Figure 3. FPGA AI Suite SoC Design Example Inference Sequence Overview



For a detailed overview of the FPGA AI Suite IP inference sequence, refer to the *FPGA AI Suite IP Reference Manual*.

6.2. Memory-to-Memory (M2M) Variant Design

The memory-to-memory (M2M) variant of the SoC design example illustrates a technique for embedded (SoC) FPGA AI Suite operations where the input data sets are primarily drawn from a memory or file sources. In this scenario, the data is typically not real time and is processed as fast as possible.

This design combines the HPS SoC FPGA device (Arm Cortex*-A9 on Arria 10 or Arm Cortex-A53 on Agilex 7) with an additional DMA engine to allow for efficient transfer of data to and from the CPU and system memory.

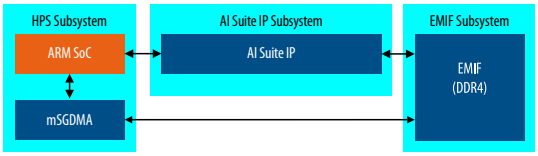
In the M2M design, the source data originally resides within the host CPU domain on an SD card. The application uses the DMA controller to move the host-side data to the device side domain. This movement mimics the process that an application would typically do.

The test program then initiates FPGA AI Suite IP inference operations and wait for the IP to complete its process. Command-line options to the user application define how many inferences are executed.

After the inference operation is completed, the application uses the DMA to transfer the results back from external memory to the host domain. The results are then displayed on the Linux console.

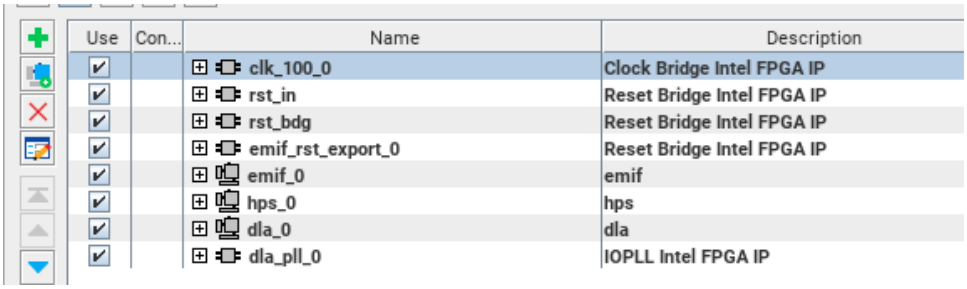
The Intel modular scatter-gather direct memory access (mSGDMA) controller IP provides this DMA facility.

Figure 4. Block Diagram of M2M Variant



The M2M variant appears in Platform Designer as follows:

Figure 5. M2M Variant in Platform Designer



6.2.1. The mSGDMA Intel FPGA IP

The modular scatter-gather direct memory access (mSGDMA) Intel FPGA IP used in this design example serves as an example of how you can integrate a DMA into your own system. You can replace this DMA engine by another 3rd party controller.

The FPGA AI Suite runtime software must be modified if you want to use another DMA engine.

6.2.2. RAM considerations

An FPGA-based external memory interface is used to store all machine learning input, output, and intermediate data.

The FPGA AI Suite IP uses the DDR memory extensively in its operations.

Typically, you dedicate a memory to the FPGA AI Suite IP and avoid sharing it with the host CPU DDR memory. Although a design can use the host memory, other services that use the DDR memory impact the FPGA AI Suite IP performance and increase non-determinism in inference durations. Consider this impact when you choose to use a shared DDR resource.

The FPGA AI Suite IP requires an extensive depth of memory that prohibits the use of onboard RAM such as M20Ks. Consider the RAM/DDR memory footprint when you design with the FPGA AI Suite IP.

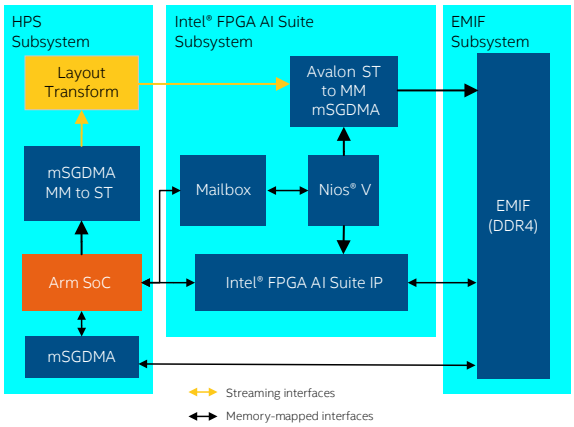
6.3. Streaming-to-Memory (S2M) Variant Design

The streaming-to-memory (S2M) variant of the SoC design example builds on top of the M2M design to demonstrate a method of using the FPGA AI Suite IP with continuously streaming input data.

The application example is a typical video stream being processed with ResNet50 to detect physical objects in the images, such as a person, cat, or dog.

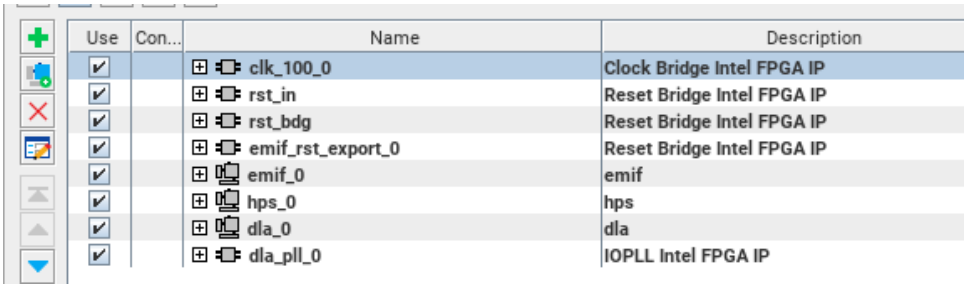
In the example, test images are stored on the SD card file system. These images are loaded into host memory and a DMA (memory-to-streaming) IP is used to create a simulated video stream.

Figure 6. Block Diagram of S2M Variant



The s2M variant appears in Platform Designer as follows:

Figure 7. S2M Variant in Platform Designer



6.3.1. Streaming Enablement for FPGA AI Suite

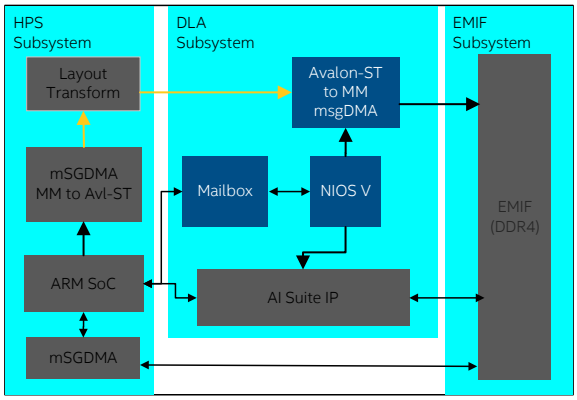
In an M2M system, input buffers are provided by the host CPU. However, in a streaming system (S2M), input buffers are created by an external hardware stream. For the FPGA AI Suite IP to process this external stream, several operations must happen in a coordinated way:

- The raw stream data must pass through a layout-transform IP core to reformat the raw data into an FPGA AI Suite compliant data format
- The formatted data must be written into system memory at specific locations, known only to the host application and the FPGA AI Suite software library at run time.
- The FPGA AI Suite IP job queue must be primed at the correct time, in synchronization with the input stream buffers, such that the FPGA AI Suite IP starts an inference immediately upon a new input buffer becoming ready.

Within Platform Designer, a Nios V based subsystem is added alongside the FPGA AI Suite IP to provide the streaming capabilities. This subsystem highlighted in blue in the block diagram that follows.

