Versal ACAP VCK190 Base Targeted Reference Design

User Guide

UG1442 (v2020.2) January 8, 2021

XILINX®

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
01/08/2021 Version 2020.2	
Initial release.	N/A

Table of Contents

Introduction

The Versal™ ACAP VCK190 Base Targeted Reference Design (TRD) is an embedded video processing application partitioned among the Versal ACAP processing system (PS), the programmable logic (PL), and the artificial intelligent engines (AIE). The data flow between the control interfaces and processing system (CIPS), the PL, and the AIE is managed by a network on a chip (NOC). The design demonstrates the value of offloading computation-intensive image processing tasks such as 2D-convolution filter from the PS onto the PL or AIE. The benefits achieved are two-fold:

1. Ultra HD video stream real-time processing up to 60 frames per second
2. Freed-up CPU resources for application-specific tasks

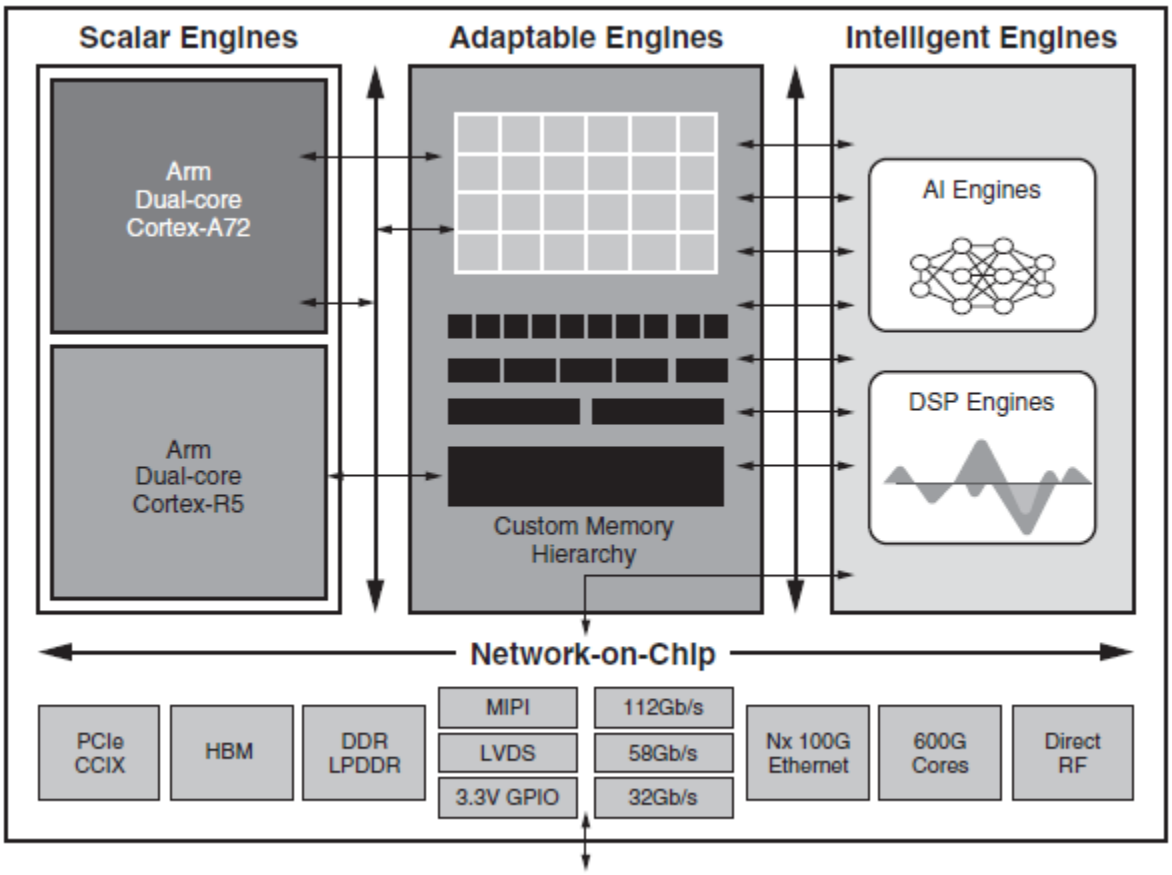
This user guide describes the architecture of the reference design and provides a functional description of its components. It is organized as follows:

- This chapter provides a high-level overview of the Versal ACAP device architecture, the reference design architecture, and key features.
- Chapter 2, *Out of Box Designs*, provides an overview of the Jupyter notebooks that are available to run designs out of the box using pre-built design images.
- Chapter 3, *Software Architecture*, describes the application, middleware, and operating system layers of the Linux software stack running on the APU.
- Chapter 4, *System Considerations*, describes boot flow and the device image required to program the ACAP.
- Chapter 5, *Hardware Architecture*, describes the hardware platform including key PS and PL peripherals.

Versal ACAP Device Architecture

The Versal™ adaptive compute acceleration platform (ACAP) is a platform that combines Scalar Engines, Adaptable Engines, and Intelligent Engines with leading-edge memory and interfacing technologies to deliver powerful heterogeneous acceleration for any application. Built on the TSMC 7 nm FinFET process technology, the Versal ACAP is the first platform to combine software programmability and domain-specific hardware acceleration with the adaptability necessary to meet today's rapid pace of innovation.

Figure 1: **Xilinx Versal ACAP Block Diagram**



The following summarizes the Versal ACAP's key features:

- Scalar Engines comprising
  - Application processing unit (APU) with 64-bit dual-core Arm® Cortex-A72 processor for compute tasks.
  - Real-time processing unit (RPU) with 32-bit dual-core Arm CortexR5-processor for low latency and deterministic operations supporting functional safety.
- Platform management controller (PMC) for securely booting and configuring the platform. It is also responsible for life-cycle management, which includes device integrity and debug, and system monitoring.
- Adaptable Engines are a combination of programmable logic blocks and memory (Block RAM, Ultra RAM) for high-compute density.
- Intelligent Engines are very large instruction word (VLIW) AI Engines for adaptive inference and DSP Engines for floating point and complex MAC operations.

- Processing system peripherals
  - Gigabit Ethernet, CAN, UART, SPI, USB, etc., to connect to external devices. The Scalar engines and these peripherals together form the Processing System (PS).
- High-speed connectivity
  - Gigabit Transceivers (GT) with a broad range of speeds up to 58 Gbps supporting multiple protocols such as PCIe, Ethernet, and Video
  - Integrated block for PCIe that supports Gen1, Gen2, Gen3 data rates at link widths of x1, x2, x4, x8, or x16, and Gen4 data rates at link widths of x1, x2, x4, or x8. The block can be configured as an Endpoint or Root Port.
  - CCIX and PCIe (CPM) has two integrated blocks for PCIe and components to support CCIX (Cache Coherent Interconnect) compliant devices. It additionally has a DMA when configured as a PCIe device.
  - Multirate Ethernet MAC (MRMAC) provides high-performance, low-latency Ethernet ports supporting a wide range of customization and statistics gathering. Supported configurations are: 1 x 100GE; 2 x 50GE; 1 x 40GE; 4 x 25GE; and 4 x 10GE.
- Integrated memory controllers that support either DDR4 or LPDDR4.
- I/Os
  - XPIO are optimized for high-performance communication including, but not limited to, interfacing to DDR4 memory through the integrated memory controller blocks.
  - High-density I/O (HDIO) banks are designed to be a cost-effective method for supporting lower-speed, higher-voltage range I/O standards.
  - MIO are multiple banks of general-purpose I/O implemented within the PS and PMC, each with a dedicated power supply. The main category of I/O are the three banks of multiplexed I/O (MIO), which can be accessed by the PS, the PMC, and the PL
- The NoC is an AXI4 based network of interconnect architecture that easily enables high-bandwidth connections to be routed around the device. The NoC extends in both horizontal and vertical directions and connects together areas of the device that demand and use large quantities of data alleviating any resource burden on the local and regional device interconnect.

For more information refer to the *Versal Architecture and Product Data Sheet: Overview* (DS950).

Reference Design Overview

The TRD built on the Versal ACAP device provides a framework for building and customizing video platforms that consist of three pipeline stages.

- Capture pipeline (input)
- Acceleration pipeline (memory-to-memory)
- Display pipeline (output)

The reference design has platforms and integrated accelerators. The platform consists of Capture pipeline and Display pipeline for Video in and Video out. This approach makes the design leaner and provides users the maximum Programmable logic (PL) for accelerator/role development. Platforms supported in this reference design are:

- Platform 1: MIPI single sensor Capture and HDMI TX display
- Platform 2: MIPI quad sensor Capture and HDMI TX display
- Platform 3: HDMI RX Capture and HDMI TX display. Along with video this platform also supports audio capture

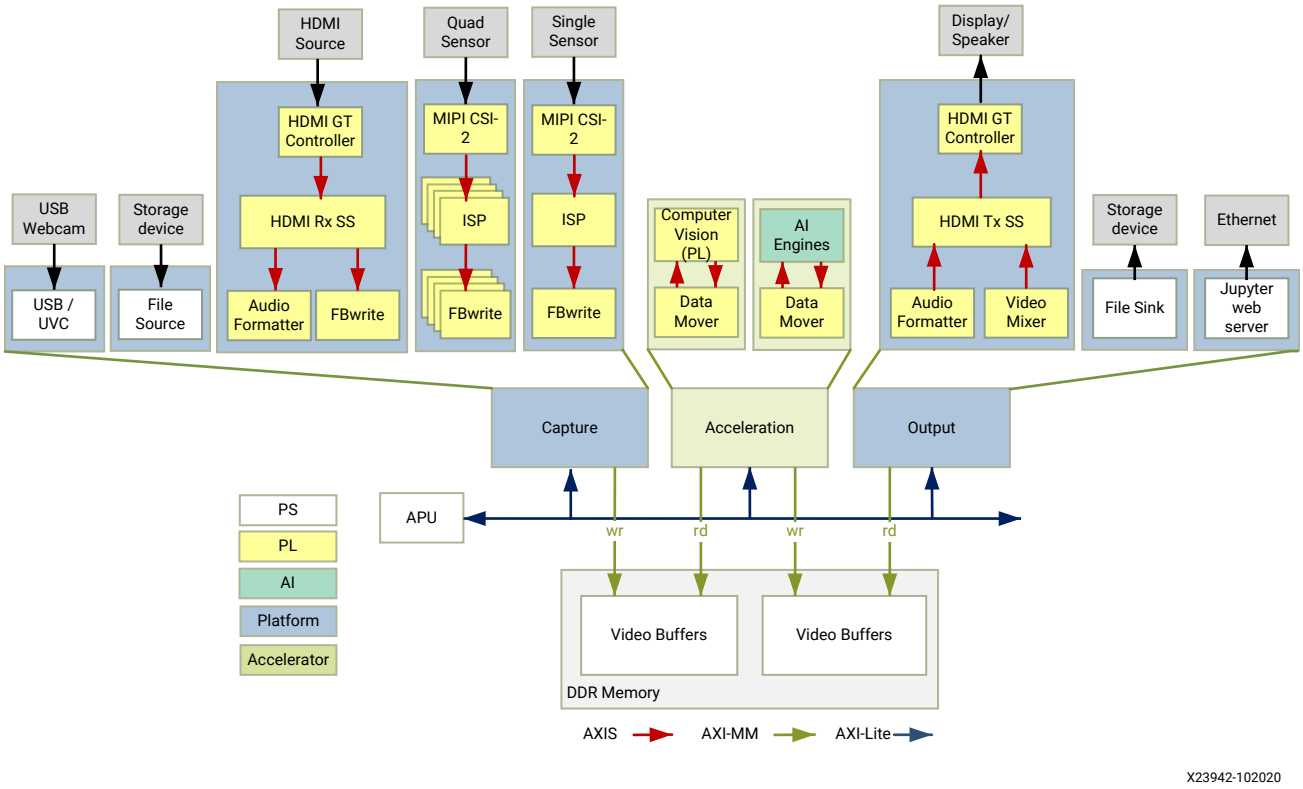
Platforms also include a virtual video device (vivid), a USB webcam, and a file as an input capture source. The platforms support audio replay from a file as well.

The following types of acceleration kernels can be run on the platforms:

- PS: Running software kernels directly on the PS (OpenCV for example)
- PL: Running HLS or RTL kernels on the PL (Vitis Vision Libraries for example)
- AIE+PL: Running kernels on AI engines with data movers in the PL

The following figure shows the various platforms supported by the design.

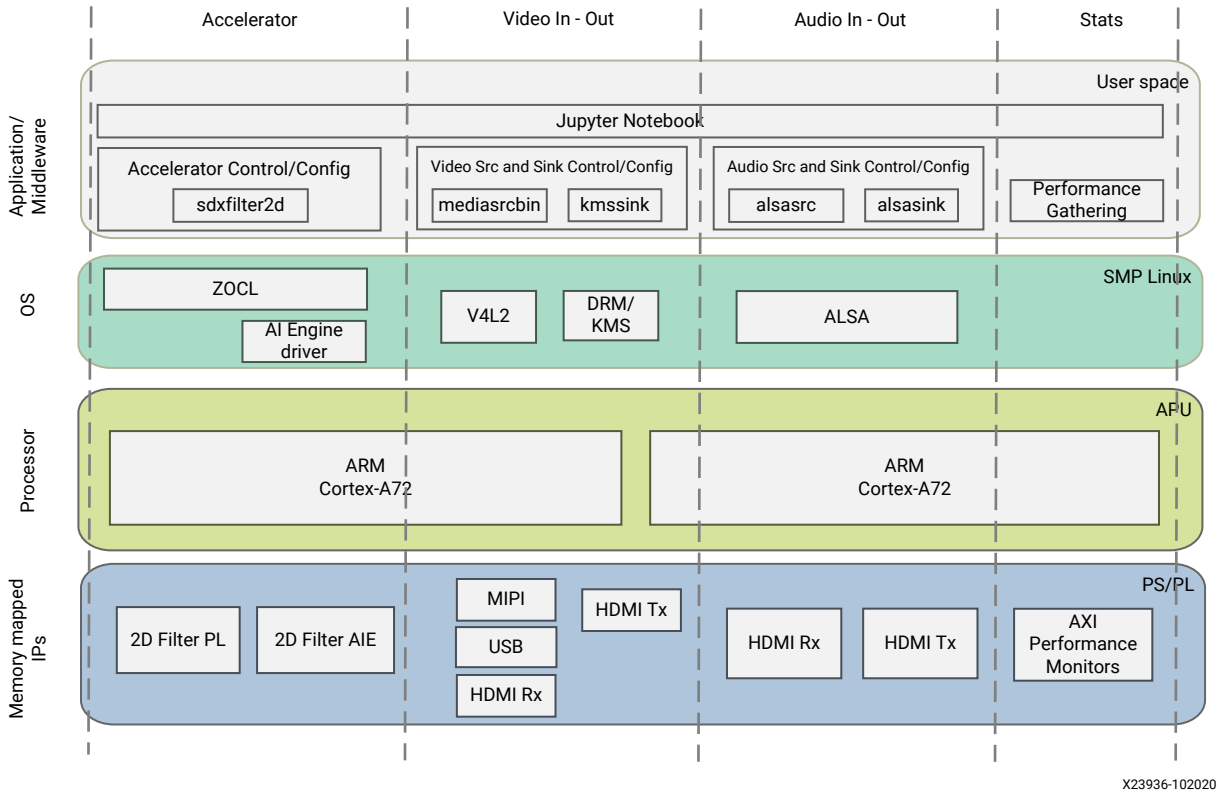
Figure 2: **Base TRD Block Diagram**



The application processing unit (APU) in the Versal ACAP consists of two Arm Cortex-A72 cores and is configured to run in SMP (symmetric multi-processing) Linux mode in the reference design. The application running on Linux is responsible for configuring and controlling the audio/video pipelines and accelerators using Jupyter notebooks. It also communicates with the APU to visualize system performance.

