H.264/H.265 Video Codec Unit v1.0

LogiCORE IP Product Guide

Vivado Design Suite

PG252 December 20, 2017

XILINX
ALL PROGRAMMABLE™

Table of Contents

Introduction

The Xilinx® LogiCORE™ IP H.264/H.265 Video Codec Unit (VCU) core for Zynq® UltraScale+™ MPSoC devices is capable of performing video compression and decompression of simultaneous video streams at resolutions up to 3840×2160 pixels at 60 frames per second (4K UHD at 60Hz). H.264/H.265 functionality is implemented as an embedded hard IP inside Zynq UltraScale+ MPSoC EV devices. The VCU is suitable for applications including but not limited to video surveillance and video over IP connectivity including video conferencing, embedded vision, biomedical instrumentation, etc.

Features

- Multi-standard encoding/decoding support, including:
  - ISO MPEG-4 Part 10: Advanced Video Coding (AVC)/ITU H.264
  - ISO MPEG-H Part 2: High Efficiency Video Coding (HEVC)/ITU H.265
  - HEVC: Main, Main Intra, Main10, Main10 Intra, Main 4:2:2 10, Main 4:2:2 10 Intra up to 5.1 High Tier
  - AVC: Baseline, Main, High, High10, High 4:2:2 up to 5.2 level
- Support simultaneous encoding and decoding of up to 8 streams with a maximum aggregated bandwidth of 3840x2160@60fps
- Add low latency rate control
- Flexible rate control: CBR, VBR, and Constant QP
- Supports simultaneous encoding and decoding up to 4K UHD resolution at 60Hz
- Supports 8K UHD at reduced frame rate (~15Hz)

LogiCORE™ IP Facts Table	
Core Specifics	
Supported Device Family[1]	Zynq® UltraScale+™ MPSoC Family EV Devices
Supported User Interfaces	AXI4-Lite, AXI4-Memory Mapped
Resources	Performance and Resource Utilization web page
Provided with Core	
Design Files	Encrypted RTL
Example Design	Verilog
Test Bench	Verilog
Constraints File	Xilinx Design Constraints (XDC)
Simulation Model	Verilog or VHDL source HDL Model
Supported S/W Driver	Included in PetaLinux
Tested Design Flows[2]	
Design Entry	Vivado® Design Suite
Simulation	For supported simulators, see the Xilinx Design Tools: Release Notes Guide.
Synthesis	Vivado Synthesis
Support	
Provided by Xilinx at the Xilinx Support web page	

Notes:

1. For a complete list of supported devices, see the Vivado IP catalog.
2. For the supported versions of the tools, see the Xilinx Design Tools: Release Notes Guide.

Features (Continued)

- Progressive support for H.264 and H.265; interlaced support for H.265 in development
- Video input:
  - YCBCR 4:2:2, YCBCR 4:2:0, and Y-only (monochrome)
  - 8- and 10-bit per channel color

Overview

Introduction

The LogiCORE™ IP H.264/H.265 Video Codec Unit (VCU) core supports multi-standard video encoding and decoding, including support for the High-efficiency Video Coding (HEVC) and Advanced Video Coding (AVC) H.264 standards. The unit contains both encode (compress) and decode (decompress) functions, and is capable of simultaneous encode and decode.

The VCU is an integrated block in the programmable logic (PL) of selected Zynq® UltraScale™+ MPSoCs with no direct connections to the processing system (PS), and contains encoder and decoder interfaces. The VCU also contains additional functions that facilitate the interface between the VCU and the PL. VCU operation requires the application processing unit (APU) to service interrupts to coordinate data transfer. The encoder is controlled by the APU through a task list prepared in advance, and the APU response time is not in the execution critical path. The VCU has no audio support. Audio encoding and decoding can be done in software using the PS or through soft IP in the PL. Figure 1-1 shows the top-level block diagram with the VCU core.