In the diagram, the yellow interconnect lines indicate Avalon streaming interfaces, and the black interconnect lines indicate memory-mapped interfaces.

Figure 8. Nios V Streaming Subsystem



6.3.2. Nios V Subsystem

Three IP modules make up the Nios V subsystem:

- **mSGDMA** (Avalon streaming to memory-mapped mode). This module is used to take the formatted input data stream and place it into system memory to create the FPGA AI Suite IP input buffers.
- **Mailbox** (On-Chip Memory II Intel FPGA IP). This module is used to provide a communication API between the host-application and the Nios Subsystem. FPGA AI Suite IP command and status message are conveyed through this interface.
- **Nios V processor**. This module manages the FPGA AI Suite IP job-queue, mailbox and mSGDMA buffer allocation. Using the Nios V processor offloads the latency-sensitive ingest and buffer management from the HPS.

All C source-code to the Nios V application is provided. You can modify the Nios software to enable third-party DMA controllers, if required.

6.3.3. Streaming System Operation

Two operations must occur in parallel for streaming systems to work:

- Buffers must be managed appropriately. That is, buffers of streaming data must be written into system memory at a specific location ready for the FPGA AI Suite IP to process.
- Inference jobs must be managed appropriately. That is, inference jobs must be primed at the correct time to process the new buffer.

6.3.3.1. Streaming System Buffer Management

Before machine learning inference operations can occur, the system requires some initial configuration.

As in the M2M variant, the S2M application allocates sections of system memory to handle the various FPGA AI Suite IP buffers at startup. These include the graph buffer, which contains the weights, biases and configuration, and the input and output buffers for individual inference requests.

Instead of fully managing these buffers, the input-data buffer management is offloaded to the Nios processor. The Nios processor owns the Avalon streaming to memory-mapped mSGDMA, and the processor programs this DMA to push the formatted data into system memory.

As the buffers are allocated at startup, the input buffer locations are written into the mailbox. The Nios V processor then holds onto these buffers until a new set is received. All stream data is now constantly pushed into these buffers in a circular ring-buffer concept.

6.3.3.2. Streaming System Inference Job Management

In a M2M system, the host CPU handles pushing jobs into the FPGA AI Suite IP job queue by writing to the IP registers. In the streaming configuration, this task is offloaded to the Nios system and must be done in a coordinated way with the input buffer writing.

In streaming mode, the job-queue management is pushed to the mailbox instead of being managed by the host application. The job queue entry is then received by the Nios processor. After an input buffer is written, the mSGDMA interrupts the Nios processor, and the Nios processor now pushes one job into the FPGA AI Suite IP.

For every buffer stored by the mSGDMA, the Nios processor attempts to start another job.

For more details about the Nios V Stream Controller and the mailbox communication protocol, refer to Streaming-to-Memory (S2M) Streaming Demonstration on page 61.

6.3.4. Resolving Input Rate Mismatches Between the FPGA AI Suite IP and the Streaming Input

When designing a system, the stream buffer rate should be matched to the FPGA AI Suite IP inferencing rate, so that the input data does not arrive faster than the IP can process it.

The SoC design example has safeguards in the Nios subsystem for when the input data rate exceeds the FPGA AI Suite processing rate.

To prevent input buffer overflow (potentially writing to memory still being processed by the FPGA AI Suite IP), the Nios subsystem has a buffer dropping technique built into it. If the subsystem detects that the FPGA AI Suite IP is falling behind, it starts dropping input buffers to allow the IP to catch up.

Using mailbox commands, the host application can check the queue depth level of the Nios subsystem and see if the subsystem needs to drop input data.

Depending on the buffer processing requirements of a design, dropping input data might not be considered a failure. It is up to you to ensure that the IP inference rate meets the needs of the input data.

If buffer dropping is not desired, you can try to alleviate buffer dropping and increase FPGA AI Suite IP performance with the following options:

- Configure a higher performance `.arch` file (IP configuration), which requires more FPGA resource. The `.arch` can be customized for the target machine learning graphs.
- Increase the system clock-speed.
- Reduce the size of the machine learning network, if possible.
- Implement multiple instances of the FPGA AI Suite IP and multiplex input data between them.

6.3.5. The Layout Transform IP as an Application-Specific Block

The layout transformation IP in the S2M design is provided as RTL source as an example layout transformation within a video inferencing application.

The flexibility of the FPGA AI Suite and the scope of projects it can support means that a layout transformation IP cannot serve all inference applications.

Each target application typically requires its own layout transformation module to be designed. System architectures need to budget for this design effort within their project.

Input data to the FPGA AI Suite IP must be formatted in memory so that the data matches the structure of the IP PE array and uses FP16 values.

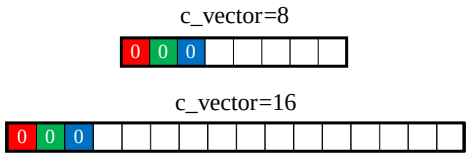
The structure of the PE array is defined by the architecture file and the `c_vector` parameter setting describes the number of layers required for the input buffer. Typical `c_vector` values are 8, 16, and 32.

When considering streaming data, the `c_vector` can be understood in comparison to the number of input channels of data that is present. For example, video has red, green, and blue channels that make up each pixel color. The following diagram shows how the video channels map to the input data stream required by the FPGA AI Suite IP.

Figure 9. Input Data Steam Mapping

The S2M design demonstrates an example of video streaming. Pixel data is sent through the layout-transform as RGB pixels, where each color is considered an input channel of data.

As the input data comprise only three channels of input data, the input data must be padded with zeros for any unused channels. The following diagram shows an example of two architectures, one with `c_vector` value of 8 and another with `c_vector` value of 16.



In the first example where `c_vector` is set to 8, the first pixel of RGB is placed on the input stream filling the first 3 channels, but there are 5 more channels remaining that must be initialized. These are filled with zero (represented by the white squares). This padded stream is then fed into the Nios subsystem.

This example layout transform does not support input folding. Input folding is an input preprocessing step that reduces the amount of zero padding in the `c_vector`. This folding then enables more efficient use of the dot product engine in the FPGA AI Suite IP. The efficiency gains can be significant depending on the graph and C_VEC. For more details, refer to "Input Folding" in *FPGA AI Suite IP Reference Manual*.

Related Information

"Parameter: `c_vector`" in *FPGA AI Suite IP Reference Manual*

6.3.5.1. Layout Transform Considerations

Pixels are typically 8-bit integer values, and the FPGA AI Suite requires FP16 values. As well as the `c_vector` padding, the layout transformation module converts the integer values to floating-point values.

The S2M example is a video-oriented demonstration. For networks such as ResNet50, the input pixel data must further be manipulated with a "mean" and "variance" value. The layout transformation module performs basic operation of Y=A*B+C operation on each pixel to meet the needs of a ResNet50 graph trained for ImageNet.

6.3.5.2. Layout Transform IP Register Map

The layout transform IP has the following register address space:

Table 3. Register Address Space of Layout Transformation IP

Register	Address Range	Description
Control	0x00	Main Control Register
C-Vector	0x04	Global C-Vector Control
Reserved	0x08 .. 0x03f	
Variance	0x40 .. 0x7f	Variance values per plane
Mean	0x80 .. 0xbf	Mean Values per plane
Reserved	0xc0 .. 0xff	

Control Register (0x00)

This is the global control register. When altering other CSR registers, software should issue a reset to the LT module via this register to commission the new settings. The stream then generates outputs based on the new settings and flush out all stale data.

Attempting to alter the configuration registers of the LT during active streaming generates undefined output data.