The following figure shows the software state after the boot process has completed and the individual applications have been started on the APU. Details are described in Chapter 3: Software Architecture.

Figure 3: **Key Reference Design Components**



The APU application controls the following video data paths implemented in a combination of the PS and PL:

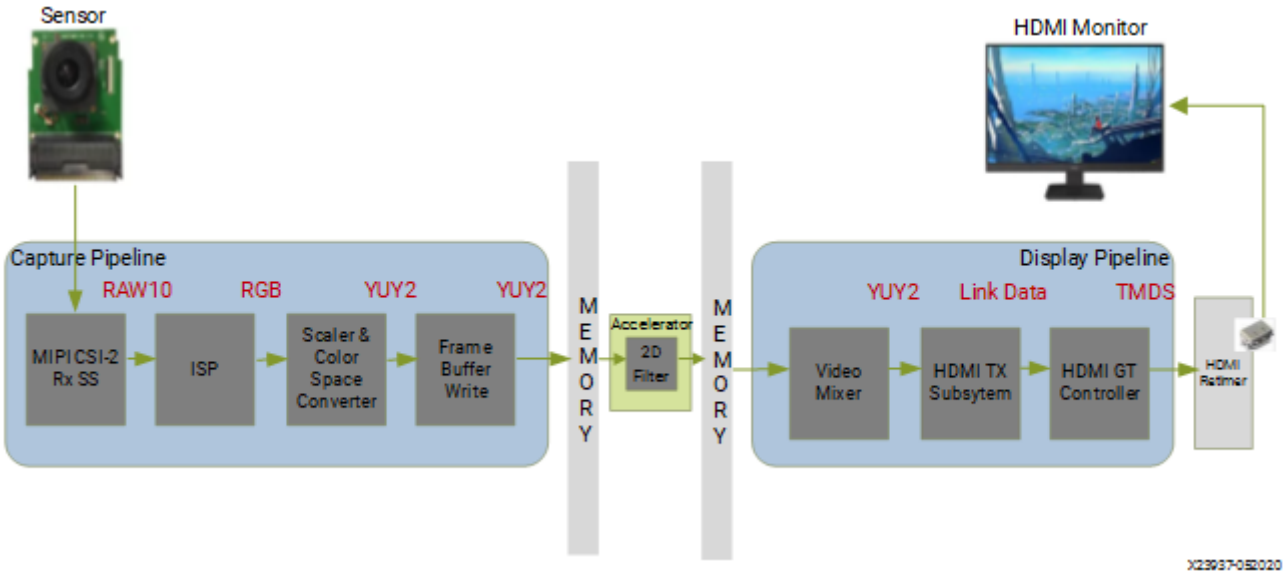
- Capture pipeline capturing video frames into DDR memory from
  - A file on a storage device such as an SD card
  - A USB webcam using the USB interface inside the PS
  - An image sensor on an FMC daughter card connected via MIPI CSI-2 Rx through the PL
  - A quad image sensor on an FMC daughter card connected via MIPI CSI-2 Rx through the PL
  - An HDMI source such as a laptop connected via the HDMI Rx subsystem through the PL. HDMI Rx also captures audio along with video.
- Memory-to-memory (M2M) pipeline implementing typical video processing algorithms
  - A 2D convolution filter – In this reference design this algorithm is implemented in the PS, PL and AIE. Video frames are read from DDR memory, processed by the accelerator, and then written back to memory.

- Display pipeline reading video frames from memory and sending them to a monitor by means of the HDMI TX subsystem through the PL. Along with video, the HDMI TX subsystem also forwards audio data to a HDMI speaker.

The APU reads performance metrics from the AXI performance monitors (APM) and sends the data to the Jupyter notebook to be displayed.

The following figure shows an example end-to-end pipeline with a single image sensor as the video source, 2D convolution filter as an accelerator, and HDMI display as the video sink. The figure also shows the image processing blocks used in the capture path. The video format in the figure is the output format on each block. Details are described in the *Hardware Architecture* chapter.

Figure 4: **End-to-End Pipeline from Video In to Video Out**



***Note*:** The audio works in a pass-through mode, RX to TX. There is no processing done on the audio data.

Reference Design Key Features

The following summarizes the TRD's key specifications:

- Target platforms and extensions
  - VCK190 evaluation board. See the *VCK180 Evaluation Board User Guide* (UG1366) for detailed information about the board.
  - Avnet Quad Sensor FMC daughter card (2 Megapixel per sensor)
  - Leopard Sony IMX274 Single Sensor FMC card (8 Megapixel)

- Xilinx® tools
  - Vivado® Design Suite
  - Vitis™ Unified Software platform
  - Petalinux tools
- Hardware interfaces and IP
  - Video inputs
    - File
    - USB webcam
    - MIPI CSI-2 Rx
    - HDMI Rx
  - Video outputs
    - HDMI Tx
    - File
    - Ethernet web server (Jupyter notebook)
  - Audio inputs
    - HDMI Rx
    - File
  - Audio outputs
    - HDMI Tx
  - Video processing
    - 2D convolution filter
  - Auxiliary Peripherals
    - SD
    - I2C
    - UART
    - Ethernet
    - General purpose I/O (GPIO)
- Software components
  - Operating system
    - APU: SMP Linux

    - Linux kernel subsystems
        - Video source: Video4 Linux (V4L2)
        - Display: Direct Rendering Manager (DRM)/Kernel Mode Setting (KMS)
    - Linux user space frameworks
        - Jupyter
        - GStreamer
        - OpenCV
        - Xilinx run-time (XRT)
- Supported video formats
    - Resolutions
        - 1080p60
        - 2160p60
        - Lower resolution and lower frame rates for USB and file I/O
    - Pixel format
        - YUV 4:2:2 (16 bits per pixel)

Out of Box Designs

This TRD consists of Jupyter notebooks (NBs) which allow you to evaluate the TRD "out of the box". A short summary of what pipeline is executed by each notebook follows.

Notebooks

- **NB1 – Encoded Video File Playback:** Demonstrates how to create a GStreamer video pipeline to decode a video file. The video is displayed in the notebook.
- **NB2 – V4L2 Video Capture:** Demonstrates how to capture video from a V4L2 device. Supported V4L2 devices include a virtual video test driver, a USB camera, and MIPI. The video is displayed in the Jupyter notebook.
- **NB3 – DRM/KMS Display:** Shows how to capture video from a V4L2 device and display the output on a monitor using a DRM/KMS display device. The Xilinx DRM/KMS driver is used by the display device. Video mixer and HDMI encoder are implemented inside the PL.
- **NB4 – Parallel Video Pipelines:** Shows how to create two parallel GStreamer video pipelines. The first pipeline captures video from a V4L2 device and the second pipeline decodes the video file and displays the output on the same DRM/KMS display device.
- **NB5: MIPI Quad Sensor:** Shows how to create four parallel GStreamer video pipelines. All four pipelines capture video from a V4L2 device and displays the output on a DRM/KMS display device.
- **NB6 – Filter2D:** Shows the process of 2D filtering with PS/PL/AIE implementations. Uses GStreamer to construct the pipeline.
- **NB7 – Pipeline Splitting:** Shows the process of splitting video pipelines and running them through 2D filters. Input is taken from eligible V4L2 devices and filtered with PS/PL filters. The pipeline is constructed with GStreamer.
- **NB8 – HDMI Audio:** Shows how to capture video and audio and forward them to HDMI TX. The pipeline is constructed with GStreamer.
- **APM Monitoring:** Displays read/write throughput metrics for specific slots in the PL design by configuring soft APMs. Data is plotted in a graph.
- **CPU Monitoring:** Uses the psutil library to provide CPU metrics. Data is plotted in a graph.
- **Power Monitoring:** Provides power metrics for various rails using the INA226 power monitors on the board. Data is plotted in a graph. Refer to the *VCK190 Evaluation User Guide* for details.

The Versal Base TRD documentation provides additional content including:

- Instructions for running the pre-built SD card image on the evaluation board
- Prerequisites for building and running the reference design
- Detailed step-by-step instructions on how to build platforms with integrated accelerators

Design Components

The reference design zip file can be downloaded from the *Versal AI Core Series VCK190 Evaluation Kit* website at https://www.xilinx.com/products/boards-and-kits/vck190.html.

The reference design zip file has the following contents:

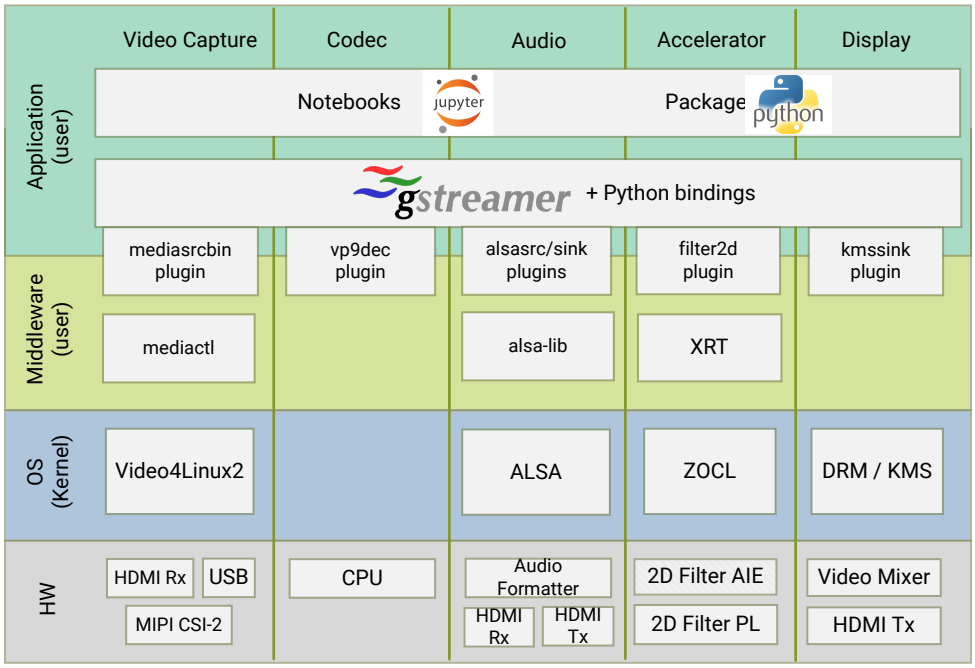
- Documentation (html webpages)
- Petalinux Board Support Package (BSP)
- Pre-built SD card image
- Vivado hardware design project
- Vitis platform
- Vitis accelerator projects
- README file
- Design sources and licenses zip file

Software Architecture

Introduction

This chapter describes the application processing unit (APU) Linux software stack. The stack and vertical domains are shown in the following figure.

Figure 5: **APU Linux Software Stack and Vertical Domains**



The stack is horizontally divided into the following layers:

- Application layer (user-space)
  - A series of Jupyter notebooks with a simple control and visualization interface
  - GStreamer multimedia framework with python bindings for video pipeline control

- Middleware layer (user-space)
  - Implements and exposes domain-specific functionality by means of GStreamer plugins to interface with the application layer
  - Provides access to kernel frameworks
- Operating system (OS) layer (kernel-space)
  - Provides a stable, well-defined API to user-space
  - Includes device drivers and kernel frameworks (subsystems)
  - Access to hardware IPs

Vertically, the software components are divided by domain:

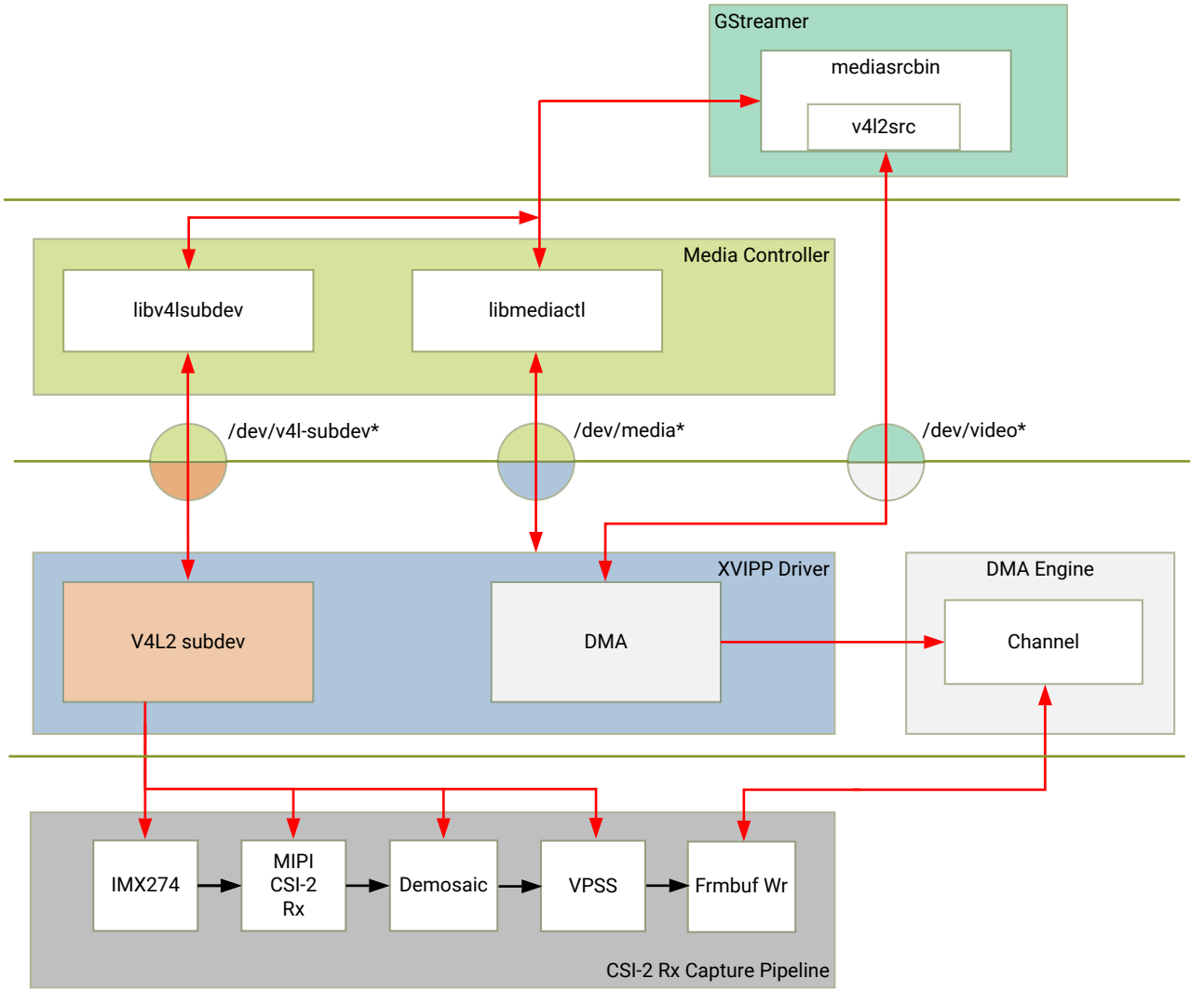
- Video capture
- Codec
- Accelerator
- Display
- Audio

The subsequent chapters describe the components of each vertical domain first and cover application layer components next.

Video Capture

The Video Capture software stack is depicted in the following figure using the single-sensor MIPI CSI capture pipeline as an example.