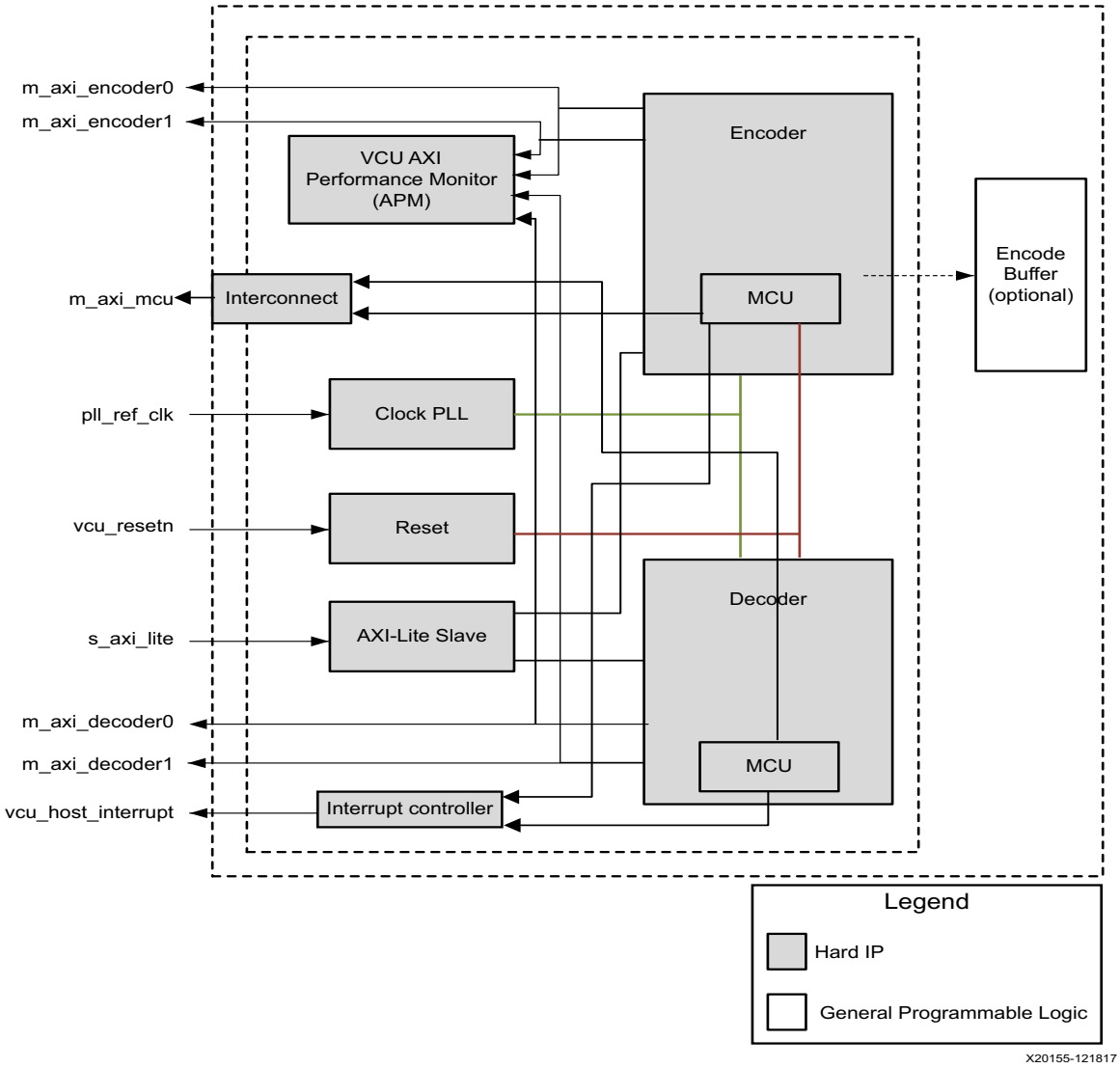


Figure 1-1: **Top-level Block Diagram**

Encoder Block Overview

The Encoder engine is designed to process video streams using the HEVC (ISO/IEC 23008-2 High Efficiency Video Coding) and AVC (ISO/IEC 14496-10 Advanced Video Coding) standards. It provides complete support for these standards, including support for 8-bit and 10-bit color, Y-only (monochrome), 4:2:0 and 4:2:2 chroma formats, up to 4K UHD at 60Hz performance. Figure 3-1 shows the top-level interfaces and detailed architecture of the Encoder block. The encoder also contains global registers, an interrupt controller, and a timer. The encoder is controlled by a microcontroller (MCU) subsystem. VCU applications running on the APU use the Xilinx VCU Control Software library API to interact with the encoder microcontroller. See *Zynq UltraScale+ MPSoC Data Sheet: DC and AC Switching Characteristics* [Ref 13] for more details. The microcontroller firmware (MCU Firmware) is not user modifiable.

A 32-bit AXI4-Lite interface is used by the APU to control the MCU (to configure encoding parameters). Two 128-bit AXI4 master interfaces are used to move video data and metadata to and from the system memory. A 32-bit AXI4 master interface is used to fetch the MCU software (instruction cache interface) and load/store additional MCU data (data cache interface).

Decoder Block Overview

The Decoder block is capable of processing video streams using the HEVC (ISO/IEC 23008-2 High Efficiency Video Coding) and AVC (ISO/IEC 14496-10 Advanced Video Coding) standards. It provides a complete support for these standards, including support for 8-bit and 10-bit color depth, Y-only (monochrome), 4:2:0 and 4:2:2 chroma formats, up to 4K UHD at 60Hz performance. It also contains global registers, an interrupt controller, and a timer.

The VCU decoder is controlled by a microcontroller (MCU) subsystem. A 32-bit AXI4-Lite slave interface is used by the APU to control the MCU. Two 128-bit AXI4 master interfaces are used to move video data and metadata to and from the system memory. A 32-bit AXI4 master interface is used to fetch the MCU software (instruction cache interface) and load/store additional MCU data (data cache interface). VCU applications running on the APU use the Xilinx VCU Control Software library API to interact with the decoder microcontroller. See VCU Control Software in Chapter 11 for more information. The microcontroller firmware is not user modifiable.

The decoder includes control registers, a bridge unit and a set of internal memories. The bridge unit manages the request arbitration, burst addresses, and burst lengths for all external memory accesses required by the decoder.

MCU Overview

The Encoder and Decoder blocks each implement a 32-bit MCU to handle interaction with the hardware blocks. The MCU receives commands from the APU, parses the command into multiple slice- or tile-level commands, and executes them on the encoder and decoder blocks. After the command is executed, the MCU communicates the status to the APU and the process is repeated.

Applications

The VCU core is dedicated circuitry located in the PL to enable maximum flexibility for a wide selection of use cases, memory bandwidth being a key driver. Whether the application requires simultaneous 4K UHD at 60 Hz encoding and decoding or a single SD stream to be processed, a system design and memory topology can be implemented that balances performance, optimization, and integration for the specific use case. Figure 1-2 shows the use case example where the VCU core works with the PS and the PL DDR external memory.

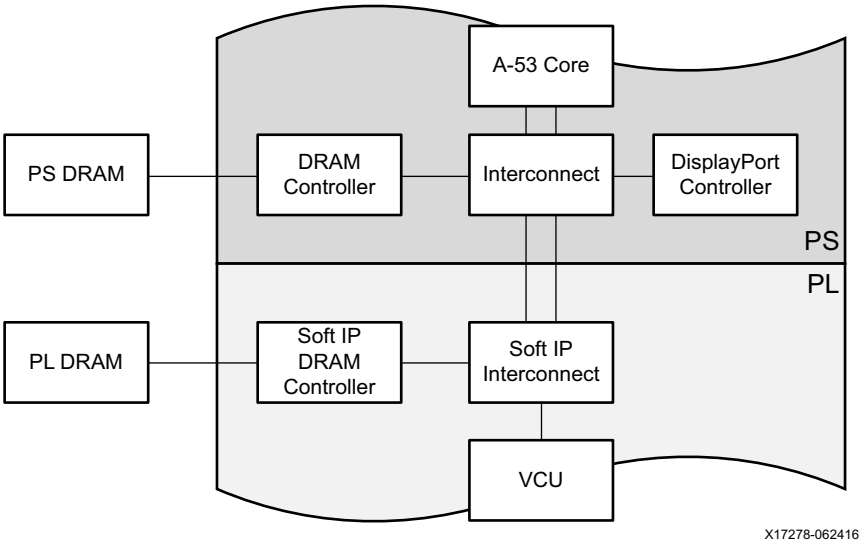


Figure 1-2: **VCU Application**

Unsupported Features

The following features are not supported:

- Scalable Video coding in AVC/HEVC
- Interlaced video format
- H.264 (AVC)
  - Encoder:
    - Lossless mode (transform bypass, I_PCM)
  - Decoder:
    - Flexible macroblock ordering (FMO)
    - Arbitrary slice ordering (ASO)
    - Redundant slice (RS)
    - Dynamic resolution/chroma format/profile/level change within a single stream
- H.265 (HEVC)
  - Encoder:
    - Sample adaptive offset (SAO) filter
    - Asymmetric motion partition (AMP)
    - Lossless mode (transquant bypass, PCM)
    - Transform skip mode

- Wavefront Parallel Processing (WPP)

◦ Decoder:

- Dynamic resolution/chroma format/profile/level change within a single stream

See *Zynq UltraScale+ MPSoC Production Errata* [Ref 9] for more information.

Licensing and Ordering Information

License Type

This Xilinx LogiCORE IP module is only available on Zynq® UltraScale+™ MPSoC Family EV Devices and is provided at no additional cost with the Xilinx Vivado Design Suite under the terms of the Xilinx End User License. Information about this and other Xilinx LogiCORE IP modules is available at the Xilinx Intellectual Property page. For information about pricing and availability of other Xilinx LogiCORE IP modules and tools, contact your local Xilinx sales representative.

For more information, visit the Zynq UltraScale+ MPSoC product page.

Implementation of H.264 or H.265 video compression standards may require a license from third parties as well as the payment of royalties; further information may be obtained from individual patent holders and industry consortiums such as MPEG LA and HEVC Advance.

Product Specification

Standards

The Encoder and Decoder blocks are compatible with the following standards:

- ISO/IEC 14496-10:2014(en)
  Information technology — Coding of audio-visual objects — Part 10: Advanced Video Coding
- Recommendation ITU — T H.264 | International Standard ISO/IEC 14496-10: Advanced video coding for generic audiovisual services
- Recommendation ITU — T H.265 | International Standard ISO/IEC 23008-2: High efficiency video coding
- ISO/IEC 23008-2:2017
  Information technology — High efficiency coding and media delivery in heterogeneous environments — Part 2: High efficiency video coding

Performance

The following sections detail the performance characteristics of the H.264/H.265 Video Codec Unit.

Maximum Frequencies

The following are typical clock frequencies for the target devices are described in the *Zynq UltraScale+ MPSoC Data Sheet: DC and AC Switching Characteristics ref*. The maximum achievable clock frequency of the system can vary. The maximum achievable clock frequency and all resource counts can be affected by other tool options, additional logic in the device, using a different version of Xilinx® tools and other factors.

Throughput

The VCU supports simultaneous encoding and decoding up to 4K UHD resolution at 60Hz. This throughput can be a single stream at 4K UHD or can be divided into up to eight smaller

streams of up to 1080p at 30 Hz. Several combinations of one to eight streams can be supported with different resolutions provided the cumulative throughput does not exceed 4K UHD at 60 Hz.

Resource Utilization

Streams of 4K UHD at 60Hz consume the majority of the bandwidth of the external memory interfaces and the majority of the ARM AMBA® AXI4 bus bandwidth between the Processing System and the Programmable Logic. See DDR Memory Footprint Requirement in Chapter 3 for more information.

For details about resource utilization, visit Performance and Resource Utilization.

Core Interfaces and Register Space

The core has the following interfaces:

- Four 128-bit AXI master interfaces to communicate with external memory
- One 32-bit AXI master interface for control communication
- An AXI4-Lite interface for communicating with the application processing unit (APU)

The four 128-bit AXI master interfaces are used for moving video data into and out of external memory through the PS memory and the PL memory interfaces. Two AXI interfaces are allocated to encoding and two are allocated to decoding.

Port Descriptions

The VCU core top-level signaling interface is shown in Figure 2-1.

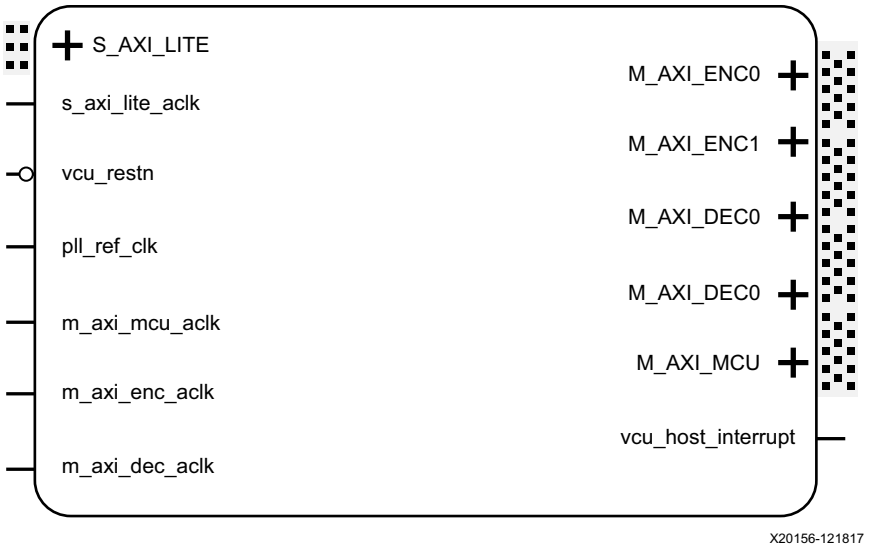


Figure 2-1: **VCU Core Top-Level Signaling Interface**

Table 2-1 summarizes the core interfaces.