Table 4. Control Register Layout

Bit Location	Register Description	Attributes
0	**Reset** '1' = **In reset**: All streaming input data is discarded and no output data generated. '0' = **Running**: Streaming input data produces LT output data.	RW
31:1	Reserved	RO

C-Vector Register (0x04)

This register should be configured to match the C-Vector value of the architecture built into the FPGA AI Suite. This value need only be written once at startup as the architecture of the FPGA AI Suite is fixed at build time.

Table 5. C-Vector Register Layout

Bit Location	Register Description	Attributes
5:0	**C-vector** Value must match architecture of the DLA as defined in the .arch file	RW
31:6	Reserved	RO

Variance Registers (0x40 .. 0x7F)

Each plane has a unique variance value. Software must configure a value for each plane. The values are stored in FP32 format.

There are 16 registers in this section, where each register relates to a given plane. This register is write-only and returns 0xFFFFFFFF when reading.

Table 6. Variance Registers Layout

Bit Location	Register Description	Attributes
31:0	**FP32 formatted Variance value per plane**	WO

Mean Registers (0x80 .. 0xBF)

Each plane has a unique a mean value. Software must configure a value for each plane. The values are stored in FP32 format.

There are 16 registers in this section, where each register relates to a given plane. This register is write-only and returns 0xFFFFFFFF when reading.

Table 7. Mean Registers Layout

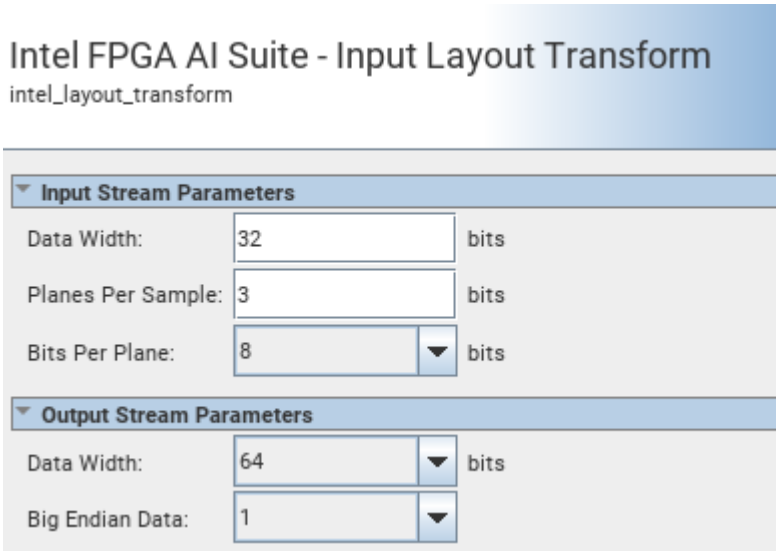
Bit Location	Register Description	Attributes
31:0	**FP32 formatted Mean value per plane**	WO

6.3.5.3. Layout Transform Configuration Options

The example layout transform has a range of parameters to adjust to the data width based on the number of input planes being processed.

A maximum of 16 CSR mean and variance values are supported. The **Planes per sample** field sets this upper threshold.

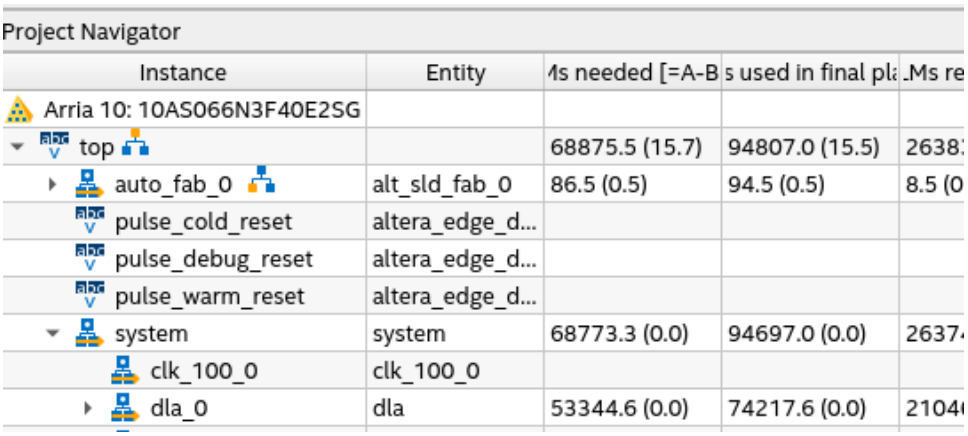
All output data is in FP16 format which is the expected input format for the FPGA AI Suite.



6.4. Top Level

After the Quartus Prime project has finished compiling, the design should look similar to the following image in the Quartus Prime Project Navigator:

Figure 10. SoC Design Example Hierarchy



The top-level Verilog file and HPS configuration is derived directly from the GSRD designs located at RocketBoards.org:

- For more information about the GSRD for the Agilex 7 FPGA I-Series Transceiver-SoC Development Kit, refer to the following URL https://www.rocketboards.org/foswiki/Documentation/AgilexSoCGSRDSIAGI027
- For more information about the GSRD for the Arria 10 SX SoC FPGA Development Kit (DK-SOC-10AS066S), refer to the following URL: https://www.rocketboards.org/foswiki/Documentation/arria10SoCGSRD

The GSRD designs have been modified to include the FPGA AI Suite IP. All unnecessary logic has been removed, which provides a concise design example.

The main FPGA AI Suite SoC design example is contained within a single Platform Designer system, called **system**. Double-click this node in the Quartus Prime Project Navigator to launch Platform Designer.

Related Information

- *GSRD for Agilex 7 I-Series Transceiver-SoC DevKit (4x F-Tile)* at RocketBoards.org
- *Arria 10 SoC GSRD* at RocketBoards.org

6.4.1. Clock Domains

There are three main clocks within this design. All the clocks are considered asynchronous to each other. The SDC file provided has the clocking constraints for this design.

The design clocks are as follows:

- **100MHz Board clock**

  This clock is used for all mSGDMA infrastructure and CPU CSR interfaces. The HPS AXI interfaces all run off this clock.

- **200MHz DLA clock**

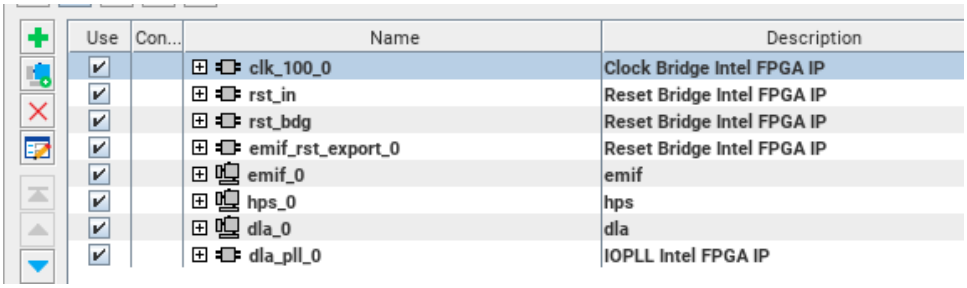
  This clock is used only by the FPGA AI Suite IP. It feeds the `dla_clk` pin and is used inside FPGA AI Suite IP PE array.

- **266MHZ DDR Clock**

  This is used for the DDR controller and interconnect between the DLA and DDR. This interface is used by the DLA to transfer workloads back and forth to system memory.

6.5. The SoC Design Example Platform Designer System

At the center of the SoC design example is the Platform Designer system.