Figure 6: **Video Capture Software Stack**



The software stack looks similar for a Quad-sensor MIPI CSI capture pipeline as well. At a high-level it consists of the following layers from top to bottom:

- User-space layers
  - GStreamer: Media source bin plugin (wrapper around generic v4l2src plugin)
  - Media controller: Library to configure v4l subdevices and media devices
- Kernel-space layers
  - V4L2/Media subsystems: Xilinx video IP pipeline (XVIPP) driver
  - DMA engine: Xilinx framebuffer driver

Media Source Bin GStreamer Plugin

The *mediasrcbin* plugin is designed to simplify the usage of live video capture devices in this design, otherwise the user must take care of initialization and configuration. The plugin is a bin element that includes the standard *v4l2src* GStreamer element. It configures the media pipelines of the supported video sources in this design.

The *v4l2src* element inside the mediasrcbin element interfaces with the *V4L2* Linux framework and the Xilinx VIPP driver through the video device node. The mediasrcbin element interfaces with the *Media Controller* Linux framework through the *v412-subdev* and *media* device nodes which allows you to configure the media pipeline and its sub-devices. It uses the *libmediactl* and *libv4l2subdev* libraries which provide the following functionality:

- Enumerate entities, pads and links
- Configure sub-devices
    - Set media bus format
    - Set dimensions (width/height)
    - Set frame rate
    - Export sub-device controls

The *mediasrcbin* plugin sets the media bus format and resolution on each sub-device source and sink pad for the entire media pipeline. The formats between pads that are connected through links need to match. Refer to the *Media Framework* section for more information on entities, pads and links.

Kernel Subsystems

In order to model and control video capture pipelines such as the ones used in this TRD on Linux systems, multiple kernel frameworks and APIs are required to work in concert. For simplicity, we refer to the overall solution as Video4Linux (V4L2) although the framework only provides part of the required functionality. The individual components are discussed in the following sections.

Driver Architecture

The *Video Capture Software Stack* figure in the *Capture* section shows how the generic V4L2 driver model of a video pipeline is mapped to the single-sensor MIPI CSI-2 Rx capture pipelines. The video pipeline driver loads the necessary sub-device drivers and registers the device nodes it needs, based on the video pipeline configuration specified in the device tree. The framework exposes the following device node types to user space to control certain aspects of the pipeline:

- Media device node: /dev/media*
- Video device node: /dev/video*

- V4L2 sub-device node: /dev/v4l-subdev*

Media Framework

The main goal of the media framework is to discover the device topology of a video pipeline and to configure it at run-time. To achieve this, pipelines are modeled as an oriented graph of building blocks called entities connected through pads.

An entity is a basic media hardware building block. It can correspond to a large variety of blocks such as physical hardware devices (e.g. image sensors), logical hardware devices (e.g. soft IP cores inside the PL), DMA channels or physical connectors. Physical or logical devices are modeled as sub-device nodes and DMA channels as video nodes.

A pad is a connection endpoint through which an entity can interact with other entities. Data produced by an entity flows from the entity's output to one or more entity inputs. A link is a point-to-point oriented connection between two pads, either on the same entity or on different entities. Data flows from a source pad to a sink pad.

A media device node is created that allows the user space application to configure the video pipeline and its sub-devices through the *libmediactl* and *libv4l2subdev* libraries. The media controller API provides the following functionality:

- Enumerate entities, pads and links
- Configure pads
  - Set media bus format
  - Set dimensions (width/height)
- Configure links
  - Enable/disable
  - Validate formats

The following figures show the media graphs for MIPI CSI-2 Rx (single-sensor and quad-sensor) as well as the HDMI Rx video capture pipeline as generated by the *media-ctl* utility. The sub-devices are shown in green with their corresponding control interface base address and sub-device node in the center. The numbers on the edges are pads and the solid arrows represent active links. The yellow boxes are video nodes that correspond to DMA channels, in this case write channels (outputs).

Figure 7: **Video Capture Media Pipeline: Single MIPI CSI-2 RX**

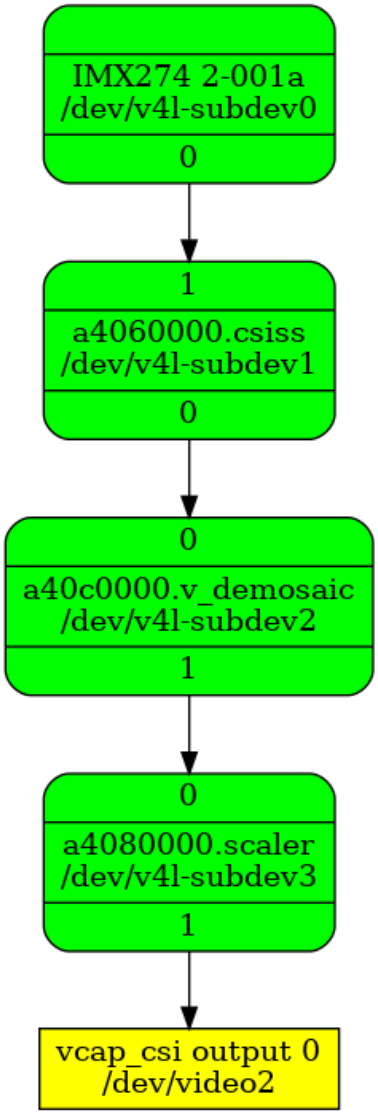


Figure 8: **Video Capture Media Pipeline: Quad MIPI CSI-2 Rx**

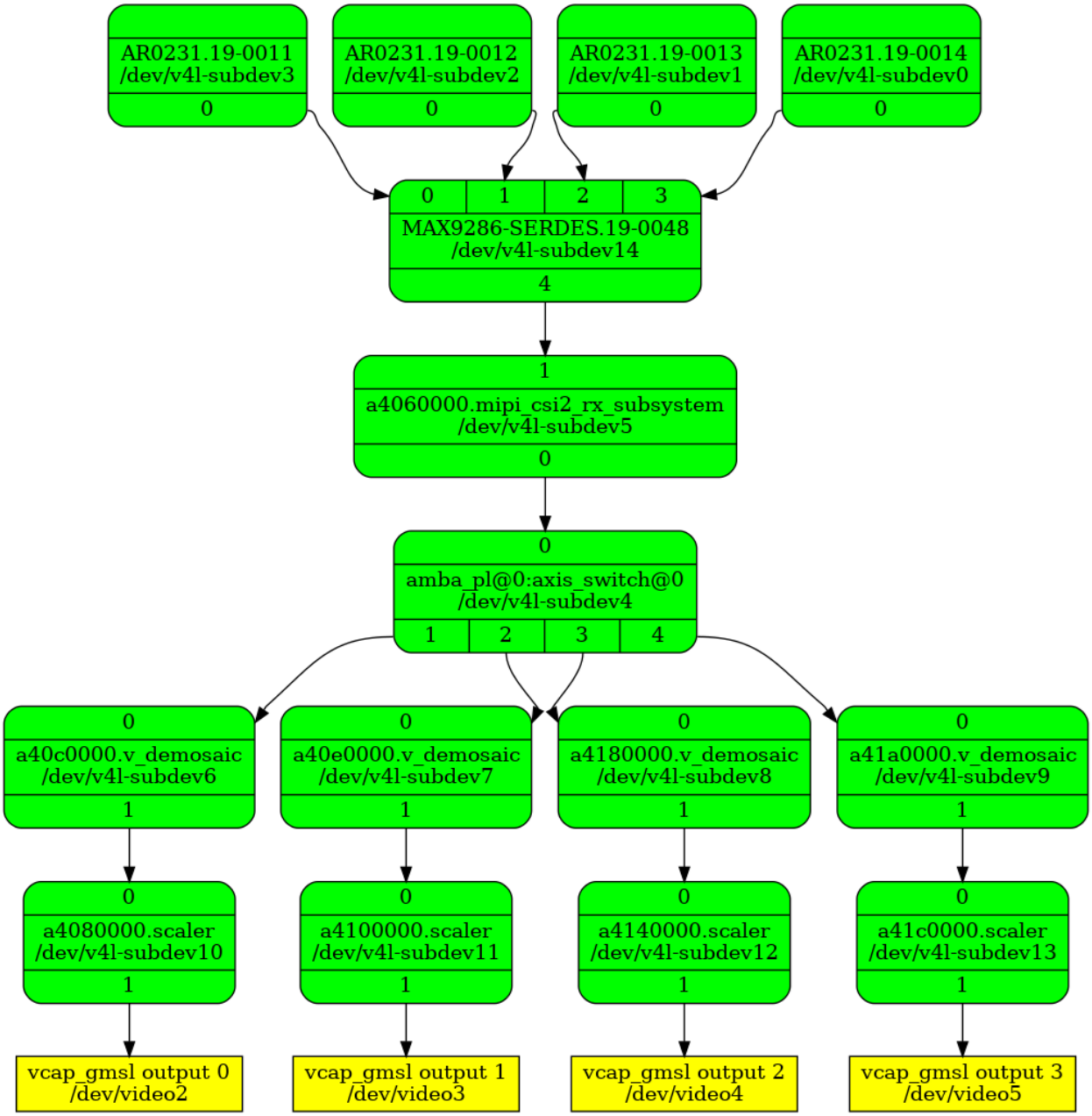
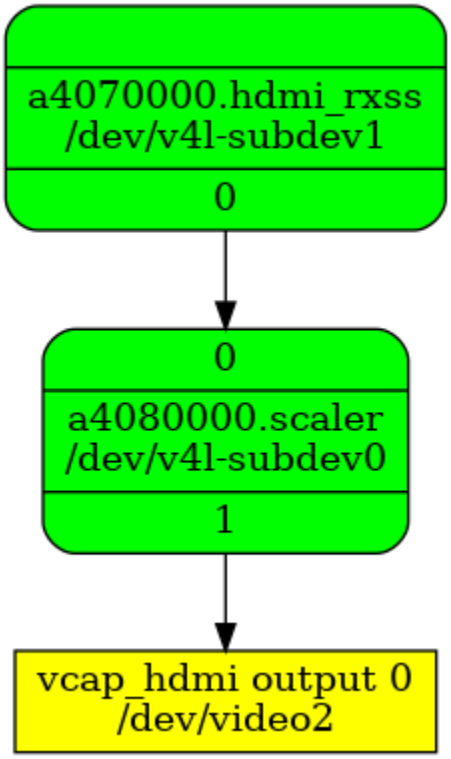


Figure 9: **Video Capture Media Pipeline: HDMI RX**



V4L2 Framework

The V4L2 framework is responsible for capturing video frames at the video device node, typically representing a DMA channel, and making those video frames available to user space. The framework consists of multiple sub-components that provide certain functionality.

Before video frames can be captured, the buffer type and pixel format need to be set using the VIDOC_S_FMT ioctl. On success the driver can program the hardware, allocate resources, and generally prepare for data exchange. Optionally, you can set additional control parameters on V4L devices and sub-devices. The V4L2 control framework provides ioctls for many commonly used, standard controls such as brightness and contrast.

The videobuf2 API implements three basic buffer types but only physically contiguous memory is supported in this driver because of the hardware capabilities of the Frame Buffer Write IP. Videobuf2 provides a kernel internal API for buffer allocation and management as well as a user-space facing API. VIDIOC_QUERYCAP and VIDIOC_REQBUFS ioctls are used to determine the I/O mode and memory type. In this design, the streaming I/O mode in combination with the DMABUF memory type is used.

DMABUF is dedicated to sharing DMA buffers between different devices, such as V4L devices or other video-related devices such as a DRM display device (see the *GStreamer Pipeline Control* section). In DMABUF, buffers are allocated by a driver on behalf of an application. These buffers are exported to the application as file descriptors.

For capture applications, it is customary to queue a number of empty buffers using the VIDIOC_QBUF ioctl. The application waits until a filled buffer can be de-queued with the VIDIOC_DQBUF ioctl and re-queues the buffer when the data is no longer needed. To start and stop capturing applications, the VIDIOC_STREAMON and VIDIOC_STREAMOFF ioctls are used.

The ioctls for buffer management, format and stream control are implemented inside the *v4l2src* plugin and the application developer does not need to know the implementation details.

Video IP Drivers

Xilinx adopted the V4L2 framework for most of its video IP portfolio. The currently supported video IPs and corresponding drivers are listed under V4L2. Each V4L driver has a sub-page that lists driver-specific details and provides pointers to additional documentation. The following table provides a quick overview of the drivers used in this design.

Table 1: **V4L2 Drivers Used in Capture Pipelines**

Linux Driver	Function
Xilinx Video Pipeline (XVIPP)	• Configures video pipeline and register media, video and sub-device nodes. • Configures all entities in the pipeline and validate links. • Configures and controls DMA engines (Xilinx Video Framebuffer Write). • Starts/stops video stream.
Xilinx Video Processing Subsystem (Scaler Only configuration)	• Sets media bus format and resolution on input pad. • Sets media bus format and resolution on output pad. (Output configuration can be different from the input configuration as the block enables color space conversion and scaling).
MIPI CSI-2 Rx	• Sets media bus format and resolution on input pad. • Sets media bus format and resolution on output pad.
Xilinx Video Image Signal Processing (ISP)	• Sets media bus format and resolution on input pad. • Sets media bus format and resolution on output pad.
Sony IMX274 Image Sensor	• Sets media bus format and resolution on output pad. • Sets sensor control parameters: exposure, gain, test pattern, vertical flip.

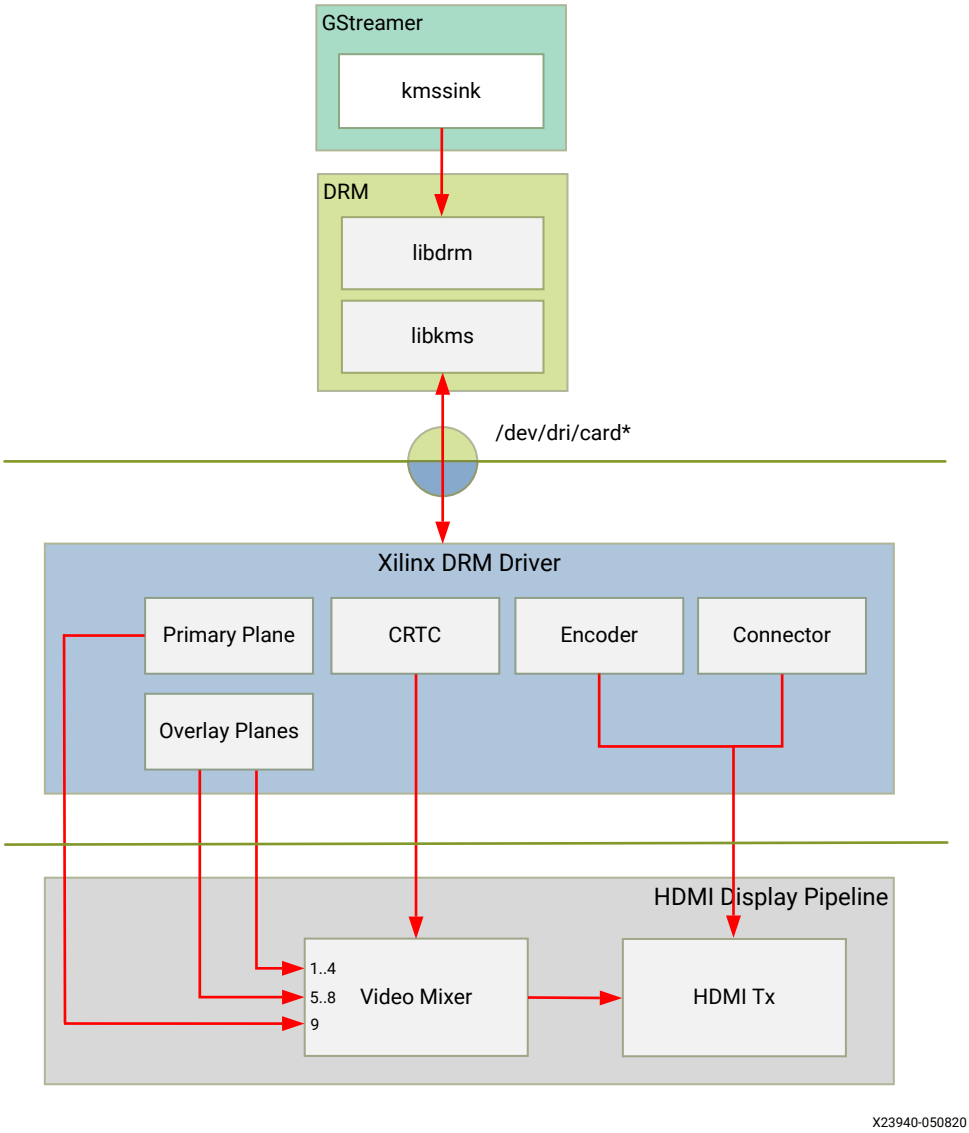
Table 1: **V4L2 Drivers Used in Capture Pipelines** *(cont'd)*

Linux Driver	Function
OnSemi AR0231 Image Sensor	• Sets media bus format and resolution on output pad. • Sets sensor control parameters: exposure, gain, test pattern, h/v flip, r/g/b balance.
MAX9286 GMSL Deserializer	• Sets media bus format and resolution on input pad. • Sets media bus format and resolution on output pad.
AXI-Stream Switch	• Sets media bus format and resolution on input pad. • Sets media bus format and resolution on output pad.
HDMI Rx Subsystem	• Query digital video (DV) timings on output pad. • Sets media bus format and resolution on output pad.