Table 2-1: **VCU Interfaces**

Interface Name	Interface Type	Description
M_AXI_ENC0	AXI-4 memory mapped master interface	128-bit memory mapped interface for Encoder block. Refer to Encoder Block Overview in Chapter 1 for more details.
M_AXI_ENC1	AXI-4 memory mapped master interface	128-bit memory mapped interface for Encoder block. Refer to Encoder Block Overview in Chapter 1 for more details.
M_AXI_DEC0	AXI-4 memory mapped master interface	128-bit memory mapped interface for Decoder block. Refer to Decoder Block Overview in Chapter 1 for more details.
M_AXI_DEC1	AXI-4 memory mapped master interface	128-bit memory mapped interface for Decoder block. Refer to Decoder Block Overview in Chapter 1 for more details.
M_AXI_MCU	AXI-4 memory mapped master interface	32-bit memory mapped interface for MCU. Refer to MCU Overview in Chapter 1 for more details.
S_AXI__LITE	AXI4-Lite memory mapped slave interface	AXI4-Lite memory mapped interface for external master access. Refer to MCU Overview in Chapter 1 for more details.

Common Interface Signals

Table 2-2 summarizes the signals which are either shared by, or not part of the dedicated AXI4 interfaces.

Table 2-2: **VCU Ports**

Port Name	Direction	Description
m_axi_enc_aclk	Input	AXI clock input for M_AXI_VCU_ENCODER0 and M_AXI_VCU_ENCODER1
s_axi_lite_aclk	Input	AXI clock input for S_AXI_PL_VCU_LITE
pll_ref_clk	Input	PLL reference clock input
vcu_resetn	Input	Active Low reset input from PL
vcu_host_interrupt	Output	Active High interrupt output from VCU. Can be mapped to PL-PS interrupt pin.
m_axi_dec_aclk	Input	AXI input clock for M_AXI_VCU_DECODER0 and M_AXI_VCU_DECODER1
m_axi_mcu_aclk	Input	Input clock for M_AXI_MCU interface

Register Space

The registers implemented in the embedded VCU are documented in Chapter 11, Application Software Development. For more details about bit level definition of registers, refer to [Ref 11].

Soft IP Registers

The Zynq UltraScale+ VCU soft IP implements registers in the programmable logic. Table 2-3 summarizes the soft IP registers.

Table 2-3: **Soft IP Registers**

Number	Parameter	Data Type	Address Offset	Definition
Video Configuration				
1	VCU_ECODER_ENABLE	Integer	0x41000	1 = Enable 0 = Disable
2	VCU_DECODER_ENABLE	Integer	0x41004	1 = Enable 0 = Disable
3	VCU_MEMORY_DEPTH	Integer	0x41008	Number of entries in Encoder Buffer
4	VCU_ENC_COLOR_DEPTH	Integer	0x4100C	0 = 8 bits per pixel 1 = 8 and 10 bits per pixel

Table 2-3: **Soft IP Registers *(Cont'd)***

Number	Parameter	Data Type	Address Offset	Definition
5	VCU_ENC_VERTICAL_RANGE	Integer	0x41010	0 = Low 1 = Medium 2 = High
6	VCU_ENC_FRAME_SIZE_X	Integer	0x41014	Horizontal pixel size
7	VCU_ENC_FRAME_SIZE_Y	Integer	0x41018	Vertical pixel size
8	VCU_ENC_COLOR_FORMAT	Integer	0x4101C	0 = 4:2:0 1 = 4:2:2 2 = 4:0:0
9	VCU_ENC_FPS	Integer	0x41020	Denotes frames per second
10	VCU_MCU_CLK	Integer	0x41024	Denotes the MCU clock that needs to be configured in PLL
11	VCU_CORE_CLK	Integer	0x41028	Denotes the CORE clock that needs to be configured in PLL
12	VCU_PLL_BYPASS	Integer	0x4102C	0 = Don't bypass VCU PLL 1 = Bypass VCU PLL and use external clock
13	VCU_ENC_CLK	Integer	0x41030	
14	VCU_PLL_CLK	Integer	0x41034	
15	VCU_ENC_VIDEO_STANDARD	Integer	0x41038	0 = HEVC 1 = H.264 (AVC)
16	VCU_STATUS	Integer	0x4103C	{28'b0, PLL_LOCK_STATUS, POWER_STATUS_VCCAUX, POWER_STATUS_VCUINT, INTERRUPT}
17	VCU_AXI_ENC_CLK	Integer	0x41040	
18	VCU_AXI_DEC_CLK	Integer	0x41044	
19	VCU_AXI_MCU_CLK	Integer	0x41048	
20	VCU_DEC_VIDEO_STANDARD	Integer	0x4104C	0 = HEVC 1 = H.264 (AVC)
21	VCU_DEC_FRAME_SIZE_X	Integer	0x41050	
22	VCU_DEC_FRAME_SIZE_Y	Integer	0x41054	
23	VCU_DEC_FPS	Integer	0x41058	
24	VCU_BUFFER_B_FRAME	Integer	0x4105C	

Table 2-3: **Soft IP Registers *(Cont'd)***

Number	Parameter	Data Type	Address Offset	Definition
25	VCU_WPP_EN	Integer	0x41060	
26	VCU_GASKET_INIT		0x41074	Write Only Register Bit 0= setting it to 1 removes gasket isolation Bit 1= Setting it to 0 asserts reset to VCU. Software needs to de-assert it to 1.

Encoder Block

Introduction

The Encoder block of the Video Codec Unit (VCU) core is a video encoder engine for processing video streams using the H.265 (ISO/IEC 23008-2 High Efficiency Video Coding) and H.264 (ISO/IEC 14496-10 Advanced Video Coding) standards. It provides complete support for these standards, including support for 8-bit and 10-bit color depth, 4:2:0, 4:2:2, and 4:0:0 chroma formats and up to 4K UHD at 60 Hz performance.

Features

Table 3-1 describes the VCU Encoder block features.