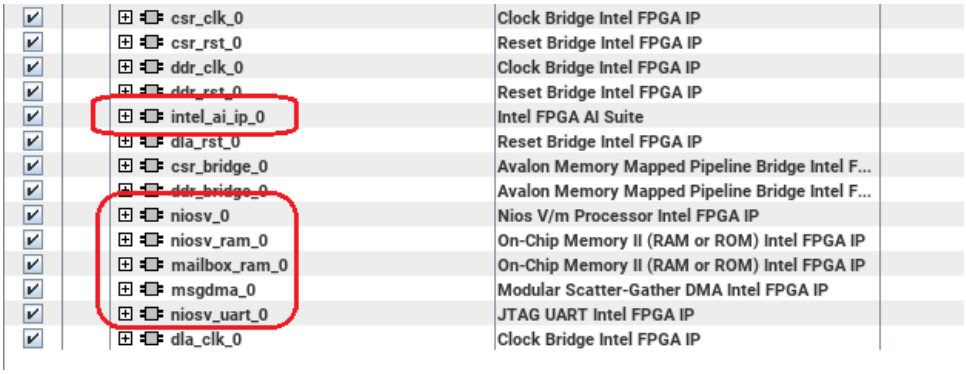
In Platform Designer, the SoC design example is separated into three hierarchical layers, the:

- **emif_0** : This layer contains the FPGA DDR4 External Memory Interface
- **dla_0** : This layer contains all the DLA IP and infrastructure IP
- **hps_0** : This layer contains all the ARM-HPS, ARM-EMIF and infrastructure IP for the ARM

The division of hierarchy demonstrates the sections of the design that are relevant to the solution. For example, if you want to target another board with a different external memory interface, you need to edit only the **emif_0** layer.

6.5.1. The dla_0 Platform Designer Layer (`dla.qsys`)

The **dla_0** layer contains the FPGA AI Suite IP and the Nios V subsystem to provide streaming capabilities.

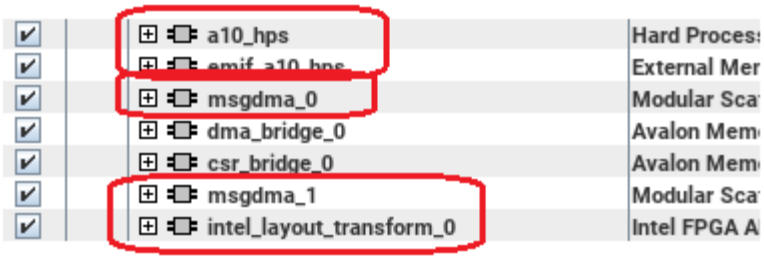


When incorporating the FPGA AI Suite IP into a custom design, you can use the `dla.qsys` file as a starting point for the new design.

6.5.2. The hps_0 Platform Designer Layer (`hps.qys`)

The **hps_0** layer contains the HPS, an mSGDMA instance (`msgdma_0`) for the FPGA AI Suite runtime, and an mSGDMA instance (`msgdma_1`) for the streaming generation app (S2M variant only).

The example layout transform is also located here and can be replaced by your version.



6.6. Fabric EMIF Design Component

The design provides a 266MHz DDR4-64Bit Avalon-based memory controller. This EMIF is used solely by the DLA.

The FPGA AI Suite IP memory interface is configured to be 512 bits wide. The EMIF interface is setup to complement this configuration.

6.7. PLL Configuration

The FPGA AI Suite IP is designed to operate at high f_{MAX} rates in Intel FPGA devices. The SoC design example provides an IOPLL that provides the IP with a fast clock.

In the design example the external board 100 MHz reference clock is fed into the PLL and a 200 MHz output clock is produced. This clock feeds the FPGA AI Suite IP directly.

You can remove or alter this PLL if you want to profile different performance curves of the system.

The FPGA AI Suite `dla_benchmark` application has no runtime method to dynamically determine the PLL operating frequency in the SOC design. The frequency of 200 MHz has been set as a constant in this application source code. If you alter the PLL frequency, you must also alter the `dla_benchmark` application to match the new clock frequency. This matching ensures that the benchmark metrics accurately reflect the performance.

See `dla_mmd_get_coredla_clock_freq` in `$COREDLA_WORK/runtime/coredla_device/mmd/hps_platform/acl_hps.cpp` and change the return value accordingly.

7. FPGA AI Suite Soc Design Example Software Components

The FPGA AI Suite SoC design example contains a software environment for the runtime flow.

The software environment for the supported FPGA development kits consists of the following components:

- Yocto build and runtime Linux environment
- Intel Distribution of OpenVINO toolkit Version 2023.3 LTS (Inference Engine, Heterogeneous plugin)
- OpenVINO Arm CPU plugin
- FPGA AI Suite runtime plugin
- MMD hardware library

The FPGA AI Suite SoC design example contains the source files, Makefiles, and scripts to cross compile all the software for the supported FPGA development kit. The Yocto SDK provides the cross compiler, and is the first component that must be built.

The machine learning network graph is compiled separately using the OpenVINO Model Optimizer and the FPGA AI Suite compiler (`dla_compiler`) command. When you compile the graph for the FPGA AI Suite SoC design example, ensure that you specify the `--foutput-format=open_vino_hetero` and `-o <path_to_file>/CompiledNetwork.bin` options.

The AOT file from the FPGA AI Suite compiler contains the compiled network partitions for FPGA and CPU devices along with the network weights. The network is compiled for a specific FPGA AI Suite architecture and batch size.

The SoC flow does not support the Just-In-Time (JIT) flow because Arm libraries are not available for the FPGA AI Suite compiler.

An architecture file (`.arch`) describes the FPGA AI Suite IP architecture to the compiler. You must specify the same architecture file to the FPGA AI Suite compiler and to the FPGA AI Suite design example build script (`dla_build_example_design.py`).

The runtime stack cannot program the FPGA with a bitstream. The bitstream must be built into the SD card (`.wic`) image that is used to program the flash card, as described in (Optional) Create an SD Card Image (.wic) on page 10 and Writing the SD Card Image (.wic) to an SD Card on page 15.

The runtime inference on the SoC FPGA device uses the OpenVINO Arm CPU plugin. To enable fallback to the OpenVINO Arm CPU plugin for graph layers that are not supported on the FPGA, the device flag must be set to `HETERO:FPGA,CPU` during the AOT compile step and when you run the `dla_benchmark` command.

In some cases, a layer might be supported by the FPGA even though the OpenVINO Arm CPU plugin does not support the layer. This support is handled by the HETERO plugin and the layer is executed on the FPGA as expected. As an example, 3D convolution layers are not supported by the OpenVINO Arm CPU plugin but still work properly provided that the `.arch` file used for the FPGA AI Suite IP configuration has enabled support for 3D convolutions.

Related Information

- "Running the Graph Compiler" in *FPGA AI Suite Getting Started Guide*
- "Compiling a Graph" in *FPGA AI Suite Compiler Reference Manual*
- "Architecture Description File Parameters" in *FPGA AI Suite IP Reference Manual*
- "Compilation Options (`dla_compiler` Command Options)" in *FPGA AI Suite Compiler Reference Manual*

7.1. Yocto Build and Runtime Linux Environment

The Linux runtime and build environment is based on the Yocto build system. Yocto uses a build model based on the concept of layers and recipes.

The Yocto layers and recipes for FPGA AI Suite SoC design example are in `$COREDLA_ROOT/hps/ed4/yocto/`. The recipes extend the standard Golden System Reference Design (GSRD) that is used as the basis for the SoC design example system. For further details on setting up Yocto for Intel SoC devices see the GSRD documentation.

The rest of this section describes key recipes in the `metal-intel-fpga-coredla` layer.

Related Information

- https://yoctoproject.org
- https://www.rocketboards.org/foswiki/Documentation/GSRD

7.1.1. Yocto Recipe: `recipes-core/images/coredla-image.bb`

This Yocto recipe customizes the image used on the SoC design example.

The `IMAGE_INSTALL:append` section defines extra packages for the FPGA AI Suite SoC example design. In particular, the `msgdma-userio`, `uio-devices`, and `kernel-modules` recipes are enabled to support the UIO and mSGDMA. The `WKS_FILES:*` section specifies which definition is used for building the SD card image.