Display

The Display software stack is depicted in the following figure.

Figure 10: **Display Software Stack**



At a high-level it consists of the following layers from top to bottom which are further described in the next sections:

- User-space layers
  - GStreamer: KMS sink plugin
  - libdrm: DRM user-space library
- Kernel-space layers
  - DRM/KMS subsystem: Xilinx DRM driver
  - DMA engine: Xilinx framebuffer driver

KMS Sink GStreamer Plugin

The *kmssink* element interfaces with the DRM/KMS Linux framework and the Xilinx DRM driver through the *libdrm* library and the *dri-card* device node.

The *kmssink* element library uses the *libdrm* library to configure the cathode ray tube controller (CRTC) based on the monitor's extended display identification data (EDID) information with the video resolution of the display. It also configures plane properties such as the alpha value.

Libdrm

The DRM/KMS framework exposes two device nodes to user space: the */dev/dri/card** device node and an emulated */dev/fb** device node for backward compatibility with the legacy *fbdev* Linux framework. The latter is not used in this design. *libdrm* was created to facilitate the interface of user space programs with the DRM subsystem. This library is merely a wrapper that provides a function written in C for every ioctl of the DRM API, as well as constants, structures and other helper elements. The use of *libdrm* not only avoids exposing the kernel interface directly to user space, but presents the usual advantages of reusing and sharing code between programs.

DRM/KMS Kernel Subsystem

Linux kernel and user-space frameworks for display and graphics are intertwined and the software stack can be quite complex with many layers and different standards/APIs. On the kernel side, the display and graphics portions are split with each having their own APIs. However, both are commonly referred to as a single framework: DRM/KMS.

This split is advantageous, especially for SoCs that often have dedicated hardware blocks for display and graphics. The display pipeline driver responsible for interfacing with the display uses the kernel mode setting (KMS) API and the GPU responsible for drawing objects into memory uses the direct rendering manager (DRM) API. Both APIs are accessed from user-space through a single device node.

A brief overview of the DRM is provided but the focus is on KMS as there is no GPU present in the design.

Direct Rendering Manager

The Xilinx DRM driver uses the GEM (Graphics Execution Manager) memory manager and implements DRM PRIME buffer sharing. PRIME is the cross-device buffer sharing framework in DRM. To user-space PRIME buffers are DMABUF-based file descriptors. The DRM GEM/CMA helpers use the Continuous Memory Access (CMA) allocator as a means to provide buffer objects that are physically contiguous in memory. This is useful for display drivers that are unable to map scattered buffers via an I/O memory management unit (IOMMU).

Frame buffers are abstract memory objects that provide a source of pixels to scan out to a CRTC. Applications explicitly request the creation of frame buffers and receive an opaque handle that can be passed to the KMS CRTC control, plane configuration, and page flip functions.

Kernel Mode Setting

Mode setting is an operation that sets the display mode including video resolution and refresh rate. It was traditionally done in user-space by the X-server which caused a number of issues due to accessing low-level hardware from user-space which, if done incorrectly, can lead to system instabilities. The mode setting API was added to the kernel DRM framework, hence the name *kernel mode setting*.

The KMS API is responsible for handling the frame buffer and planes, setting the mode, and performing page-flips (switching between buffers). The KMS device is modeled as a set of planes, CRTCs, encoders, and connectors as shown in the *Display Software Stack* figure in the *Display* section. The figure also shows how the driver model maps to the physical hardware components inside the HDMI Tx display pipeline

CRTC

CRTC is an antiquated term that stands for cathode ray tube controller, which today would be simply named display controller as CRT monitors have disappeared and many other display types are available. The CRTC is an abstraction that is responsible for composing the frame to be scanned out to the display and setting the mode of the display.

In the Xilinx DRM driver, the CRTC is represented by the video mixer. The bottom-most plane is the primary plane (or master layer) and configured statically in the device-tree. The primary plane always matches the currently configured display resolution set by the CRTC (width and height) with X- and Y-offsets set to 0. The primary plane can be overlayed with up to eight overlay planes inside the video mixer.

Plane

In this design, the primary plane can be overlayed and/or alpha-blended with up to eight additional planes inside the video mixer. The z-order (foreground or background position) of the planes is fixed. The global alpha mode can be configured per plane through the driver by means of custom KMS properties: an alpha value of 0% (or 0) means the layer is fully transparent (invisible); an alpha value of 100% (or 255) means that the layer is fully opaque.

Each overlay plane's width, height, X- and Y-offset is run-time programmable relative to the primary plane or CRTC which determines the display resolution. The pixel formats of the primary plane as well as the eight overlay planes are fixed: one BGR plane (primary) plus four YUY2 planes (overlay) plus four BGR planes (overlay) from bottom to top.

The Xilinx DRM driver supports the universal plane feature, therefore the primary plane and overlay planes can be configured through the same API. A page-flip is the operation that configures a plane with the new buffer index to be selected for the next scan-out. The new buffer is prepared while the current buffer is being scanned out and the flip typically happens during vertical blanking to avoid image tearing.

Encoder

An encoder takes pixel data from a CRTC and converts it to a format suitable for any attached connectors. There are many different display protocols defined, such as HDMI and DisplayPort. This design uses an HDMI transmitter implemented in the PL which sends the encoded video data to the HDMI GT Controller and PHY. The PHY serializes the data using the GTY transceivers in the PL before it goes out via the HDMI Tx connector on the board.

Connector

The connector models the physical interface to the display. The HDMI protocol uses a query mechanism to receive data about the monitor resolution and refresh rate by reading the extended display identification data (EDID) stored inside the monitor. This data can then be used to program the CRTC mode. HDMI also supports hot-plug events to detect if a cable has been connected or disconnected as well as handling display power management signaling (DPMS) power modes.

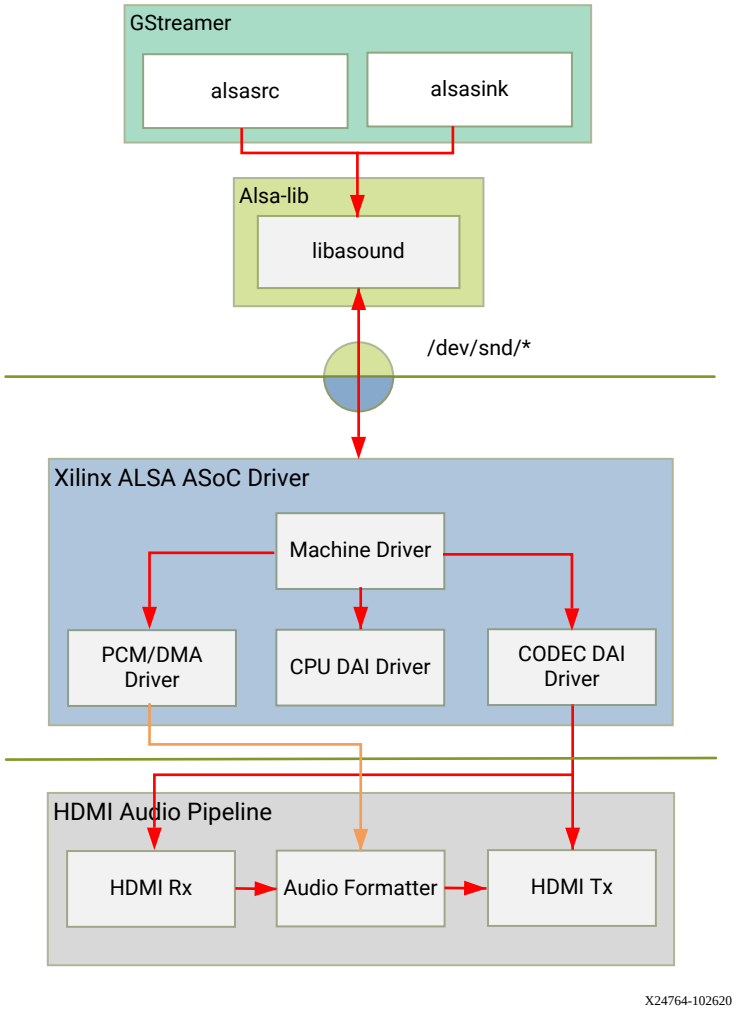
Audio

Audio Advanced Linux Sound Architecture (ALSA) arranges hardware audio devices and their components into a hierarchy of cards, devices, and subdevices. It reflects the capabilities of the hardware as seen by ALSA.

ALSA cards correspond one-to-one to hardware sound cards. A card can be denoted by its ID or by a numerical index starting at zero. ALSA hardware access occurs at the device level. The devices of each card are enumerated starting from zero.

The audio software stack is depicted in the following figure.

Figure 11: **Audio Software Stack**



At a high-level the audio software stack consists of the following layers from top to bottom:

- User-space layers
  - GStreamer: alsasrc and alsasink plugins
  - Alsa-lib: ALSA user-space library
- Kernel-space layers
  - ALSA: Xilinx ALSA ASoC driver

ALSA Source and Sink GStreamer Plugins

The *alsasrc* plugin reads audio data from an audio card and the *alsasink* plugin renders audio samples using the ALSA API. The audio device is specified by means of the *device* property referring to the ALSA device as defined in an *asound* configuration file.

Alsa-lib

The ALSA library API is the interface to the ALSA drivers. Developers need to use the functions in this API to achieve native ALSA support for their applications. The currently designed interfaces are as follows:

- Information Interface (/proc/asound)
- Control Interface (/dev/snd/controlCX)
- Mixer Interface (/dev/snd/mixerCXDX)
- PCM Interface (/dev/snd/pcmCXDX)
- Raw MIDI Interface (/dev/snd/midiCXDX)
- Sequencer Interface (/dev/snd/seq)
- Timer Interface (/dev/snd/timer)

For more information, refer to https://www.alsa-project.org/alsa-doc/alsa-lib/.

ALSA Kernel Subsystem

A sound card, encapsulating playback and capture devices will be visible as single entity to the end user. There can be many playback and capture devices within a sound card and there can be multiple sound cards in a system.

The Machine driver creates a pipeline out of the ALSA drivers. This glue or DAI (Digital Audio Interface) link is made using registered device names or device nodes (using OF kernel framework). Each proper DAI link results as a device in a sound card. A sound card is thus a logical grouping of several such devices.

The Audio Formatter driver creates the platform device for the sound card. While creating the device, it passes the HDMI device tree node of either I2S/HDMI/SDI/SPDIF depending on the kind of sound card being created. When the sound card driver detects the kind of audio node (I2S/HDMI/SDI/SPDIF), the proper DAI link is selected from the available links.

HDMI Rx receives the data from the HDMI source and separates audio from the video content. The Xilinx Audio Formatter converts this AES data to PCM data and stores it in memory. HDMI TX gets the AES data from the Audio Formatter and embeds it into video.

The AES format contains PCM and channel status information. The Audio Formatter IP separates non-audio content such as channel status and stores it in registers. The Audio Formatter driver can parse the content of channel status to get audio parameters.

A dummy CPU DAI driver is used as there needs to be a CPU DAI to be registered with ASoC framework. Codec DAI will be part of HDMI Tx and Rx video drivers, as those provide and consume the digital audio data.

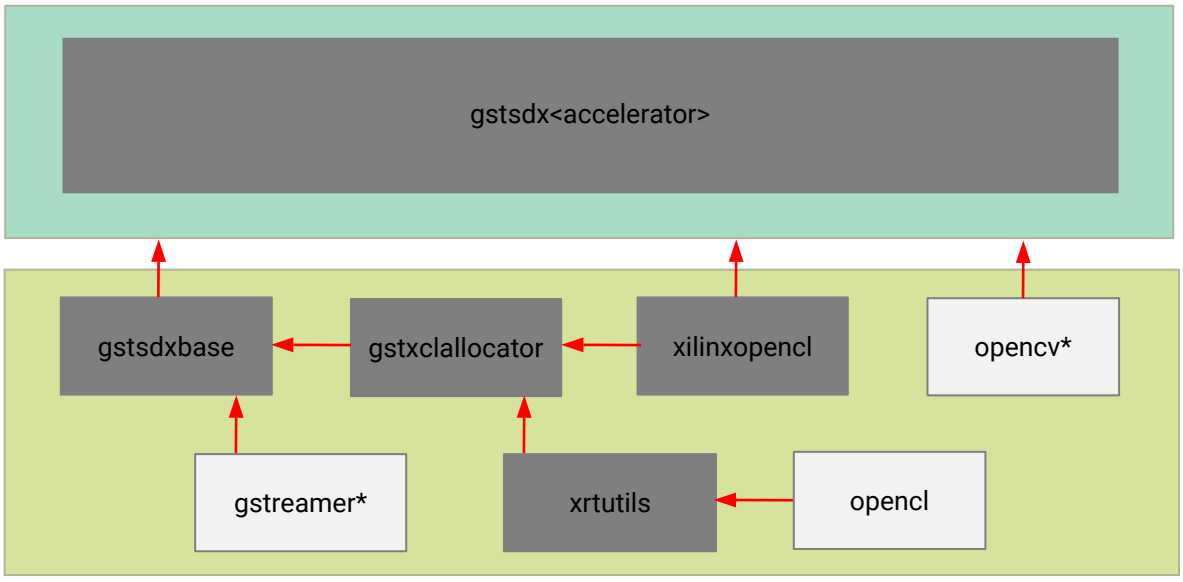
In this TRD design, a sound card is created with a record device for the HDMI-RX capture pipeline and a playback device for the HDMI-TX playback pipeline. The supported parameters are:

- Sampling rate: 48 kHz
- Sample width: 24 bits per sample
- Sample encoding: Little endian
- Number of channels: 2
- Supported format: S24_32LE

Accelerator

The accelerator GStreamer plugins are designed to implement memory-to-memory functions that can easily and seamlessly interface with other GStreamer elements such as video sources and sinks. The following figure shows the general architecture of accelerator plugins. The gray-colored boxes are components developed by Xilinx whereas the white boxes are open-source components.