Table 3-1: **Encoder Block Features**

Video Coding Feature	H.264	H.265
Performance		
Profiles	Baseline Main High High 10 High 4:2:2	Main Main Intra Main 10 Main 10 Intra Main 4:2:2 10 Main 4:2:2 10 Intra
Levels	up to 5.2(1)	up to 5.1 High Tier(1)
Performance at 667 MHz Eight streams at 1920×1080p at 30 Hz Four streams at 1920×1080p at 60 Hz Two streams at 3840×2160p at 30 Hz One stream at 3840×2160p at 60 Hz One stream at 7680×4320p at 15 Hz	Supported	Supported

Table 3-1: **Encoder Block Features *(Cont'd)***

Video Coding Feature	H.264	H.265
Configurable resolution picture width and height multiple of 8 minimum size: 128×64 maximum width or height: 16384 maximum size: 33.5 megapixel	Supported	Supported
Configurable frame rate	Supported	Supported
Configurable bit rate	Supported	Supported
Coding Tools		
Sample bit depth: 8 bpc, 10 bpc	Supported	Supported
Chroma format: YCbCr 4:2:0, YCbCr 4:2:2, Y-only (monochrome)	Supported	Supported
Slice types: I, P, B	Supported	Supported
Progressive format only	Supported	Supported
Coding block size	16×16 macroblocks	LCU size: 32×32 CU size down to 8×8
Prediction size	Down to 4×4 for intra prediction, down to 8×8 inter prediction	Down to 4×4 for intra prediction, down to 8×8 inter prediction
Transform size	4×4, 8×8	4×4, 8×8, 16×16, 32×32
Intra prediction modes	All intra 4×4, intra 8×8, intra 16×16 modes	All 33 directional modes, planar, DC
Constrained Intra Pred support	Supported	Supported
Motion estimation: 1 reference picture for P slices or 2 reference pictures for B slices	Supported	Supported
Motion estimation and compensation: quarter sample interpolation	Supported	Supported
Motion vector prediction modes	All motion vector prediction modes except spatial direct mode and direct_8×8_inference_flag=0	All motion vector prediction/merge/skip modes
Weighted prediction	Supported	Supported
QP control	Constant per frame, configurable per MB or adaptive per MB	Constant per frame, configurable per LCU or adaptive per CU
Chroma QP offset	Supported	Supported
Scaling lists	Supported	Supported
In-loop deblocking filter	Supported	Supported
Entropy coding	CABAC, CAVLC	CABAC
Configurable CABAC initialization table	Supported	Supported

Table 3-1: **Encoder Block Features *(Cont'd)***

Video Coding Feature	H.264	H.265
Slice support	Supported (slices required above 1080p60 performance)	Supported
Dependent slice support	N/A	Supported
Tile support	N/A	Supported (tiles required above 1080p60 performance)

Notes:
1. Support of 8K15 uses a subset of level 6.

Table 3-2 summarizes the maximum bit rate achievable for different profile/level combinations.

Table 3-2: **Maximum Bit Rate**

Standard	Level	Profile	Maximum bitrate (Mbits/s)
H.264 (AVC)	4.2 (1080p60)	Baseline, Main	50
		High	62.5
		High 10	150
		High 4:2:2	200
	5.2 (2160p60)	Baseline, Main	240
		High	300 (context-adaptive variable-length coding (CAVLC) or context-adaptive binary arithmetic coding (CABAC) Intra-only) 267 (CABAC non-Intra-only)
		High 10	720 (CAVLC or CABAC Intra-only) 267 (CABAC non-Intra-only)
		High 4:2:2	960 (CAVLC or CABAC Intra-only) 267 (CABAC non-Intra-only)
H.265 (HEVC)	4.1 (1080p60) High Tier	Main, Main 10	50
		Main 4:2:2 10	84
		Main 4:2:2 10 Intra	167
	5.1 (2160p60) High Tier	Main, Main 10	160
		Main 4:2:2 10	267
		Main 4:2:2 10 Intra	533

Functional Description

Figure 3-1 shows the top-level interfaces and detailed architecture of the Encoder block.