7.1.2. Yocto Recipe: `recipes-bsp/u-boot/u-boot-socfpga_%.bbappend`

This Yocto recipe appends to the `meta-intel-fpga/recipes-bsp` recipe and enables the FPGA to SDRAM bridge, if it is required for the device target. This bridge is not required for Arria 10 designs.

On devices that require the bridge, the bridge allows mSGDMA to access the HPS SDRAM. This access exposes the full HPS SDRAM to the FPGA device.

7.1.3. Yocto Recipe: recipes-drivers/msgdma-userio/msgdma-userio.bb

This Yocto recipe enables an out-of-kernel (userspace) build of the `msgdma-userio` character driver used for transfer data to and from the HPS to the FPGA DDR memory. This driver calls into the DMA_ENGINE API exposed by the `altera-msgdma` kernel driver. The kernel module `altera-msgdma.ko` is enabled in `recipe-kernel/linux` kernel configuration file: `recipes-kernel/linux/files/enable-coredla-mod.cfg`.

Device tree settings are set to assign base address, IRQ for the `altera-msgdma` and associated to the `msgdma-userio` driver. These can be found in `recipes-kernel/linux/files/coredla-dts.patch`.

7.1.4. Yocto Recipe: recipes-drivers/uio-devices/uio-devices.bb

This Yocto recipe installs a service which starts up the uio drivers within the system.

The `uio_pdrv_genirq` driver provides user mode access to mapping and umapping CSR registers in the FPGA AI Suite IP.

The kernel modules `uio.ko` and `uio_pdev_genirq.ko` are enabled in the `recipe-kernel/linux` kernel configuration file: `recipes-kernel/linux/files/enable-mod.cfg`.

The device tree settings are set to assign base addresses and IRQs for the FPGA AI Suite IP, the stream controller, and the layout transform module. These can be found in `recipes-kernel/linux/files/coredla-dts.patch`.

7.1.5. Yocto Recipe: recipes-kernel/linux/linux-socfpga-lts_%.bbappend

This Yocto recipe applies the `0001-altera-msgdma.patch`, which does the following fixes:

- Set the FPGA DDR `src` and `dest` addresses to allow memory to and from the device to work correctly.
- Fixes the calculation of the number of descriptors used for a transfer on an Arria 10 device.

For Agilex 7 devices, the `agilex-dts.patch` patch enables necessary drivers in the device tree.

For Arria 10 devices, the `coredla-dts.patch` patch enables necessary drivers in the device tree.

This recipe also includes `enable-coredla-mod.cfg`, which is the kernel configuration file to enable `altera-msgdma driver`, `uio`, and `uio_pdrv_genirq` drivers.

7.1.6. Yocto Recipe: `recipes-support/devmem2/devmem2_2.0.bb`

This Yocto recipe downloads, compiles, and installs The `devmem2` utility from https://github.com/radii/devmem2 for use in debugging designs. The `devmem2` utility is a simple program to read/write from/to any memory location.

7.1.7. Yocto Recipe: `wic`

This Yocto recipe contains two files that define the layout of the SD card image. One file is used for Arria 10 devices, and the other for Stratix® 10 devices (not currently supported by the SoC design example). The partitions are as follows:

vfat	Storage for the Linux kernel, device tree, FPGA image, and u-boot
ext4	Root file system
raw	Arria 10 only. A custom raw partition labeled "a2". This is used for the first-stage boot loader.

7.2. FPGA AI Suite Runtime Plugin

The FPGA AI Suite runtime plugin is described in "OpenVINO FPGA Runtime Plugin" in *FPGA AI Suite PCIe-based Design Example User Guide* .

Note that the FPGA AI Suite on Arm CPUs does not support the JIT (Just-In-Time) flow.

7.3. Runtime Interaction with the MMD Layer

The FPGA AI Suite runtime uses the MMD layer to interact with the memory-mapped device. In the SoC Example Design, the MMD layer communicates via Linux kernel drivers described in MMD Layer Hardware Interaction Library on page 59.

The key classes and structure of the runtime are described in "FPGA AI Suite Runtime" in *FPGA AI Suite PCIe-based Design Example User Guide*.

7.4. MMD Layer Hardware Interaction Library

Runtime communication with CSR registers for the stream controller and for the FPGA AI Suite IP happens via the UIO driver. See The Userspace I/O HOWTO for more information on the UIO communication model.

FPGA AI Suite IP graph weights and instructions (from the AOT file) are transferred from host DDR memory to EMIF DDR memory (allocated to the FPGA FPGA AI Suite IP) via the mSGDMA-USERIO driver (a custom kernel driver, see Yocto Recipe: recipes-drivers/msgdma-userio/msgdma-userio.bb on page 58). This driver is also used to send microcode and images. Lastly, inference results are transferred into host DDR via the mSGDMA-USERIO driver.

The source files for the library are in `runtime/coredla_device/mmd/hps_platform/`. The files contain classes for managing and accessing the FPGA AI Suite IP, the stream controller, the layout transform module, and DMA by UIO and MSGDMA-USERIO drivers. The remainder of this section describes the key classes.

7.4.1. MMD Layer Hardware Interaction Library Class `mmd_device`

This class has the following responsibilities:

- Acquire the FPGA AI Suite IP, the stream controller (S2M only) and the mSGDMA
- Register interrupt callback for the FPGA AI Suite IP
- Provide read/write CSR and DDR functionality to the FPGA AI Suite runtime
- Linux device discovery.

The class attempts discovery of the following UIO devices, each from the `/sys/class/uio/ui*/` namespace.

coredla0	This represents the FPGA AI Suite IP CSR registers.
stream_controller0	If present (S2M only), this represents the Stream Controller CSR registers.
layout_transform0	If present (S2M only), this represents the Layout Transform CSR registers. Note that this device is not directly controlled from the runtime.

The class also attempts to discover the following mSGDMA-USERIO devices.

/dev/msgdma_coredla0	This represents the mSGDMA used to transfer weights, instructions, microcode, and, during M2M operation, image data.
/dev/msgdma_stream0	This represents the mSGDMA used to transfer images to the Layout Transform during the S2M mode of operation.

7.4.2. MMD Layer Hardware Interaction Library Class `uio_device`

The Linux device tree is used for UIO devices. The following responsibilities are assumed by this file:

- Acquire and maps/unmap the FPGA CSRs to user mode, using the `sysfs` entries for UIO (`/sys/class/uio/uio*/`).
- Acquire and register a callback for the interrupt from the FPGA AI Suite IP.
- Provide `read_block()` and `write_block()` functions for accessing the CSRs.

7.4.3. MMD Layer Hardware Interaction Library Class `dma_device`

The Linux device tree is used for mSGDMA-USERIO devices. These devices provide a Linux character device for simple read/write access to/from the FPGA EMIF via `fseek()`, `fread()`, and `fwrite()`.

The following responsibilities are assumed by this class:

- Acquire the mSGDMA-USERIO driver interface.
- Provide `read_block()` and `write_block()` functions for transfers from and to the FPGA AI Suite assigned DDR memory

8. Streaming-to-Memory (S2M) Streaming Demonstration

A typical use case of the FPGA AI Suite IP is to run inferences on live input data. For instance, live data can come from a video source such as an HDMI IP core and stream to the FPGA AI Suite IP to perform image classification on each frame.

For simplicity, the S2M demonstration only simulates a live video source. The streaming demonstration consists of the following applications that run on the target SoC device:

- **`streaming_inference_app`**

  This application loads and runs a network and captures the results.

- **`image_streaming_app`**

  This application loads bitmap files from a folder on the SD card and continuously sends the images to the EMIF, simulating a running video source

The images are passed through a layout transform IP that maps the incoming images from their frame buffer encoding to the layout required by the FPGA AI Suite IP.