Figure 12: **Gstreamer Plugin Architecture**



An accelerator element has one source and one sink pad; it can consume N temporal input frames from its source pad and produce one output frame on its sink pad. All accelerator plugins inherit from the generic base class which in turn inherits from the GStreamer video transform class. The base class provides common infrastructure that is shared across all accelerators. It also provides a generic filter-mode property which allows the user to switch between a hardware-accelerated version of the algorithm or a pure software implementation. Note that it is not mandatory for accelerator plugins to implement both modes. Accelerator plugins can implement additional accelerator-specific properties. The allocator class wraps the low-level memory allocation and dmabuf routines. The plugins launch the PL-based kernel or Data movers generated by the Vitis software platform.

The PL-based kernel uses the Xilinx Vitis Vision libraries. These libraries provide hardware-optimized implementations of a subset of the OpenCV libraries. They are implemented in C-code that is then synthesized to PL using high level synthesis (HLS).

- Xilinx Vitis Vision libraries:
  - https://github.com/Xilinx/Vitis_Libraries/tree/master/vision/
  - https://xilinx.github.io/Vitis_Libraries/vision/

The AIE-based kernel uses Xilinx AIE engine intrinsic calls. The AI engine program is implemented in C-code that is then synthesized to target AI engines using aiecompiler. A data mover implemented in C-code is synthesized to PL using high level synthesis (HLS) which transfers data to/from the AI engine.

The XRT and hls libraries are used for memory allocation as well as memory and hardware interface generation.

Filter 2D Plugin

In this example, a 2D convolution filter is implemented in three different versions:

- A software implementation using the OpenCV library
- A PL implementation using the Xilinx Vitis Vision library
  - https://xilinx.github.io/Vitis_Libraries/vision/
- An AIE implementation based on the *Versal ACAP AI Engine Programming Environment User Guide* (UG1076). The AIE implementation also needs a data mover in the PL that the plugin configures.

The kernel implements a transform function that takes an input frame and produces an output frame. It also exports an interface that allows the user to program the kernel coefficients (not available in the AIE implementation).

The PL based implementation uses three hardware-accelerated functions to process the video:

- The first function read_f2d_input, extracts the luma component from the input image to prepare it for the next, main processing step that operates on luma only.
- The main processing function filter2d_sd uses the Vitis Vision function filter2D with a 3x3 window size, and a maximum resolution of 3840x2160.
  - https://xilinx.github.io/Vitis_Libraries/vision/api-reference.html#vitis-vision-library-functions
  - https://github.com/Xilinx/Vitis_Libraries/blob/master/vision/L1/include/imgproc/xf_custom_convolution.hpp
- As final step, the write_f2d_output function merges the unmodified UV component with the modified luma component from the main processing function.

The AIE based implementation also uses three hardware-accelerated functions to process the video:

- The first function read_f2d_input, extracts the luma component from the input image to prepare it for the next, main processing step that operates on luma only. The luma component is streamed into an AI engine.
- The AI engine performs the main processing function with a 3x3 window size, and a fixed resolution of 720x1280 and stream outs the processed data to the data mover.
- As final step, the write_f2d_output function merges the unmodified UV component with the modified luma component from the main processing function.

GStreamer

GStreamer is a cross-platform open source multimedia framework that provides infrastructure to integrate multiple multimedia components and create pipelines/graphs. GStreamer graphs are made of two or more plugin elements which are delivered as shared libraries. The following is a list of commonly performed tasks in the GStreamer framework:

- Selection of a source GStreamer plugin
- Selection of a processing GStreamer plugin
- Selection of a sink GStreamer plugin
- Creation of a GStreamer graph based on above plugins plus capabilities
- Configuration of properties of above GStreamer plugins
- Control of a GStreamer pipeline/graph

Plugins

The following GStreamer plugin categories are used in this design:

- Source
  - *mediasrcbin*: V4l2 sources such as USB webcam, MIPI single-sensor, MIPI quad-sensor
  - *multisrc/filesrc*: video file source for raw or encoded image/video files
- Sink
  - *kmssink*: KMS display sink for HDMI Tx
  - *filesink*: video file sink for raw or encoded image/video files
  - *appsink*: sink that makes video buffers available to an application such as the display inside jupyter notebooks
- Encode/decode
  - *jpegenc/dec*: jpg image file encode/decode
  - *vp9enc/dec*: vp9 video file encode/decode
- Processing/acceleration
  - sdxfilter2d: 2D filter accelerator
- Other
  - *capsfilter*: filters capabilities
  - *tee*: tee element to create a fork in the data flow
  - *queue*: creates separate threads between pipeline elements and adds additional buffering
  - *perf*: measure frames-per-seconds (fps) at an arbitrary point in the pipeline

Capabilities

The pads are the element's interface to the outside world. Data streams from one element's source pad to another element's sink pad. The specific type of media that the element can handle is exposed by the pad's capabilities. The following capabilities are used between the video-source plugin and its peer plugin (either video-sink or video-processing). These capabilities (also called *capsfilter*) are specified while constructing a GStreamer graph, for example:

```
"video/x-raw, width=<width of videosrc>, height=<height of videosrc>,
format=YUY2, ramerate=<fps/1>"
```

If *multisrc* is used as video-source plugin, the *videoparse* element is used instead of a *capsfilter* to parse the raw video file and transform it to frames:

```
"video/x-raw, width=<width of videosrc>, height=<height of videosrc>,
format=YUY2, framerate=<fps/1>"
```

Pipeline Control

The GStreamer framework is used to control the GStreamer graph. It provides the following functionality:

- Start/stop video stream inside a graph
- Get/set controls
- Buffer operations
- Get frames-per-second information

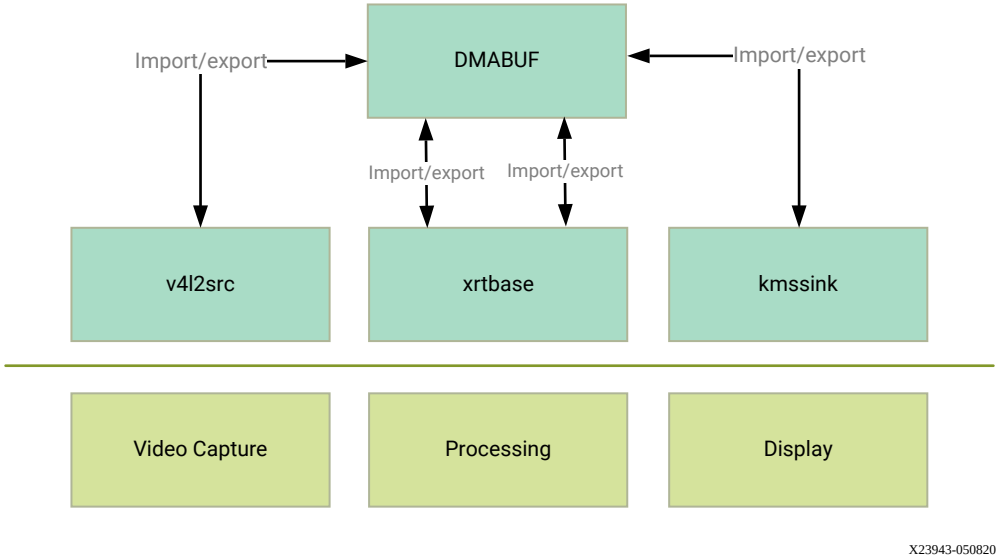
There are four states defined in the GStreamer graph: "NULL", "READY", "PAUSED", and "PLAYING". The "PLAYING" state of a GStreamer graph is used to start the pipeline and the "NULL" state is to stop the pipeline.

Allocators

GStreamer abstracts buffer allocation and pooling. Custom allocators and buffer pools can be implemented to accommodate custom use-cases and constraints. The video source controls buffer allocation, but the sink can propose parameters in the negotiation phase.

The DMABUF framework is used to import and export buffers in a 0-copy fashion between pipeline elements, which is required for high-performance pipelines, as shown in the following figure. The *v4l2src* element has a property named *io-mode* which allows allocation and export of DMABUFs to its peer element. The *kmssink* element allows import as well as export of DMABUFs to/from its peer element. The accelerator element *xrtbase* allows only import of DMABUFs, which means it relies on DMABUFs being allocated by its peer elements connected to the source and sink pads.

Figure 13: **DMABUF Sharing Mechanism**



Note that DMABUFs are not necessarily physically contiguous depending on the underlying kernel device driver, that is, the UVC v4l2 driver does not allocate CMA memory which results in a data copy if its peer element can only handle contiguous memory.

Jupyter Notebooks

The reference design provides several notebooks to exercise and evaluate the reference design. The notebooks follow this general sequence:

- Create Gstreamer elements
  - Example: *mediasrcbin* for V4L devices, *kmssink* for HDMI display, *perf* for performance monitoring.
- Configure the Gstreamer elements
  - Example: set source type, video resolution
- Create a Gstreamer pipeline by adding and linking the elements
- Run the pipeline by setting the Gstreamer state to PLAYING. Then click on the stop button which will put the Gstreamer state to NULL and stop the pipeline.

Additionally, the notebooks:

- Display a Gstreamer pipeline graph
- Plot the live memory bandwidth by running the APM notebook
- Plot CPU utilization and power utilization in a real-time graph

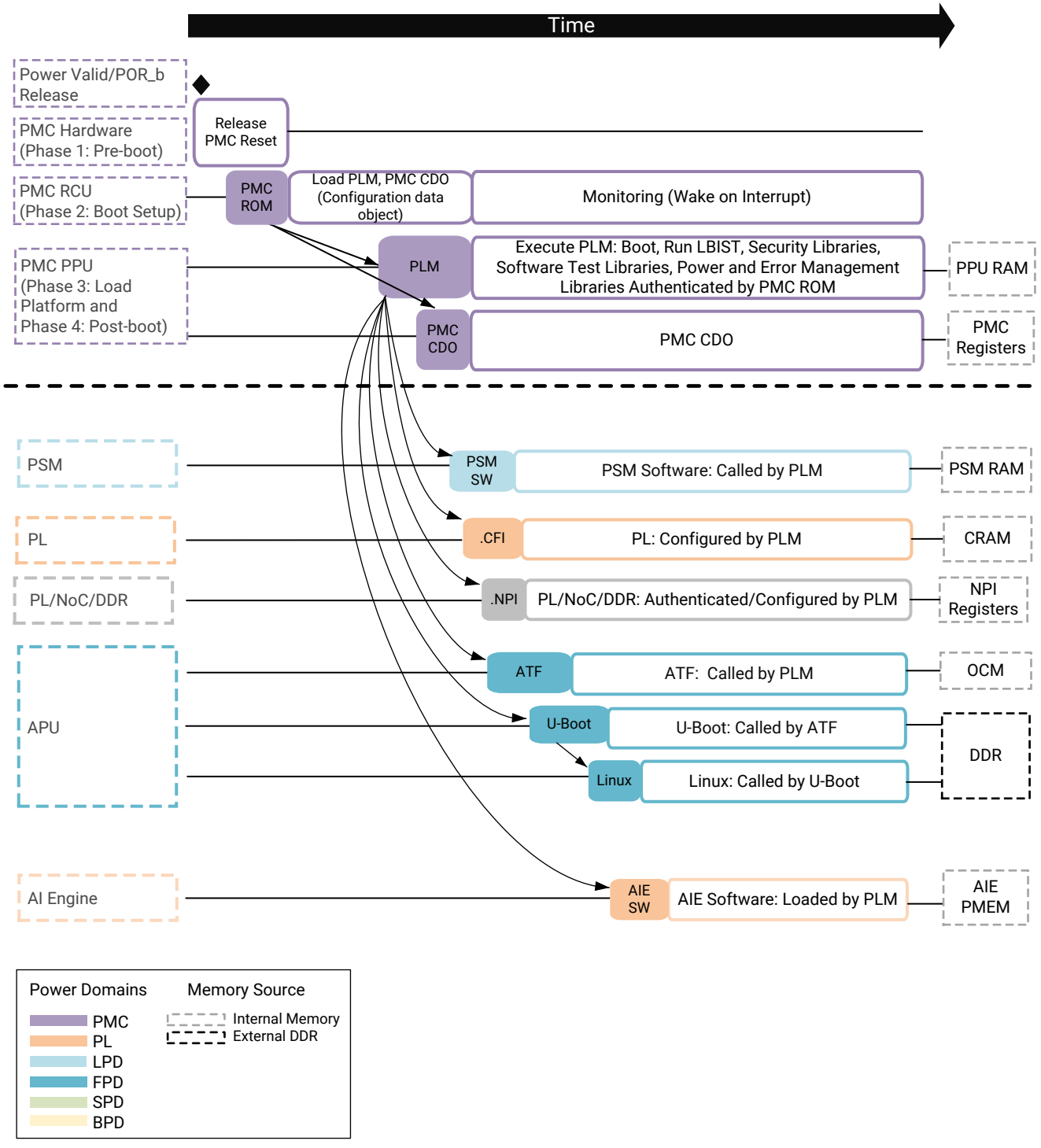
System Consideration

This chapter describes an example boot sequences in which you can bring up various components and execute the required boot tasks.

Boot Process

The following figure depicts the primary responsibilities of the Platform Management Controller (PMC) unit, along with the memory source at each phase of the non-secure boot flow. The figure also shows how the platform loader and manager (PLM) loads the major partition components of the ACAP software stack (exceptfor Linux). U-Boot loads the Linux OS.

Figure 14: **Boot Flow Sequence**



The boot process is divided into four phases that are independent of the selected boot mode.

- Phase 1: Pre-boot (power-up and reset)
  - The pre-boot phase is initiated when the PMC senses the PMC power domains (VCCAUX_PMC and VCC_PMC) and when the external POR_B (power on reset) pin is released.
  - PMC reads the boot mode pins and stores the value in the boot mode register.
  - PMC sends the reset signal to the ROM Code Unit (RCU).
- Phase 2: Boot setup (initialization and boot header processing)
  - The RCU begins to execute the BootROM executable from the RCU ROM.
  - The BootROM executable reads the boot mode register to select the boot device.
  - The BootROM executable reads the boot header in the PDI from the boot device and validates it.
  - The BootROM executable finds the PLM in the Programmable Device Image (PDI).
  - The BootROM executable loads the PLM from the PDI into Platform Processing Unit (PPU) RAM and validates it.
  - The BootROM executable releases the reset to the PPU to execute the PLM.
  - The BootROM executable enters a sleep state. The BootROM executable continues to run until the next power-on-reset (POR) or system reset, and is responsible for post-boot platform tasks.
- Phase 3: Load platform (boot image processing and configuration)
  - The PPU begins to execute the PLM from the PPU RAM.
  - The PLM reads and processes the PDI, validating PDI components.
  - The PLM loads the applications and data for the Arm Cortex-A72 and Cortex-R5F processors to various memories specified by the ELF file. These memories include onboard DDR and internal memories, such as on-chip memory and TCMs.
  - The PLM sends configuration information to various Versal ACAP components
    - NoC initialization
    - DDR initialization
    - PS
    - PL
    - AI engines

- Phase 4: Post-boot (Platform management and monitoring services)
  - The BootROM executable continues to run until the next power-on reset (POR) or system reset, and is responsible for its post-boot platform management tasks. The BootROM executable sleeps, and wakes up for security tampering event interrupts and for service routines.
  - The PLM continues to run until the next POR or system reset, and is responsible for its post-boot platform management tasks.

For detailed information on the boot sequence see the *Versal ACAP System and Software Developers Guide* (UG1304).

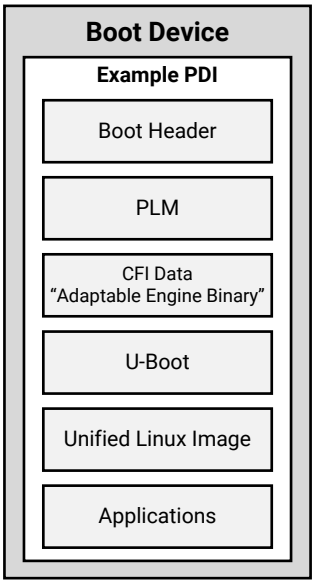
Programmable Device Image (PDI)

The PDI is a Xilinx format file which is processed by the PMC as part of the Versal ACAP boot process. The full PDI contains the following information needed to boot, configure, and manage the Versal ACAP.