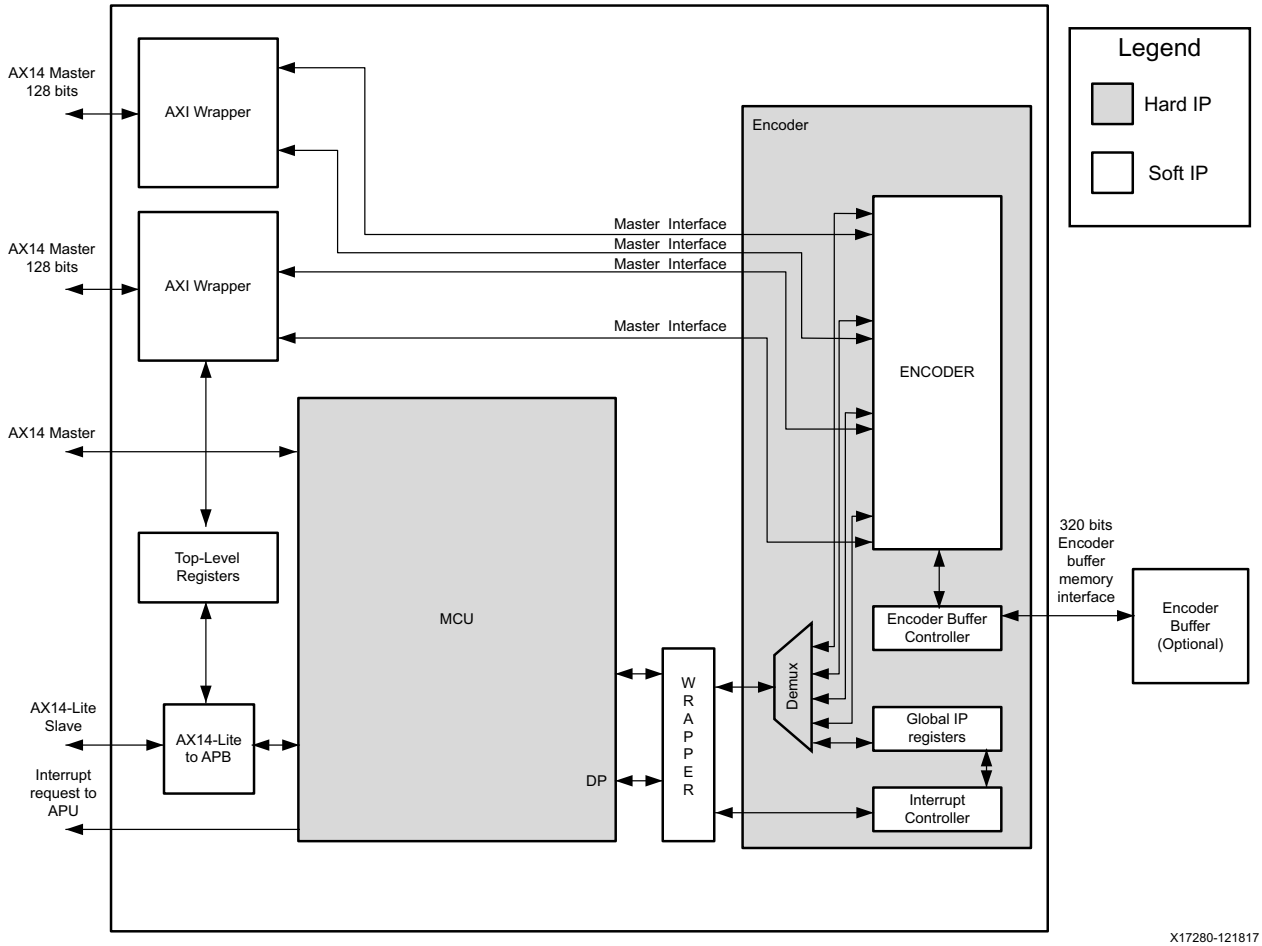


Figure 3-1: **Detailed Architecture of the Encoder Block**

Note: The AXI-4 Master interface from the MCU is multiplexed with the corresponding AXI-4 Master interface from the Decoder. The multiplexer output is available at the embedded VCU.

- The Encoder block includes the compression engines, control registers, an interrupt controller, and an optional encoder buffer with a memory controller. The encoder buffer is connected to UltraRAM or block RAM in the programmable logic and enabled via registers.
- The Encoder block is controlled by a microcontroller unit (MCU) subsystem, including a 32-bit MCU with a 32 KB instruction cache, a 1 KB data cache, and a 32 KB local SRAM.
- A 32-bit AXI4-Lite slave interface is used by the APU to control the MCU for the configuration of encoder parameters, to start/stop processing, to get status and to get results.

- Two 128-bit AXI-4 master interfaces are used to fetch video input data, load and store intermediate data, store compressed data back to memory.
- A 32-bit AXI-4 master interface is used to fetch the MCU software and load/store additional MCU data.

The VCU Control Software can change encoding parameters and even change between H.264 and H.265 encoding dynamically, however, the available memory and bandwidth must be selected to support the worst case needed by the application.

Interfaces and Ports

Table 3-3 shows the list of ports/interfaces of the top-level Encoder block.

Table 3-3: **Encoder Ports**

Name	Size (bits)	Dir	Description
Clocks and Resets			
pll_ref_clk	1	Input	Reference clock to the VCU PLL from PL
m_axi_enc_aclk	1	Input	Memory interface Encoder clock
m_axi_dec_aclk	1	Input	Memory interface Decoder clock
s_axi_lite_aclk	1	Input	AXI-Lite clock
m_axi_mcu_aclk	1	Input	VCU MCU AXI interface clock from PL
vcu_resetn	1	Input	Active Low. VCU reset from PL
VCU-Interrupt (s_axi_lite_aclk)			
vcu_host_interrupt	1	Output	Active High. Interrupt from VCU to PS
VCU Encoder block, 128-bit AXI Master Interface 0 (m_axi_enc_aclk domain)			
vcu_pl_enc_araddr0	44	Output	AXI Master read address bus for interface 0
vcu_pl_enc_arburst0	2	Output	AXI Master read burst type signal
vcu_pl_enc_arid0	4	Output	AXI Master read burst ID for interface 0
vcu_pl_enc_arlen0	8	Output	AXI Master read burst length for interface 0
pl_vcu_enc_arready0	1	Input	AXI Master read address ready for interface 0
vcu_pl_enc_arsize0	3	Output	AXI Master read interface size for interface 0
vcu_pl_enc_arvalid0	1	Output	AXI Master read address valid for interface 0
vcu_pl_enc_awaddr0	44	Output	AXI Master write address for interface 0
vcu_pl_enc_awburst0	2	Output	AXI Master write burst type for interface 0
vcu_pl_enc_awid0	4	Output	AXI Master write burst ID for interface 0
vcu_pl_enc_awlen0	8	Output	AXI Master write burst length for interface 0
pl_vcu_enc_awready0	1	Input	AXI Master write address ready for interface 0
vcu_pl_enc_awsize0	3	Output	AXI Master write burst size for interface 0

Table 3-3: **Encoder Ports *(Cont'd)***

Name	Size (bits)	Dir	Description
vcu_pl_enc_awvalid0	1	Output	AXI Master write address valid for interface 0
pl_vcu_enc_bresp0	2	Input	AXI Master write response for interface 0
vcu_pl_enc_bready0	1	Output	AXI Master write response ready for interface 0
pl_vcu_enc_bvalid0	1	Input	AXI Master write response valid for interface 0
pl_vcu_enc_bid0	4	Input	AXI Master write response ID for interface 0
pl_vcu_enc_rdata0	128	Input	AXI Master read data for interface 0
pl_vcu_enc_rid0	4	Input	AXI Master read ID signal for interface 0
pl_vcu_enc_rlast0	1	Input	AXI Master read last signal for interface 0
vcu_pl_enc_rready0	1	Output	AXI Master read ready signal for interface 0
Pl_vcu_enc_rresp0	2	Input	AXI Master read response signal for interface 0
pl_vcu_enc_rvalid0	1	Input	AXI Master read valid signal for interface 0
vcu_pl_enc_wdata0	128	Output	AXI Master write data for interface 0
vcu_pl_enc_wlast0	1	Output	AXI Master write last signal for interface 0
pl_vcu_enc_wready0	1	Input	AXI Master write ready signal for interface 0
vcu_pl_enc_wvalid0	1	Output	AXI Master write valid signal for interface 0
vcu_pl_enc_awprot0	1	Output	AXI Master write protection signal for interface 0, controlled from SLCR
vcu_pl_enc_arprot0	1	Output	AXI Master read protection signal for interface 0, controlled from SLCR
vcu_pl_enc_awqos0	4	Output	AXI Master write QOS signal for interface 0, controlled from SLCR
vcu_pl_enc_arqos0	4	Output	AXI Master read QOS signal for interface 0, controlled from SLCR
vcu_pl_enc_awcache0	4	Output	AXI Master write cache signal for interface 0,controlled from SLCR
vcu_pl_enc_arcache0	4	Output	AXI Master read cache signal for interface 0,controlled from SLCR
VCU Encoder block, 128-bit AXI Master Interface 1 (m_axi_enc_aclk domain)			
vcu_pl_enc_araddr1	44	Output	AXI Master read address bus for interface 1
vcu_pl_enc_arburst1	2	Output	AXI Master read burst type signal
vcu_pl_enc_arid1	4	Output	AXI Master read burst ID for interface 1
vcu_pl_enc_arlen1	8	Output	AXI Master read burst length for interface 1
pl_vcu_enc_arready1	1	Input	AXI Master read address ready for interface 1
vcu_pl_enc_arsize1	3	Output	AXI Master read interface size for interface 1
vcu_pl_enc_arvalid1	1	Output	AXI Master read address valid for interface 1
vcu_pl_enc_awaddr1	44	Output	AXI Master write address for interface 1