There is a module called the stream controller that runs on a Nios V microcontroller that controls the scheduling of the source images to the FPGA AI Suite IP.

The `streaming_inference_app` application creates OpenVINO inference requests. Each inference request is allocated memory on the EMIF for input and output buffers. This information is sent to the stream controller when the inference requests are submitted for asynchronous execution.

In its running state, the stream controller waits for input buffers to arrive from the `image_streaming_app` application. When the buffer arrives, the stream controller programs the FPGA AI Suite IP with the details of the received input buffer, which triggers the FPGA AI Suite IP to run an inference.

When an inference is complete, a completion count register is incremented within the FPGA AI Suite IP CSRs. This counter is monitored by the currently executing inference request in the `streaming_inference_app` application, and is marked as complete when the increment is detected. The output buffer is then fetched from the EMIF and the FPGA AI Suite IP portion of the inference is now complete.

Depending on the model used, there might be further processing of the output by the OpenVINO HETERO plugin and OpenVINO Arm CPU plugin. After the complete network has finished processing, a callback is made to the application to indicate the inference is complete.

The application performs some post processing on the buffer to generate the results and then resubmits the same inference request back to OpenVINO, which lets the stream controller use the same input/output memory block again.

8.1. Nios Subsystem

The stream controller module runs autonomously in a Nios V microcontroller. An interface to the module is created by the FPGA AI Suite OpenVINO plugin when an "external streaming" flag is enabled by the inference application. At startup, the interface checks that the stream controller is present by sending a ping message and waiting for a response.

The following sections describe details of the various messages that are sent between the plugin and the stream controller, along with their packet structure.

8.1.1. Stream Controller Communication Protocol

The FPGA AI Suite OpenVINO plugin running on the HPS system includes a `coredla-device` component which in turn has a stream controller interface if the "external streaming" flag is enabled by the inference application. This stream controller interface manages the communications from the HPS end to the stream controller microcode module running on the Nios V microcontroller.

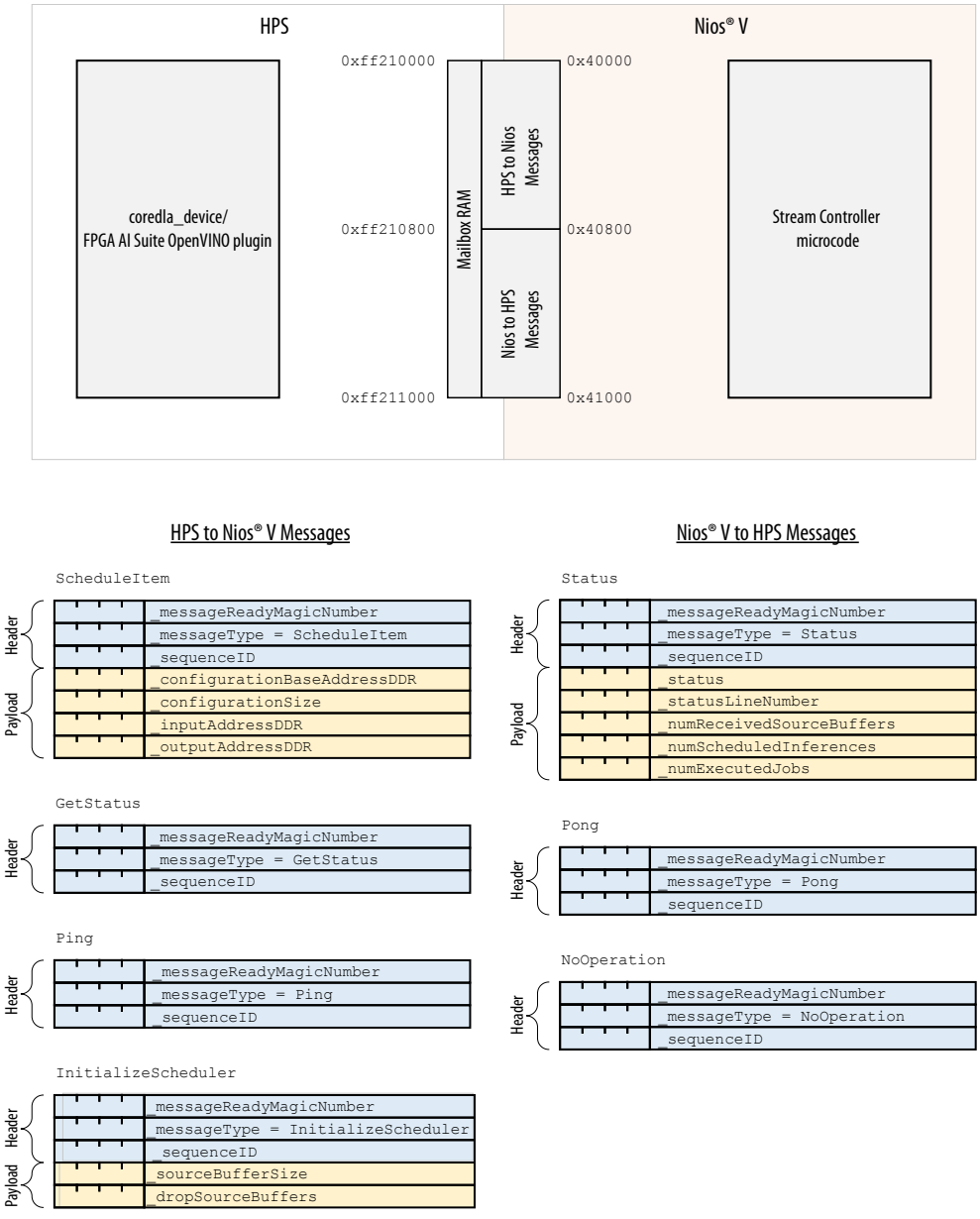
Messages are sent between the HPS and the Nios V microcontroller using the mailbox RAM which is shared between the two. In the HPS, this RAM is at physical address 0xff210000, and in the Nios V microcontroller, it is at address 0x40000. The RAM is 4K bytes. The lower 2K is used to send messages from the HPS to the Nios V microcontroller, and the upper 2K is used to send messages from the Nios V microcontroller to the HPS.

Message flow is always initiated from the HPS end, and the Nios V microcontroller always responds with a message. Therefore, after sending any message the HPS end waits until it receives a reply message. This can contain payload data (for example, status information) or just a "no operation" message with no payload.

Each message has a 3 x uint32_t header, which consists of a `messageReadyMagicNumber` field, a `messageType` field, and a `sequenceID` field. This header is followed by a payload, the size of which depends on the `messageType`. The `messageReadyMagicNumber` field is set to the value of `0x55225522` when the message is ready to be received

When a message is to be sent, all of the buffer apart from the `messageReadyMagicNumber` is first written to the mailbox RAM. The `sequenceID` increments by 1 with every message sent. Then the `messageReadyMagicNumber` is written. The sending end then waits for the value of `messageReadyMagicNumber` to change to the value of the `sequenceID`. This is set by the stream controller microcode module and indicates that the message has been received and processed by the receiving end.

Figure 11. Stream Controller Mailbox RAM and Message Packets



8.1.2. Buffer Flow in Streaming Mode using Nios V Software Scheduler

8.1.2.1. Review of M2M mode

To explain how the buffers are managed in streaming mode, it can help to review the existing flow for M2M mode.

The inference application loads source images from `.bmp` files into memory allocated from its heap. These buffers are 224x224x3 uint8_t samples (150528 bytes). During the load, the BGR channels are rearranged from interleaved channels into planes.