- Boot header
- PLM
- Meta header that contains an image header table, image tables, and partition tables.
- Additional subsystem images and image partitions used to configure the Versal ACAP.

The folowing figure shows an example PDI.

Figure 15: **Programmable Device Image**



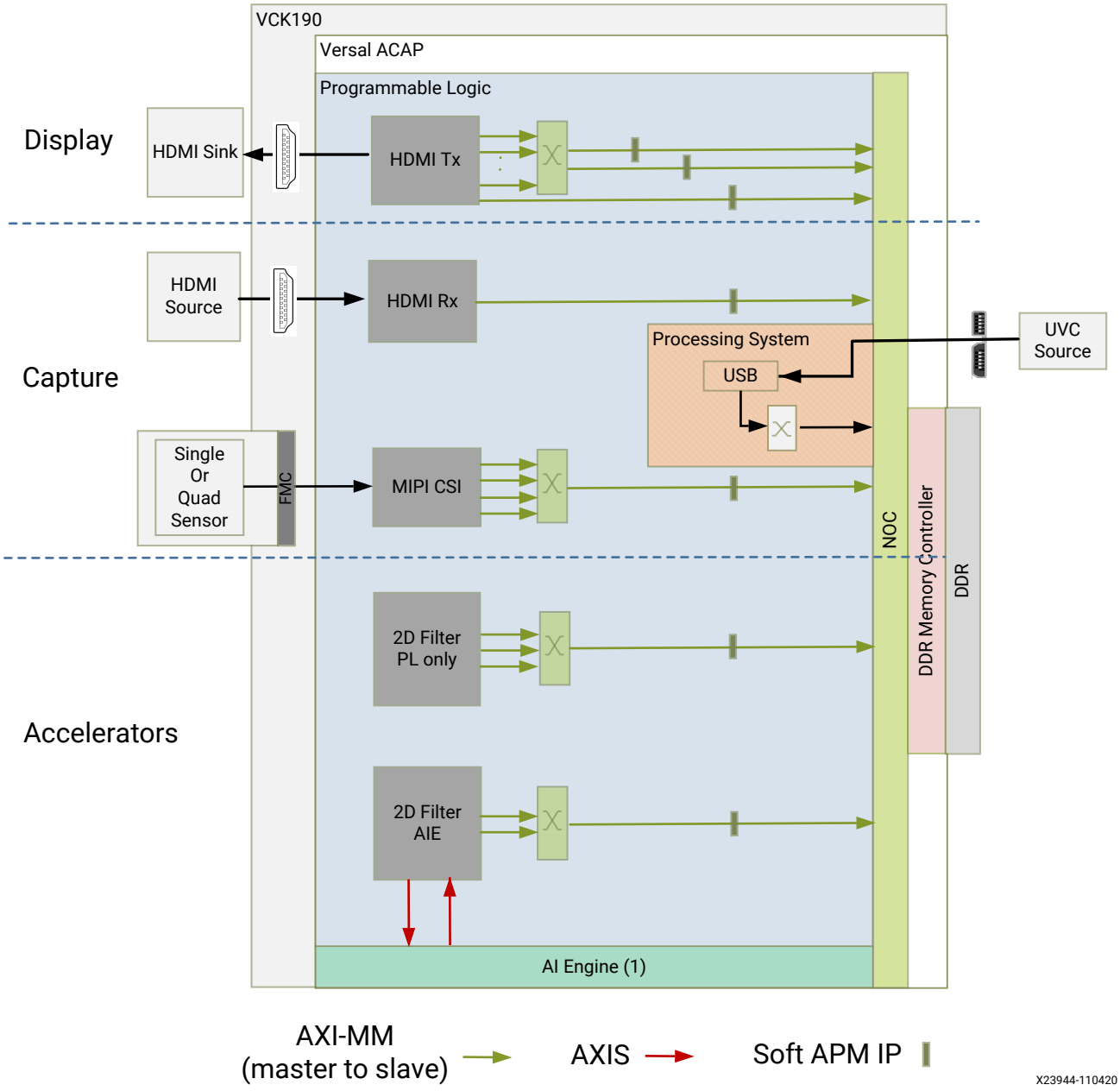
Chapter 5

Hardware Architecture

Introduction

This chapter describes the targeted reference design (TRD) hardware architecture. The following figure shows a block diagram of the design components inside the Versal ACAP on the VCK190 board. See *VCK190 Evaluation Board User Guide* (UG1366) for more information.

Figure 16: **Hardware Block Diagram**



At a high level, the design comprises three pipelines:

- Capture/input pipeline
  - USB capture pipeline (PS)
  - Single or quad MIPI CSI-2 Rx capture pipeline (FMC + PL)
  - HDMI RX video and audio capture pipeline

- Processing Pipeline
  - 2D filter processing pipeline
- Display/Output Pipeline
  - HDMI TX display pipeline
  - HDMI RX audio pipeline

The block diagram comprises two parts: platforms and accelerators.

- Platforms
  - This mainly consist of I/O interfaces and their data motion network. This is the fixed part of the design. Platforms supported in this reference design:
    - Platform 1: Single sensor MIPI CSI-2 Rx (capture), USB-UVC (capture), HDMI Tx (display)
    - Platform 2: Quad sensor MIPI CSI-2 Rx (capture), USB-UVC (capture), HDMI Tx (display)
    - Platform 3: HDMI Rx (capture), USB-UVC (capture), HDMI Tx (display)
- Accelerators
  - This is a block which performs different video processing functions. This is the variable part of the design. Hardware accelerators supported in this reference design:
    - 2D convolution filter in the PL
    - 2D convolution filter in the AIE

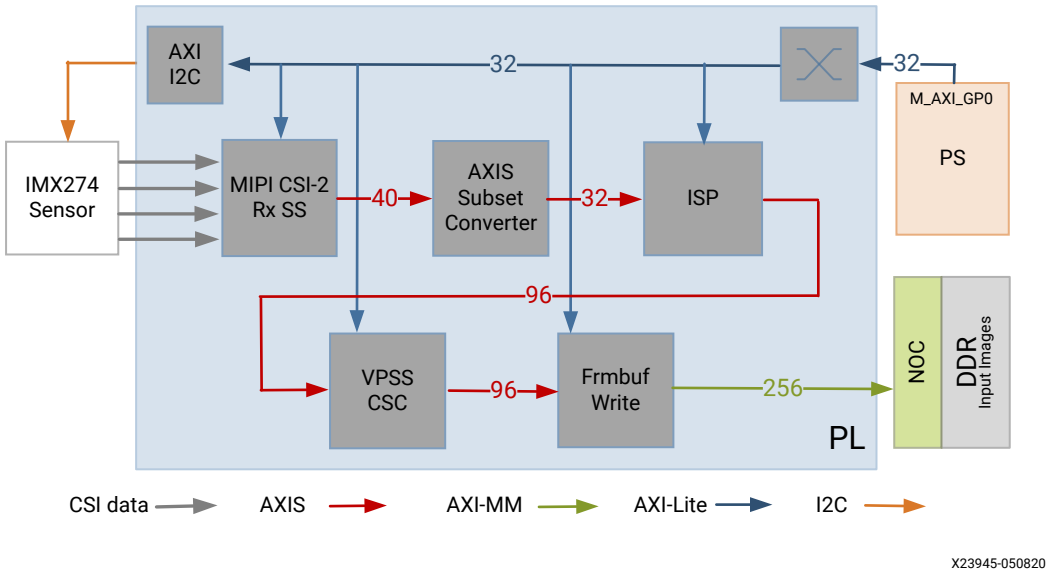
The accelerator and corresponding data/control interfaces (AXI-MM, AXI-Lite, interrupts) are generated by the Vitis tool and is integrated into the platform.

Capture Pipeline

Single Sensor MIPI Capture

A capture pipeline receives frames from an external source and writes it into memory. The single sensor MIPI CSI-2 receiver capture pipeline is shown in the following figure.

Figure 17: **MIPI CSI Video Capture Pipeline**



This pipeline consists of six components, of which four are controlled by the APU via an AXI-Lite based register interface; one is controlled by the APU via an I2C register interface, and one is configured statically.

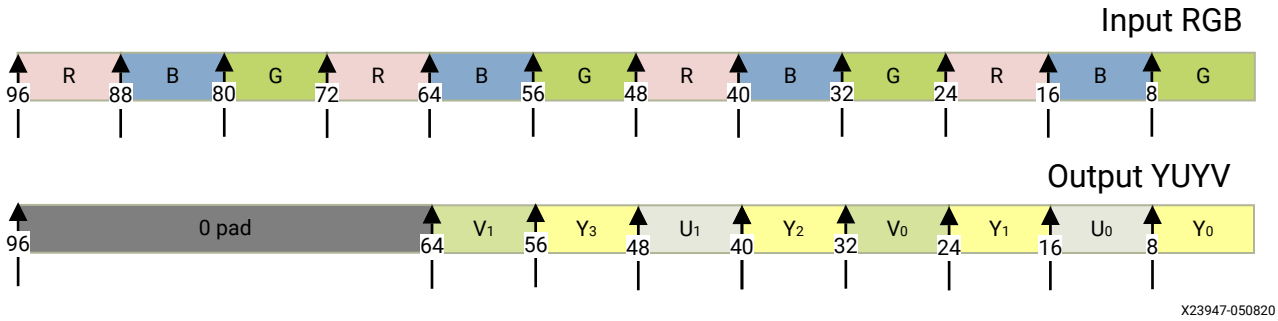
- The Sony IMX274 is a 1/2.5 inch CMOS digital image sensor with an active imaging pixel array of 3864H x2196V. The image sensor is controlled via an I2C interface using an AXI I2C controller in the PL. It is mounted on a FMC daughter card and has a MIPI output interface that is connected to the MIPI CSI-2 RX subsystem inside the PL. For more information refer to the LI-IMX274MIPI-FMC_datasheet.
- The MIPI CSI-2 receiver subsystem (CSI Rx) includes a MIPI D-PHY core that connects four data lanes and one clock lane to the sensor on the FMC card. It implements a CSI-2 receive interface according to the MIPI CSI-2 standard v2.0 with underlying MIPI D-PHY standard v1.2. The subsystem captures images from the IMX274 sensor in RAW10 format and outputs AXI4-Stream video data. For more information see the *MIPI CSI-2 Receiver Subsystem Product Guide* (PG232).
- The AXI subset converter, see *AXI4-Stream Infrastructure IP Suite LogiCORE IP Product Guide* (PG085), is a statically-configured IP core that converts the raw 10-bit (RAW10) AXI4-Stream input data to raw 8-bit (RAW8) AXI4-Stream output data by truncating the two least significant bits (LSB) of each data word. At four pixels per clock (4ppc), the AXIS width is 32 bits.
- The Image Single Processing IP available in the Vitis vision librarires (https://github.com/Xilinx/Vitis_Libraries/tree/master/vision/L1) implements the following functions.
  - The Badpixelcorrection module removes the defective pixels in the image as an image sensor may have a certain number of defective/bad pixels that may be the result of manufacturing faults or variations in pixel voltage levels based on temperature or exposure.

- The Gain control module improves the overall brightness of the input image by applying a multiplicative gain (weight) for red and blue channel to the input bayerized image.
- The Demosaicing module converts a single plane Bayer pattern output, from the digital camera sensors to a color image.
- The histogram module computes the histogram of given input image. The normalization module changes the range of pixel intensity values. Both modules are used to improve the contrast in the image.

  See https://xilinx.github.io/Vitis_Libraries/vision/api-reference.html#vitis-vision-library-functions for more details
- The ISP IP receives the RAW AXI4-Stream input data and interpolates the missing color components for every pixel to generate a 24-bit, 8 bits per pixel (8 bpc) RGB output image transported via AXI4-Stream. At 4 ppc, the AXIS width is 96-bit. A GPIO from the PS is used to reset the IP between resolution changes.

- The video processing subsystem (VPSS), see *Video Processing Subsystem Product Guide* (PG231), is a collection of video processing IP subcores. This instance uses the scaler only configuration which provides scaling, color space conversion, and chroma resampling functionality. The VPSS takes AXI4-Stream input data in 24-bit RGB format and converts it to a 16-bit, 8bpc YUV 4:2:2 output format. The following figure shows AXIS data interface at 4ppc. A GPIO pin from the PS is used to reset the subsystem between resolution changes.

Figure 18: **AXI-Stream Data Bus Encoding**



- The video frame buffer, see *Video Frame Buffer Read and Video Frame Buffer Write LogiCORE IP Product Guide* (PG278) takes YUV 4:2:2 sub-sampled AXI4-Stream input data and converts it to AXI4-MM format which is written to memory as 16-bit packed YUYV. The AXI-MM interface is connected to the system DDR via NOC. For each video frame transfer, an interrupt is generated. A GPIO is used to reset the IP between resolution changes.

All the IPs in this pipeline are configured to transport 4ppc @ 150 MHz, enabling up to 3840x2160 resolution at 60 frames per second (fps).

- Time to transfer one frame: (3840 + 560) x (2160 + 90) / (150 MHz * 4ppc) = 0.0165 ms
- Number of frames transferred per second = 1/0.0165 = 60 frames

Note: In this calculation the vertical blanking accounts for 90 pixels per line and the horizontal blanking for 560 lines per video frame.

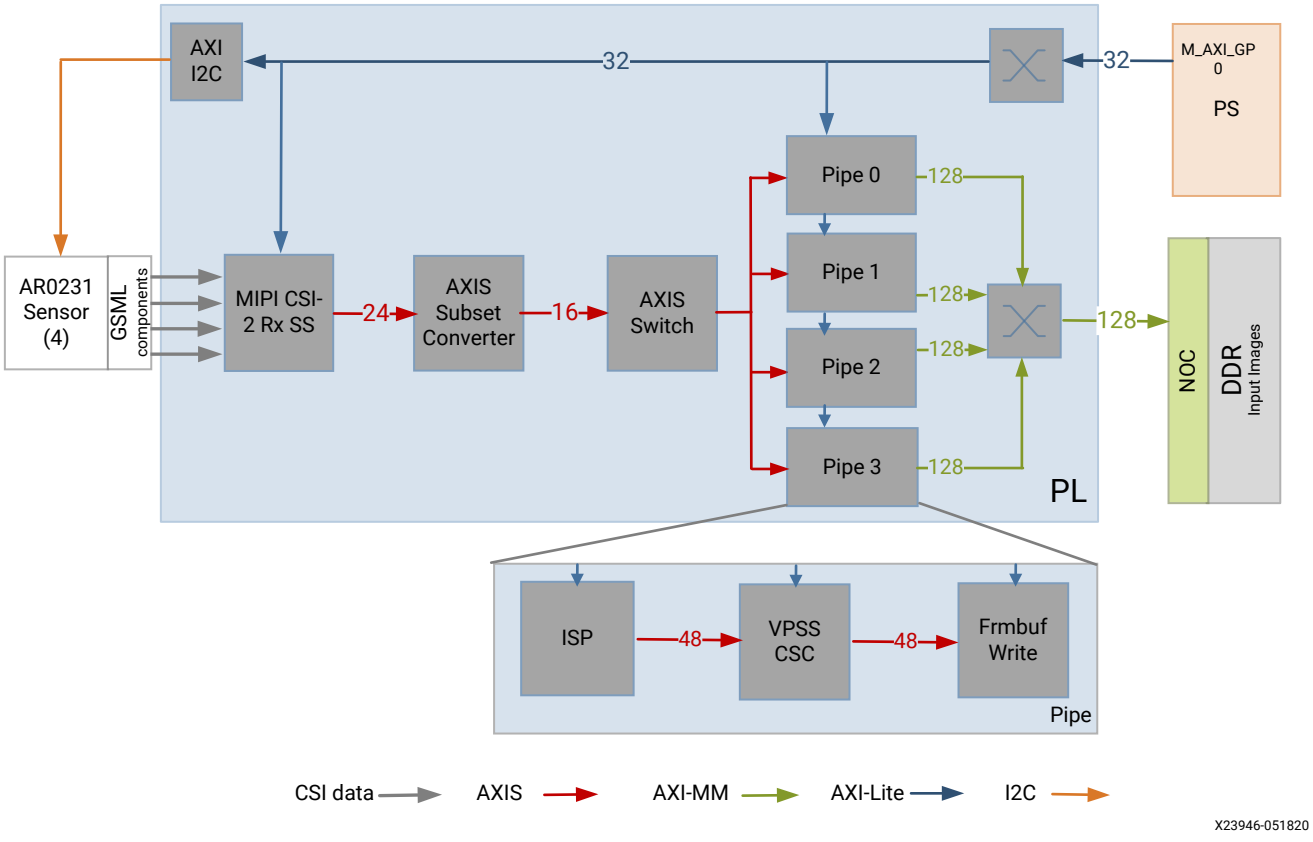
The video resolution, frame format and frame rate are set via register writes through the AXI-Lite interface of the IPs at run-time. The drivers for the above blocks provide APIs to set these values in a user application.