Table 3-3: **Encoder Ports *(Cont'd)***

Name	Size (bits)	Dir	Description
vcu_pl_enc_awburst1	2	Output	AXI Master write burst type for interface 1
vcu_pl_enc_awid1	4	Output	AXI Master write burst ID for interface 1
vcu_pl_enc_awlen1	8	Output	AXI Master write burst length for interface 1
pl_vcu_enc_awready1	1	Input	AXI Master write address ready for interface 1
vcu_pl_enc_awsize1	3	Output	AXI Master write burst size for interface 1
vcu_pl_enc_awvalid1	1	Output	AXI Master write address valid for interface 1
Pl_vcu_enc_bresp1	2	Input	AXI Master write response for interface 1
vcu_pl_enc_bready1	1	Output	AXI Master write response ready for interface 1
pl_vcu_enc_bvalid1	1	Input	AXI Master write response valid for interface 1
pl_vcu_enc_bid1	4	Input	AXI Master write response ID for interface 1
pl_vcu_enc_rdata1	128	Input	AXI Master read data for interface 1
pl_vcu_enc_rid1	4	Input	AXI Master read ID signal for interface 1
pl_vcu_enc_rlast1	1	Input	AXI Master read last signal for interface 1
vcu_pl_enc_rready1	1	Output	AXI Master read ready signal for interface 1
Pl_vcu_enc_rresp1	2	Input	AXI Master read response signal for interface 1
pl_vcu_enc_rvalid1	1	Input	AXI Master read valid signal for interface 1
vcu_pl_enc_wdata1	128	Output	AXI Master write data for interface 1
vcu_pl_enc_wlast1	1	Output	AXI Master write last signal for interface 1
pl_vcu_enc_wready1	1	Input	AXI Master write ready signal for interface 1
vcu_pl_enc_wvalid1	1	Output	AXI Master write valid signal for interface 1
vcu_pl_enc_awprot1	1	Output	AXI Master write protection signal for interface 1, controlled from SLCR
vcu_pl_enc_arprot1	1	Output	AXI Master read protection signal for interface 1, controlled from SLCR
vcu_pl_enc_awqos1	4	Output	AXI Master write QOS signal for interface 1, controlled from SLCR
vcu_pl_enc_arqos1	4	Output	AXI Master read QOS signal for interface 1, controlled from SLCR
vcu_pl_enc_awcache1	4	Output	AXI Master write cache signal for interface 1, controlled from SLCR
vcu_pl_enc_arcache1	4	Output	AXI Master read cache signal for interface 1, controlled from SLCR
VCU Encoder- 32-bit AXI Master MCU Instruction and Data Cache Interface			
vcu_pl_mcu_m_axi_ic_dc_araddr	44	Output	AXI Master read address bus for MCU
vcu_pl_mcu_m_axi_ic_dc_arburst	2	Output	AXI Master read burst type signal
vcu_pl_mcu_m_axi_ic_dc_arcache	4	Output	AXI Master read cache for MCU

Table 3-3: **Encoder Ports *(Cont'd)***

Name	Size (bits)	Dir	Description
vcu_pl_mcu_m_axi_ic_dc_arid	3	Output	AXI Master read burst ID for MCU
vcu_pl_mcu_m_axi_ic_dc_arlen	8	Output	AXI Master read burst length for MCU
vcu_pl_mcu_m_axi_ic_dc_arlock	1	Output	AXI Master read lock for MCU
vcu_pl_mcu_m_axi_ic_dc_arprot	3	Output	AXI Master read protection signal for MCU
vcu_pl_mcu_m_axi_ic_dc_arqos	4	Output	AXI Master read QoS for MCU
pl_vcu_mcu_m_axi_ic_dc_arready	1	Input	AXI Master read address ready for MCU
vcu_pl_mcu_m_axi_ic_dc_arsize	3	Output	AXI Master read address size for MCU
vcu_pl_mcu_m_axi_ic_dc_arvalid	1	Output	AXI Master read address valid for MCU
vcu_pl_mcu_m_axi_ic_dc_awaddr	44	Output	AXI Master write address for MCU
vcu_pl_mcu_m_axi_ic_dc_awburst	2	Output	AXI Master write burst type for MCU
vcu_pl_mcu_m_axi_ic_dc_awcache	4	Output	AXI Master write cache for MCU
vcu_pl_mcu_m_axi_ic_dc_awid	3	Output	AXI Master write address ID for MCU
vcu_pl_mcu_m_axi_ic_dc_awlen	8	Output	AXI Master write burst length for MCU
vcu_pl_mcu_m_axi_ic_dc_awlock	1	Output	AXI Master write lock for MCU
vcu_pl_mcu_m_axi_ic_dc_awprot	3	Output	AXI Master write protection for MCU
vcu_pl_mcu_m_axi_ic_dc_awqos	4	Output	AXI Master write QoS for MCU
pl_vcu_mcu_m_axi_ic_dc_awready	1	Input	AXI Master write address ready signal for MCU
vcu_pl_mcu_m_axi_ic_dc_awsize	3	Output	AXI Master write burst size signal for MCU
vcu_pl_mcu_m_axi_ic_dc_awvalid	1	Output	AXI Master write address valid signal for MCU
pl_vcu_mcu_m_axi_ic_dc_bid	3	Input	AXI Master write response ID for MCU
vcu_pl_mcu_m_axi_ic_dc_bready	1	Output	AXI Master write response ready signal for MCU
pl_vcu_mcu_m_axi_ic_dc_bresp	2	Input	AXI Master write response for MCU
pl_vcu_mcu_m_axi_ic_dc_bvalid	1	Input	AXI Master write response valid signal for MCU
pl_vcu_mcu_m_axi_ic_dc_rdata	32	Input	AXI Master read data signal for MCU
pl_vcu_mcu_m_axi_ic_dc_rid	3	Input	AXI Master read ID signal for MCU
pl_vcu_mcu_m_axi_ic_dc_rlast	1	Input	AXI Master read last signal for MCU
vcu_pl_mcu_m_axi_ic_dc_rready	1	Output	AXI Master read ready signal for MCU
pl_vcu_mcu_m_axi_ic_dc_rresp	2	Input	AXI Master read response signal for MCU
pl_vcu_mcu_m_axi_ic_dc_rvalid	1	Input	AXI Master read valid signal for MCU
vcu_pl_mcu_m_axi_ic_dc_wdata	32	Output	AXI Master write data signal for MCU
vcu_pl_mcu_m_axi_ic_dc_wlast	1	Output	AXI Master write last signal for MCU
pl_vcu_mcu_m_axi_ic_dc_wready	1	Input	AXI Master write ready signal for interface 1
vcu_pl_mcu_m_axi_ic_dc_wstrb	4	Output	AXI Master write strobe signal