OpenVINO inference requests are created by the application using the inference engine. These inference requests allocate buffers in the on-board EMIF memory. The size of each of these buffers is the size of the input buffer plus the size of the output buffer. The input buffer size depends FPGA AI Suite IP parameters (specified in the `.arch` file) for which the graph was compiled.

The BGR planar image buffers are attached as input blobs to these OpenVINO inference requests, which are then scheduled for execution.

Preprocessing Steps

In M2M mode, the preprocessing steps are performed in software.

- The samples are converted to 32-bit floating point, and the mean G, B and R values of the imagenet dataset are subtracted from each sample accordingly.
- The samples are converted to 16-bit floating point.
- A layout transform then maps these samples into a larger buffer which has padding, in the layout expected by the FPGA AI Suite.

Inference Execution Steps

- The transformed image is written directly to board memory at its allocated address.
- The FPGA AI Suite IP CSR registers are programmed to schedule the inference.
- The FPGA AI SuiteOpenVINO plugin monitors the completion count register (located on the FPGA AI Suite IP), either by polling or receiving an interrupt, and waits until the count increments.
- The results buffer (2048 bytes) is read directly from the EMIF on the board to HPS memory.

Postprocessing Steps

- The samples in the results buffer (1001 16-bit floating point values) are converted to 32-bit floating point.
- The inference application receives these buffers, sorts them, and collects the top five results.

8.1.2.2. External Streaming Mode Buffer Flow

External streaming mode is enabled in the inference application by setting the configuration value `DLIA_CONFIG_KEY(EXTERNAL_STREAMING)` to `CONFIG_VALUE(YES)` in the OpenVINO FPGA plugin.

In streaming mode, the inference application does not handle any of the input buffers. It still must create inference requests, which allocate the input and output buffers in the EMIF memory as before, but no input blobs are attached to the inference requests.

When the inference request is executed, there are no preprocessing steps required, since they do not have any input blobs.

Inference Execution Steps

- Instead of writing a source buffer directly to its allocated address, a `ScheduleItem` command is sent to the Nios V stream controller which contains details of the input buffer EMIF address.
- The FPGA AI Suite IP CSR registers are **not** programmed by the plugin.
- The plugin waits for the completion count register to increment as before.
- The results buffer is read directly from the board as before.

Postprocessing Steps

- The samples in the results buffer (1001 16-bit floating point values) are converted to 32-bit floating point.
- The inference application receives these buffers, sorts them, and collects the top five results.
- The same inference request is rescheduled with the inference engine.

8.1.2.3. Nios V Stream Controller State Machine Buffer Flow

When the network is loaded into the `coredla_device`, if external streaming has been enabled, a connection to the Nios V processor is created and an `InitializeScheduler` message is sent. This message resets the stream controller and sets the size of the raw input buffers and the drop/receive ratio of buffers from the input stream.

The inference application queries the plugin for the number of inference requests to create. When scheduled with the inference engine, these send `ScheduleItem` commands to the stream controller, and a corresponding `CoreDlaJobItem` is created. The `CoreDlaJobItem` keeps details of the input buffer address and size and has flags to indicate if it has a source buffer and to indicate if it has been scheduled for inference on the FPGA AI Suite IP. The `CoreDlaJobItem` instances are handled as if they are in a circular buffer.

When the Nios V stream controller has received a `ScheduleItem` command from all of the inference requests and created a `CoreDlaJobItem` instance for each of them, it changes to a running state, which arms the mSGDMA stream to receive buffers, and sets a pointer `pFillingImageJob` that identifies which of the buffers is the next to be filled.

It then enters a loop, waiting for two types of event:

- A buffer is received through the mSGDMA, which is detected by a callback from an ISR.
- A message is received from the HPS.

New Buffer Received

The `pFillingImageJob` pointer is marked as now having a buffer.

If the next job in the circular buffer does not have a buffer, the `pFillingImageJob` pointer is moved on and the mSGDMA is armed again to receive the next buffer at the address of this next job.

If it does have a buffer, the FPGA AI Suite IP cannot keep up with the input buffer rate, so the `pFillingImageJob` does not move and the mSGDMA is armed to capture the next buffer at the same address. This means that the previous input buffer is dropped and is not processed by the FPGA AI Suite IP.

Buffers that have not been dropped can now be scheduled for inference on the FPGA AI Suite IP provided that the IP has fewer than two jobs in its pipeline.

Scheduling a job for execution means programming the CSR registers with the configuration address, the configuration size, and the input buffers address in DDR memory. This programming also sets the flag on the job so the controller knows that the job has been scheduled.

Message Received

If the message is a `ScheduleItem` message type then an inference request has been scheduled by the inference application.

This request happens only if a previous inference request has been completed and rescheduled. The number of jobs in the FPGA AI Suite IP pipeline has decreased by 1, so another job can potentially be scheduled for inference execution, providing it has an input buffer assigned.

If there are no jobs available with valid input buffers, then the FPGA AI Suite IP is processing buffers faster than they are being received by the mSGDMA stream, and consequently all input buffers are processed (that is, none are dropped).

8.2. Building the Stream Controller Module

The stream controller is built as part of the steps described in Installing HPS Disk Image Build Prerequisites on page 12. For system development that extends the FPGA AI Suite SoC design example, you might want to compile the stream controller module independently.

The stream controller module source code can be found in the distribution, in the `runtime/coredla_device/stream_controller/` directory.

There is a script `build.sh` in the source code directory that builds a binary `.hex` file. This file is then used by Quartus Prime when building the firmware to embed the microcode module.

The script should be run from a Nios V command shell, which is part of Quartus Prime. It requires a Quartus Prime project file and a Quartus Prime `.qsys` file. For this design example, the project file is `top.qpf`, and the Platform Designer file is `dla.qsys`.

An example command to build the stream controller module is as follows:

```
./build.sh top.qpf dla.qsys stream_controller.hex
```

8.3. Building the Streaming Demonstration Applications

The two streaming demonstration applications, `streaming_inference_app` and `image_streaming_app`, are built as part of the runtime and are included on the SD card image for the target device in the directory `/home/root/app/`.

8.4. Running the Streaming Demonstration

You need two terminals connected to the target device, one for each of the streaming applications.

One can be a serial terminal, and the other can be an SSH connection from a desktop PC. For details, refer to Running the S2M Mode Demonstration Application on page 29.

8.4.1. The `streaming_inference_app` Application

The `streaming_inference_app` application is an OpenVINO-based application. It loads a given precompiled ResNet50 network, then creates inference requests that are executed asynchronously by the FPGA AI Suite IP.

The resulting tensors are captured from the EMIF using the mSGDMA controller. The postprocessing required in the software involves converting the output tensors to floating point, assigning the values to the appropriate image classification, sorting the results, and selecting the top 5 classification results.

For each inference, the result is displayed on the terminal, and the results for each inference up to the 1000th one are logged in a `results.txt` file in the application folder.

The application depends on the following shared libraries . The system build adds these libraries to the directory `/home/root/app` on the SD card image, along with the application binary and a `plugins.xml` file that defines the plugins available to OpenVINO.

- `libhps_platform_mmd.so`
- `libngraph.so`
- `libinference_engine.so`
- `libinference_engine_transformations.so`
- `libcoreDLAHeteroPlugin.so`
- `libcoreDlaRuntimePlugin.so`

You also need a compiled network binary file and an `.arch` file (which describes the FPGA AI Suite IP parameterization) to run inferences. These have been copied to the `/home/root/resnet-50-tf` directory.