- For the pass-through design (no accelerator) user can choose between 720p60, 1080p60, 2160p30, and 2160p60.
- For the 2D filter PL accelerator user can choose between 720p60, 1080p60, 2160p30 and 2160p60.
- For the 2D filter AIE accelerator resolution is fixed at 720p60.

Quad Sensor MIPI Capture

The quad sensor MIPI CSI-2 receiver capture pipeline is shown in the following figure.

Figure 19: **Quad MIPI CSI Video Capture Pipeline**



- The Avnet Multicamera FMC module bundles fours ON Semi image sensors (AR0231) with GMSL (Gigabit Multimedia Serial Link) serializers (MAX96705) and deserialzer (MAX9286).

- The MIPI CSI-2 subsystem, see the *MIPI CSI-2 Receiver Subsystem Product Guide* (PG232), captures images from the deserializer in RAW12 format on four lanes and outputs AXI4-Stream video data.
- The AXI subset converter converts the raw 12-bit (RAW12) AXI4-Stream input data to raw 8-bit (RAW8) AXI4-Stream output data by truncating the four least significant bits (LSB) of each data word. The AXIS switch splits the incoming data into four streams using the destination id.
- The ISP IP receives the RAW AXI4-Stream input data and interpolates the missing color components for every pixel to generate a 24-bit, 8 bits per pixel (8 bpc) RGB output image transported via AXI4-Stream.
- The VPSS takes AXI4-Stream input data in 24-bit RGB format and converts it to a 16-bit, 8 bpc YUV 4:2:2 output format.
- The video frame buffer takes YUV 4:2:2 sub-sampled AXI4-Stream input data and converts it to AXI4-MM format which is written to memory as 16-bit packed YUYV.

All of the IPs in this pipeline are configured to transport 2 ppc @ 150 MHz, enabling up to 1920x1080 resolution at 120 fps, or 30 fps per stream.

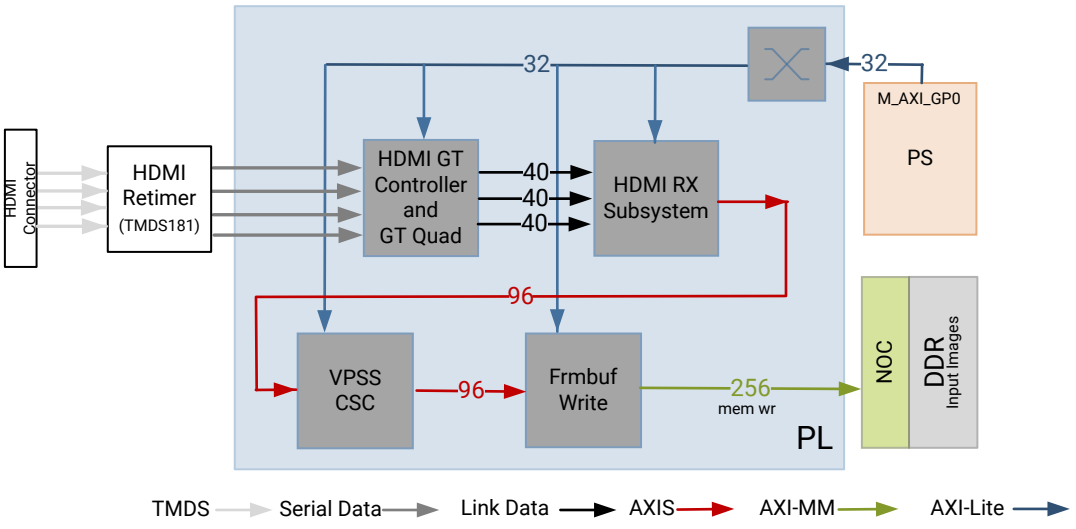
- Time to transfer one frame: (1920 + 280) x (1080 + 45) / (150 MHz * 2 ppc) = 0.00825 ms
- Number of frames transferred per second = 1/0.00825 = 120 frames

Note: The AR0231 sensor is limited to 1080p30 applications.

HDMI Rx Capture

The HDMI receiver capture pipeline is shown in the following figure.

Figure 20: **HDMI RX Capture Pipeline Block Diagram**



This pipeline consists of four main components, each of them controlled by the APU via an AXI4-Lite base register interface:

- The HDMI retimer converts TMDS data from the HDMI connector to serial data and clock, and provides them to the GT QUAD.
- The HDMI GT controller and PHY (GT QUAD) enable plug-and-play connectivity with the video transmit or receive subsystems. The interface between the media access controller (MAC) and physical (PHY) layers are standardized to enable ease of use in accessing shared gigabit-transceiver (GT) resources. The data recovery unit (DRU) supports lower line rates for the HDMI protocol. An AXI4-Lite register interface is provided to enable dynamic accesses of transceiver controls/status. See the *HDMI GT Controller LogiCORE IP Product Guide* (PG334) for more information. The HDMI GT controller and PHY are shared with the HDMI TX display pipeline
- The HDMI receiver subsystem (HDMI RX) interfaces with PHY layers and provides HDMI decoding functionality. The subsystem is an hierarchical IP that bundles a collection of HDMI RX-related IP subcores and outputs them as a single IP. The subsystem receives the captured TMDS data from the PHY layer. It then extracts the video stream from the HDMI stream and generates a 96-bit AXI4-Stream data stream corresponding to four pixels per clock. The data format is dependent on the HDMI source format. See the *HDMI 1.4/2.0 Receiver Subsystem Product Guide* (PG236) for more information.

- The video processing subsystem (VPSS) is a collection of video processing IP subcores. This instance of the VPSS uses the video scaler only configuration which provides scaling, color space conversion, and chroma resampling functionality. The VPSS takes AXI4-Stream input data from the HDMI RX subsystem and depending on the input format and resolution, converts and scales it to YUV 4:2:2 format transferred on a 96-bit AXI4-Stream interface. A GPIO is used to reset the subsystem between resolution changes. See the *Video Processing Subsystem Product Guide* (PG231) for more information.(
- The video frame buffer takes YUV 4:2:2 sub-sampled AXI4-Stream input data and converts it to AXI4-MM format which is written to memory as 16-bit packed YUYV. The AXI-MM interface is connected to the system DDR via the NOC. An interrupt is generated for each video frame transfer. A GPIO is used to reset the IP between resolution changes. See the *Video Frame Buffer Read and Video Frame Buffer Write LogiCORE IP Product Guide* (PG278) for more information.

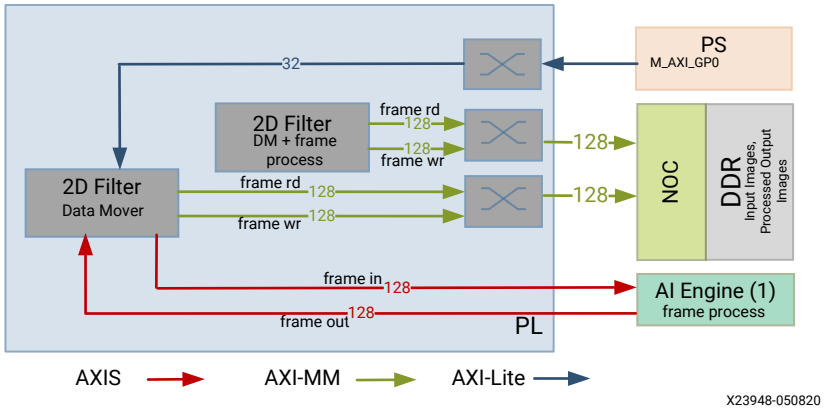
All of the IPs in this pipeline are configured to transport 4ppc @ 150 MHz, enabling up to 3840x2160 resolution at 60 frames per second (fps).

- Time to transfer one frame: (3840 + 560) x (2160 + 90) / (150 MHz * 4ppc) = 0.0165 ms
- Number of frames transferred per second = 1/0.0165 = 60 frames

Processing Pipeline

A memory-to-memory (M2M) pipeline reads video frames from memory, does certain processing, and then writes the processed frames back into memory A block diagram of the process pipeline is shown in the following figure.

Figure 21: **M2M Processing Pipeline Showing Hardware Accelerator and Data Motion Network**



There are two accelerators supported in this reference design:

- 2D convolution filter implemented in PL along with a data mover (DM)

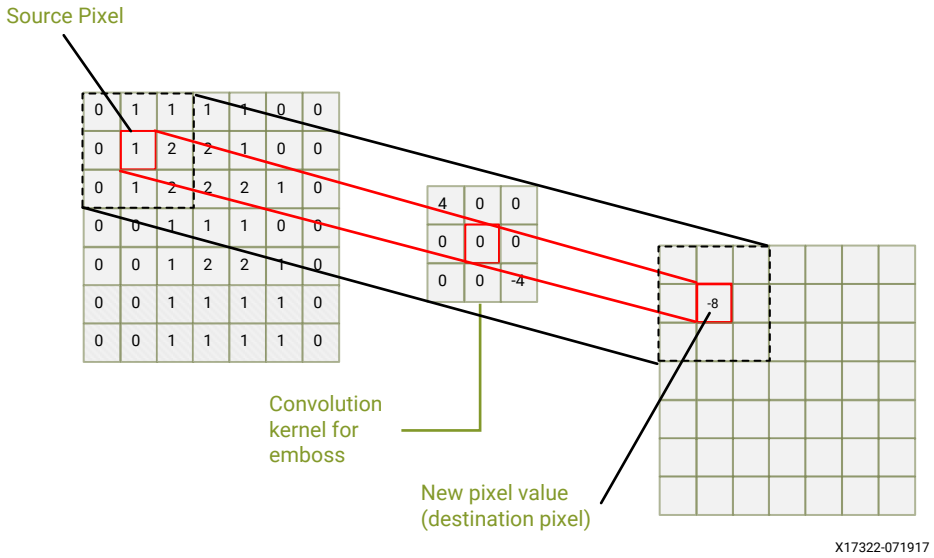
- 2D convolution filter implemented in AIE along with a data mover DM in PL

The memory-to-memory (m2m) processing pipeline with the 2D convolution filter is generated and integrated by the Vitis™ tool. The C-based 2D filter function is translated to RTL and then packaged as kernel object (.xo) using Vitis™ HLS. The case is the same for the data mover required for the 2D Convolution filter in AIE. The Cardano compiler generates the connectivity graph (.o) with the AIE engine and the program (2D convolution filter elf) to execute on AIE. The Vitis™ tool uses the .xo and .o outputs from these tools and integrates the IPs into the platform.

The data movers read input frames from the memory. The processing block runs convolution on the frame. Convolution is a common image processing technique that changes the intensity of a pixel to reflect the intensities of the surrounding pixels. This is widely used in image filters to achieve popular image effects like blur, sharpen, and edge detection.

The implemented algorithm uses a 3x3 kernel with programmable filter coefficients. The coefficients inside the kernel determine how to transform the pixels from the original image into the pixels of the processed image, as shown in the following figure.

Figure 22: **2D Convolution Filter with a 3x3 Kernel**



The algorithm performs a two-dimensional (2D) convolution for each pixel of the input image with a 3x3 kernel. Convolution is the sum of products, one for each coefficient/source pixel pair. As the reference design is using a 3x3 kernel, in this case it is the sum of nine products.

The result of this operation is the new intensity value of the center pixel in the output image. This scheme is repeated for every pixel of the image in raster-scan order, that is, line-by-line from top-left to bottom-right. In total, width x height 2D convolution operations are performed to process the entire image.

The pixel format used in this design is YUYV which is a packed format with 16 bits per pixel. Each pixel can be divided into two 8-bit components: one for luma (Y), the other for chroma (U/V alternating).

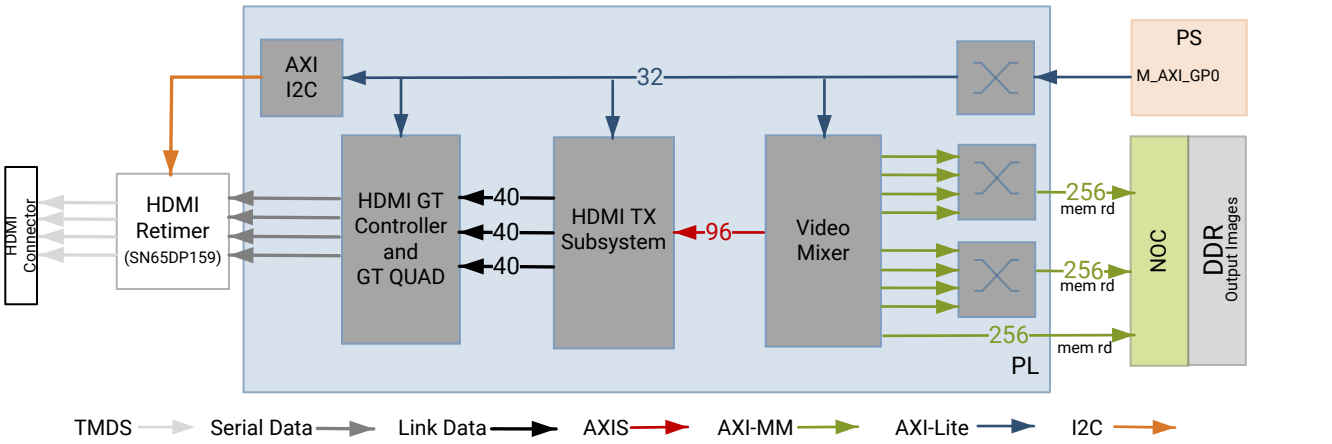
In this implementation, only the Y component is processed by the 2D convolution filter which is essentially a grayscale image. The reason is that the human eye is more sensitive to intensity than color. The combined U/Y components which accounts for the color is merged back into the final output image unmodified. The processed frame is then written back to memory.

***Note*:** The 2D filter in the PL has the option of reading coefficients from memory (AXI MM is not shown in the figure). The 2d filter in the AIE only supports fixed coefficients corresponding to a Sobel filter.

Display Pipeline

An output pipeline reads video frames from memory and sends the frames to a sink. In this case the sink is a display and therefore this pipeline is also referred to as a display pipeline. The HDMI display pipeline is shown in the following figure.

Figure 23: **HDMI Transmitter Display Pipeline**



X24753-102020

This pipeline consists of three main components, all of them controlled by the APU via an AXI-Lite base register interface:

- The video mixer IP core is configured to support blending of up to eight overlay layers into one single output video stream. The eight layers are configured to be memory-mapped AXI4 interfaces connected to the NOC via two interconnects. Two interconnects are required to reduce arbitration across ports. The main AXI-MM layer has the resolution set to match the display. The other layers, whatever their resolution, is blended with this layer. Four video

layers are configured for YUYV and the other four are configured for RGB. The AXI4-Stream output interface is a 96-bit bus that transports 4ppc for up to 2160p60 performance. It is connected to the HDMI Tx subsystem input interface. A GPIO is used to reset the subsystem between resolution changes. For more information refer to the input interface *Video Mixer LogiCORE IP Product Guide* (PG243).

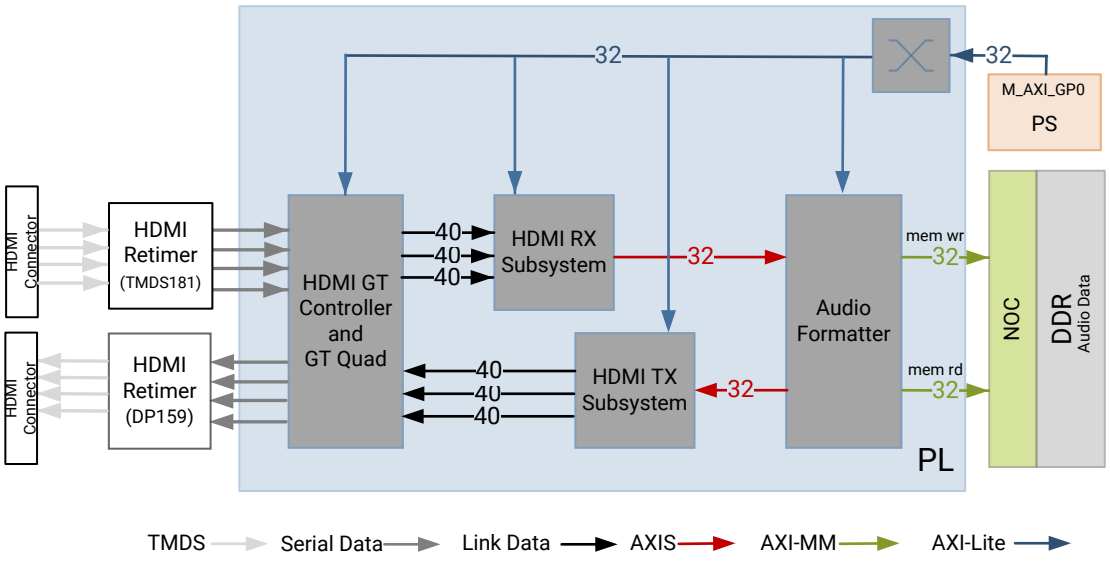
***Note*:** The mixer configuration remains the same for different capture sources. To enable/disable various layers, software programs the layer enable register in the IP

- The HDMI transmitter subsystem (HDMI Tx) interfaces with PHY layers and provides HDMI encoding functionality. The subsystem is a hierarchical IP that bundles a collection of HDMI TX-related IP sub-cores and outputs them as a single IP. The subsystem generates an HDMI stream from the incoming AXI4-Stream video data and sends the generated link data to the video PHY layer. For more information refer to the *HDMI 1.4/2.0 Transmitter Subsystem Product Guide* (PG235).
- The HDMI GT controller and PHY (GT) enables plug-and-play connectivity with the video transmit or receive subsystems. The interface between the media access control (MAC) and physical (PHY) layers are standardized to enable ease of use in accessing shared gigabit-transceiver (GT) resources. The data recovery unit (DRU) is used to support lower line rates for the HDMI protocol. An AXI4-Lite register interface is provided to enable dynamic accesses of transceiver controls/status. For more information refer to the *HDMI GT Controller LogiCORE IP Product Guide* (PG334).
- The HDMI re-timer converts serial HDMI output signals to transition minimized differential signals (TMDS) compliant with HDMI signaling.. For more information refer to SNx5DP159 datasheet.

HDMI Audio Pipeline

In Platform3, where video capture and display are enabled via HDMI it also possible to capture and replay audio. The HDMI audio RX-to-TX pipeline is shown in the following figure. This pipeline consists of four components, each of them controlled by the APU through an AXI4-Lite base register interface.

Figure 24: **The HDMI Audio Pipeline**



X24754-102020

- The HDMI GT controller is shared with the HDMI RX and HDMI TX pipelines.
- The HDMI RX subsystem converts the captured audio to a multiple channel AXI audio stream and outputs the audio data on 32-bit AXI Stream interface. This design supports two audio channels. The subsystem also outputs Audio Clock Regeneration (ACR) signals that allow regeneration of the audio clock. The ACR signals are passed to hdmi_acr_ctrl which calculates Cycle Time Stamp (CTS) values for the transmit. It basically counts the cycles of the TX TMDS clock for a given audio clock. See the *HDMI 1.4/2.0 Receiver Subsystem Product Guide* (PG236) for more information.
- • The audio formatter provides high-bandwidth direct memory access between memory and AXI4-Stream target peripherals. Initialization, status, and management registers are accessed through an AXI4-Lite slave interface. It is configured with both read and write interface enabled for a maximum of two audio channels and interleaved memory packing mode with memory data format configured as AES to PCM. The IP receives audio input from the HDMI RX subsystem IP and writes the data to memory. It reads audio data from memory and sends it out to the HDMI TX subsystem IP, which forwards it to the output device. See the *Audio Formatter Product Guide* (PG330) for more information.
- The HDMI TX subsystem receives the 32-bit AXI stream audio data from the audio formatter and transfers it to the HDMI GT controller as Link Data. This is further transferred as TMDS data on the HDMI and finally to a HDMI replay device. This block also receives ACR signals used to transmit an audio packet. See the *HDMI 1.4/2.0 Transmitter Subsystem Product Guide* (PG235) for more information.

Clocks, Resets, and Interrupts

The following table lists the clock frequencies of key ACAP components and memory. For more information refer to the *Versal ACAP Technical Reference Manual* (AM011).

Table 2: **Key Component Clock Frequencies**

Component	Clock Frequency
ACPU	1,000 MHz
NOC	950 MHz
NPI	300 MHz
LPDDR	1,600
AIE	1,000

The following table identifies the main clocks of the PL design, their source, their clock frequency, and their function.

Table 3: **System Clocks**

Clock	Clock Source	Clock Frequency	Function
pl0_ref_clk	CIPS	100 MHz	Clock source for clocking wizard.
clk_out1	Clocking wizard	150 MHz	AXI MM clock and AXI Stream clock used in the capture of platform2, display pipeline, and processing pipeline.
clk_out2	Clocking wizard	105 MHz	AXI-Lite clock to configure the different IPs in the design.
clk_out3	Clocking wizard	200 MHz	MIPI D-PHY core clock. Also the AXI MM clock and AXI Stream clock used in the capture pipeline of plaform2.
sys_clk0	SI570 (External)	200 MHz	Differential clock source used internally by the memory controller to generate various clocks to access DDR memory.
HDMI DRU clock	SI570 (External)	200 MHz	Clock for data recovery unit for low line rates.
HDMI GT TX reference clock	IDT 8T49N241 (External)	Variable	GT Transmit clock source to support various HDMI resolutions.
HDMI GT RX reference clock	Si570 (External);	Variable	GT receive clock to support various HDMI resolutions.
Audio clock	Si570 (External)	Variable	Master reference clock to generate audio stream at the required sampling rate.

The PL0 clock is provided by the PPLL inside the PMC domain and is used as the reference input clock for the clocking wizard instance. This clock does not drive any loads directly. A clocking wizard instance is used to de-skew the clock and to provide three phase-aligned output clocks, clk_out1, clk_out2 and clk_out3 .

The clk_out2 is used to drive most of the AXI-Lite control interfaces of the IPs in the PL. AXI-Lite interfaces are typically used to configure registers and therefore can operate at a lower frequency than data path interfaces. Exception is the AXI-Lite interfaces of HLS based IP cores where the control and data plane use either clk_out1 or clk_out3.

The clk_out1 clock drives the AXI MM interfaces and AXI Stream interfaces of the display pipeline and processing pipeline. It also drives AXI MM interfaces and AXI Stream interfaces of the capture pipeline of platform2. The clk_out3 clock drives the AXI MM interfaces and AXI Stream interfaces of the capture pipeline in platform1.

For details on HDMI Tx and HDMI GT clocking structure and requirements refer to*HDMI 1.4/2.0 Transmitter Subsystem Product Guide* (PG235) and *HDMI GT Controller LogiCORE IP Product Guide* (PG334). For HDMI Tx, an external clock chip is used to generate the GT reference clock depending on the display resolution. Various other HDMI related clocks are derived from the GT reference clock and generated internally by the HDMI GT controller; only for the DRU a fixed reference clock is provided externally by a Si570 clock chip.

For details on the various clock chips used refer to the *VCK190 Evaluation Board User Guide* (UG1366).

The master reset (pl_resetn0) is generated by the PS during boot and is used as input to the four processing system (PS) reset modules in the PL. Each module generates synchronous, active-Low and active-High interconnect and peripheral resets that drive all IP cores synchronous to the respective, clk_out0, clk_out1, and clk_out2 clock domains.

Apart from these system resets, there are asynchronous resets driven by PS GPIO pins. The respective device drivers control these resets which can be toggled at run-time to reset HLS-based cores. The following table summarizes the PL resets used in this design.

Table 4: **System and User Resets**

Reset Source	Purpose
pl0_resetn	PL reset for proc_sys_reset modules
rst_processor_150MHz	Synchronous resets for clk_out0 clock domain
rst_processor_105MHz	Synchronous resets for clk_out1 clock domain
rst_processor_200MHz	Synchronous resets for clk_out3 clock domain
lpd_gpio_o 0	Asynchronous reset for the video mixer IP
GPIO for platorm1 – Single Senor	
lpd_gpio_o 1	Asynchronous reset for the demosaic IP
lpd_gpio_o 2	Asynchronous reset for the VPSS CSC IP
lpd_gpio_o 3	Asynchronous reset for the frame buffer write IP
lpd_gpio_o 4	Asynchronous reset for the sensor GPIO
GPIO for platorm2 – Quad Senor	
lpd_gpio_o 1	Asynchronous reset for the demosaic IP stream 0
lpd_gpio_o 2	Asynchronous reset for the VPSS CSC IP stream 0

Table 4: **System and User Resets** *(cont'd)*

Reset Source	Purpose
lpd_gpio_o 3	Asynchronous reset for the frame buffer write IP stream 0
lpd_gpio_o 4	Asynchronous reset for the demosaic IP stream 1
lpd_gpio_o 5	Asynchronous reset for the VPSS CSC IP stream 1
lpd_gpio_o 6	Asynchronous reset for the frame buffer write IP stream 1
lpd_gpio_o 7	Asynchronous reset for the demosaic IP stream 2
lpd_gpio_o 8	Asynchronous reset for the VPSS CSC IP stream 3
lpd_gpio_o 9	Asynchronous reset for the frame buffer write IP stream
lpd_gpio_o 10	Asynchronous reset for the demosaic IP stream 3
lpd_gpio_o 10	Asynchronous reset for the VPSS CSC IP stream 3
lpd_gpio_o 12	Asynchronous reset for the frame buffer write IP stream 3
GPIO for platform3 - HDMI RX	
lpd_gpio_0 1	Asynchronous reset for the VPSS CSC IP
lpd_gpio_0 2	Asynchronous reset for the frame buffer write IP

The following table lists the PL-to-PS interrupts used in this design.

Table 5: **Interrupt from PL to PS**

Interrupt ID	Instance
pl_ps_irq0	HDMI GT Controller
pl_ps_irq1	HDMI Tx subsystem
pl_ps_irq2	Video Mixer
pl_ps_irq3	HDMI I2C
pl_ps_irq4	AXI Performance Monitor
Interrupts specific to platform 1 - Single Sensor	
pl_ps_irq5	Audio formatter memory-mapped to stream
pl_ps_irq6	MIPI RX subsytem
pl_ps_irq7	MIPI I2C
pl_ps_irq8	Frame buffer write interrupt
Interrupts specific to platform 2 - Quad Sensor	
pl_ps_irq5	Audio formatter memory-mapped to stream
pl_ps_irq6	MIPI RX subsytem
pl_ps_irq7	MIPI I2C
pl_ps_irq8	Frame buffer write stream 0
pl_ps_irq9	Frame buffer write stream 1
pl_ps_irq10	Frame buffer write stream 2
pl_ps_irq10	Frame buffer write stream 3
Interrupts specific to platform 3 - HDMI RX	
pl_ps_irq5	Audio formatter memory-mapped to stream

Table 5: **Interrupt from PL to PS** *(cont'd)*

Interrupt ID	Instance
pl_ps_irq6	Audio formatter stream to memory map
pl_ps_irq7	Frame buffer write interrupt
pl_ps_irq8	HDMI RX subsytem

XILINX

Appendix A

Additional Resources and Legal Notices

Xilinx Resources

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

Documentation Navigator and Design Hubs

Xilinx® Documentation Navigator (DocNav) provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. To open DocNav:

- From the Vivado® IDE, select **Help → Documentation and Tutorials**.
- On Windows, select **Start → All Programs → Xilinx Design Tools → DocNav**.
- At the Linux command prompt, enter `docnav`.

Xilinx Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In DocNav, click the **Design Hubs View** tab.
- On the Xilinx website, see the Design Hubs page.

Note: For more information on DocNav, see the Documentation Navigator page on the Xilinx website.

References

These documents provide supplemental material useful with this guide:

1. *Versal Architecture and Product Data Sheet: Overview* (DS950)
2. *VCK190 Evaluation Board User Guide* (UG1366)
3. *Xilinx OpenCV User Guide* (UG1233)
4. *Versal ACAP AI Engine Programming Environment User Guide* (UG1076)
5. https://xilinx.github.io/Vitis_Libraries/vision/api-reference.html#vitis-vision-library-functions
6. https://github.com/Xilinx/Vitis_Libraries/blob/master/vision/L1/include/imgproc/xf_custom_convolution.hpp
7. *Versal ACAP System and Software Developers Guide* (UG1304)
8. *MIPI CSI-2 Receiver Subsystem Product Guide* (PG232)
9. *HDMI 1.4/2.0 Transmitter Subsystem Product Guide* (PG235)
10. *HDMI GT Controller LogiCORE IP Product Guide* (PG334)
11. *Video Processing Subsystem Product Guide* (PG231)
12. *Video Frame Buffer Read and Video Frame Buffer Write LogiCORE IP Product Guide* (PG278)
13. *HDMI 1.4/2.0 Receiver Subsystem Product Guide* (PG236)
14. *AXI4-Stream Infrastructure IP Suite LogiCORE IP Product Guide* (PG085)
15. *Video Mixer LogiCORE IP Product Guide* (PG243)
16. https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841700/Xilinx+ALSA+ASoC+driver
17. https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842519/Xilinx+ALSA+HDMI+Audio+driver
18. https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/75104264/Xilinx+ALSA+Audio+Formatter+driver

Please Read: Important Legal Notices

had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx's limited warranty, please refer to Xilinx's Terms of Sale which can be viewed at https://www.xilinx.com/legal.htm#tos; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx's Terms of Sale which can be viewed at https://www.xilinx.com/legal.htm#tos.

AUTOMOTIVE APPLICATIONS DISCLAIMER

AUTOMOTIVE PRODUCTS (IDENTIFIED AS "XA" IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE ("SAFETY APPLICATION") UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD ("SAFETY DESIGN"). CUSTOMER SHALL, PRIOR TO USING OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.

Copyright