For example, a ResNet50 model compiled for an Arria 10 might have the following files:

- `RN50_Performance_no_folding.bin`
- `A10_Performance.arch`

Before running the application, set the LD_LIBRARY_PATH shell environment variable to define the location of the shared libraries:

```
root@arria10-1ac87246f24f:~# cd /home/root/app
root@arria10-1ac87246f24f:~# export LD_LIBRARY_PATH=.
```

Use the `--help` of the `streaming_inference_app` command to display the command usage:

```
# ./streaming_inference_app -help
Usage:
        streaming_inference_app -model=<model> -arch=<arch> -device=<device>

Where:
        <model>    is the compiled model binary file, eg /home/root/resnet-50-tf/
RN50_Performance_no_folding.bin
        <arch>     is the architecture file, eg /home/root/resnet-50-tf/
A10_Performance.arch
        <device>   is the OpenVINO device ID, eg HETERO:FPGA or HETERO:FPGA,CPU
```

Start the streaming inference app with a command like this:

```
# ./streaming_inference_app \
-model=/home/root/resnet-50-tf/RN50_Performance_no_folding.bin \
-arch=/home/root/resnet-50-tf/A10_Performance.arch \
-device=HETERO:FPGA
```

The distribution includes a shell script utility called `run_inference_stream.sh` which calls this command above.

Note that the layout transform IP core does not support folding on the input buffer. For streaming, you must use models that have been compiled by the `dla_compiler` command with the `--ffolding-option=0` command line option specified.

8.4.2. The `image_streaming_app` Application

The `image_streaming_app` application loads images from the SD card, programs the layout transform IP, and then transfers the buffers via a streaming mSGDMA interface to the device. The buffers are sent at regular intervals at a frequency set by one of the command line options. Buffers are continually sent until the program is stopped with Ctrl+C.

The `image_streaming_app` application works only with `.bmp` files with dimensions of 224x224 pixels. The `.bmp` files can be either 24 or 32 bits per pixel format. If they are 24 bits, the buffers are padded to make them 32 bits per pixel. This format is expected by the input of the layout transform IP .

The command usage from the `-help` command option is as follows:

```
root@agilex7:~/app# ./image_streaming_app --help
Usage:
 image_streaming_app [Options]

Options:
-images_folder=folder     Location of bitmap files. Defaults to working folder.
-image=path               Location of a single bitmap file for single inference.
-send=n                   Number of images to stream. Default is 1 if -image is
set, otherwise infinite.
-rate=n                   Rate to stream images, in Hz. n is an integer. Default
is 30.
-width=n                  Image width in pixels, default = 224
-height=n                 Image height in pixels, default = 224
-c_vector=n               C vector size, default = 32
-blue_variance=n          Blue variance, default = 1.0
-green_variance=n         Green variance, default = 1.0
-red_variance=n           Red variance, default = 1.0
```

```
-blue_shift=n             Blue shift, default = -103.94
-green_shift=n            Green shift, default -116.78
-red_shift=n              Red shift, default = -123.68
```

The distribution includes a shell script utility called `run_image_stream.sh` that calls the `image_streaming_app` command with a rate of 50 Hz and default layout transform settings.

Once both applications are running, the `streaming_inference_app` application outputs the results to the terminal and a `results.txt` file.

Example output from the `streaming_inference_app` application is as follows:

```
root@agilex7:~/app# ./run_inference_stream.sh
Runtime version check is enabled.
[ INFO ] Architecture used to compile the imported model: AGX7_Performance
Using licensed IP
Read hash from bitstream ROM...
Read build version string from bitstream ROM...
Read arch name string from bitstream ROM...
Runtime arch check is enabled. Check started...
Runtime arch check passed.
Runtime build version check is enabled. Check started...
Runtime build version check passed.
Ready to start image input stream.
1 - class ID 776, score = 58.4
2 - class ID 968, score = 90.7
3 - class ID 769, score = 97.8
4 - class ID 769, score = 97.8
5 - class ID 872, score = 99.8
6 - class ID 954, score = 94.4
7 - class ID 954, score = 94.4
8 - class ID 776, score = 58.4
9 - class ID 872, score = 99.8
10 - class ID 968, score = 90.7
11 - class ID 776, score = 58.4
12 - class ID 968, score = 90.7
13 - class ID 769, score = 97.8
14 - class ID 769, score = 97.8
15 - class ID 872, score = 99.8
16 - class ID 954, score = 94.4
17 - class ID 954, score = 94.4
18 - class ID 776, score = 58.4
19 - class ID 872, score = 99.8
20 - class ID 968, score = 90.7
^C
Ctrl+C detected. Shutting down application
```

The resulting `results.txt` file from the example is as follows:

```
root@agilex7:~/app# cat results.txt
Result: image[1]
1. class ID 776, score = 58.4
2. class ID 683, score = 27.6
3. class ID 513, score = 10.2
4. class ID 432, score = 1.8
5. class ID 558, score = 1.6

Result: image[2]
1. class ID 968, score = 90.7
2. class ID 901, score = 1.1
3. class ID 868, score = 1.0
4. class ID 899, score = 1.0
5. class ID 725, score = 0.9

Result: image[3]
1. class ID 769, score = 97.8
```

```
2. class ID 845, score = 0.5
3. class ID 587, score = 0.4
4. class ID 798, score = 0.1
5. class ID 618, score = 0.1

Result: image[4]
1. class ID 769, score = 97.8
2. class ID 845, score = 0.5
3. class ID 587, score = 0.4
4. class ID 798, score = 0.1
5. class ID 618, score = 0.1
```

768979 | 2025.03.28

Send Feedback

A. FPGA AI Suite SoC Design Example User Guide Archives

For the latest and previous versions of this user guide, refer to *FPGA AI Suite SoC Design Example User Guide*. If an FPGA AI Suite software version is not listed, the user guide for the previous software version applies.

B. FPGA AI Suite SoC Design Example User Guide Document Revision History

Document Version	FPGA AI SuiteVersion	Changes
2024.12.16	2024.3	• Added "Yocto Recipe: `recipes-support/devmem2/devmem2_2.0.bb`". • Revised "Yocto Recipe: `recipes-kernel/linux/linux-socfpga-lts_%.bbappend`" (formerly "octo Recipe: `recipes-kernel/linux/linux-socfpga-lts_5.15.bbappend`")
2024.11.25	2024.3	• Updated required Quartus Prime Pro Edition version to Version 24.3. • Updated "Building the Bootable SD Card Image (`.wic`)" to Yocto project version 5.0.5.
2024.07.31	2024.2	• Revised "Agilex 7 FPGA I-Series Transceiver-SoC Development Kit Hardware Requirements". • Updated required Quartus Prime Pro Edition version to Version 24.2.
2024.03.29	2024.1	• Updated the document for Ubuntu 22.04 support. • Updated required Quartus Prime Pro Edition version to Version 23.4.
2024.02.12	2023.3.1	• Updated the document for Agilex 7 FPGA I-Series Transceiver-SoC Development Kit support, including the following new topics: — "Preparing the Intel Agilex 7 FPGA I-Series Transceiver-SoC Development Kit " — "Confirming Intel Agilex 7 FPGA I-Series Transceiver-SoC Development Kit Board Set Up" — "Programming the Intel Agilex 7FPGA Device with the JTAG Indirect Configuration (.jic) File" — "Connecting the Intel Agilex 7 FPGA I-Series Transceiver-SoC Development Kit to the Host Development System"
2023.12.01	2023.3	• Updated required Quartus Prime Pro Edition version to Version 23.3.
2023.09.06	2023.2.1	• Updated supported OpenVINO version to 2022.3.1 LTS.
2023.07.03	2023.2	• Updated supported OpenVINO version to 2022.3 LTS. • Updated OpenVINO installation paths to `/opt/intel/openvino_2023`. • Updated FPGA AI Suite installation paths to `/opt/intel/fpga_ai_suite_2023.2`. • Changed occurrences of `tools/downloader/downloader.py` to `omz_downloader`. • Changed occurrences of `tools/downloader/converter.py` to `omz_converter`.
2023.04.05	2023.1	• Initial release.