Table 3-3: **Encoder Ports *(Cont'd)***

Name	Size (bits)	Dir	Description
vcu_pl_mcu_m_axi_ic_dc_wvalid	1	Output	AXI Master write valid signal for MCU
AXI-Lite Slave Interface (s_axi_lite_aclk domain)			
pl_vcu_awaddr_axi_lite	20	Input	AXI Lite write address bus
pl_vcu_awprot_axi_lite	3	Input	AXI Lite write protection signal
pl_vcu_awvalid_axi_lite	1	Input	AXI Lite write address valid signal
vcu_pl_awready_axi_lite	1	Output	AXI Lite write address ready signal
pl_vcu_wdata_axi_lite	32	Input	AXI Lite write data channel
pl_vcu_wstrb_axi_lite	4	Input	AXI Lite write strobe signal
pl_vcu_wvalid_axi_lite	1	Input	AXI Lite write data valid signal
vcu_pl_wready_axi_lite	1	Output	AXI Lite write ready signal
vcu_pl_bresp_axi_lite	2	Output	AXI Lite write response channel
vcu_pl_bvalid_axi_lite	1	Output	AXI Lite write response valid signal
pl_vcu_bready_axi_lite	1	Input	AXI Lite write response ready signal
pl_vcu_araddr_axi_lite	20	Input	AXI Lite read address channel
pl_vcu_arprot_axi_lite	3	Input	AXI Lite read channel protection signal
pl_vcu_arvalid_axi_lite	1	Input	AXI Lite read address valid signal
vcu_pl_arready_axi_lite	1	Output	AXI Lite read address ready signal
vcu_pl_rdata_axi_lite	32	Output	AXI Lite read data bus
vcu_pl_rresp_axi_lite	2	Output	AXI Lite read response signal
vcu_pl_rvalid_axi_lite	1	Output	AXI Lite read data valid signal
pl_vcu_rready_axi_lite	1	Input	AXI Lite read data ready signal
VCU Encoder Buffer Interface			
Vcu_pl_enc_al_l2c_rvalid	1	Output	Read data valid
Pl_vcu_enc_al_l2c_rready	1	Input	Read data ready
Vcu_pl_enc_al_l2c_addr	17	Output	Address
Pl_vcu_enc_al_l2c_rdata	320	Input	Read data
Vcu_pl_enc_al_l2c_wvalid	1	Output	Write data valid
Vcu_pl_enc_al_l2c_wdata	320	Output	Write data

Clocking

Refer to Chapter 6, Clocking and Resets for more information on clocking.

Reset

Refer to Chapter 6, Clocking and Resets for more information on resets.

MCU Subsystem

The Encoder block includes an embedded MCU that runs the MCU Firmware and controls the hardware Encoder core. Refer to Chapter 5, Microcontroller Unit Overview for more information on the MCU.

Data Path

The Encoder block has two 128-bit AXI4 master interfaces to fetch video data from external DDR memory attached to either the Processing System (PS) or the Programmable Logic (PL).

The data fetched from memory includes:

- Source frame pixels
- Reference frame pixels
- Reference frame motion vectors
- Slice/frame parameters: lambda table, scaling lists, QP control, QP table
- Residual data (H.264 only)

The data written to memory includes:

- Residual data (H.264 only)
- Reconstructed frame pixels
- Reconstructed frame motion vectors
- Encoded bitstream

Control Path

The MCU slave interface is accessed once per frame by the APU, which sends a frame-level command to the IP core. This interface does not require a fast data path. Interrupts are triggered at frame level to wake up the APU at the end of each frame processing. These commands are processed by the embedded MCUs, which generate slice- and tile-level commands to the video encoder hardware. For more information, refer to Chapter 5, Microcontroller Unit Overview.

The Encoder Buffer

The buffer memory controller of the Encoder block manages the read and write access to the encoder buffer, which stores pixel data from the reference frames. It pre-fetches data blocks from the reference frames in the system memory and stores them in the encoder buffer. The encoder buffer stores Luma and Chroma pixels from the reference frames so that they are present in the buffer when needed by the encoder. The encoder buffer must be one contiguous memory region and should be aligned to a 32-byte boundary. Refer to the *Zynq UltraScale+ MPSoC Data Sheet: Overview* [Ref 12] to see the device memory available per EV device.

IMPORTANT: *Using the Encoder block buffer reduces the external DDR memory bandwidth requirement. Refer to Vivado Integrated Design Environment in Chapter 10 for more information regarding memory bandwidth requirements.*

Table 3-4 and Table 3-5 show the required encoder buffer memory size for 4:2:0 and 4:2:2 sampling formats, respectively.

Table 3-4: **Encoder Buffer Memory Requirements for 4:2:0 Sampling**

Encoder Configuration	1920×1080	3840×2160	
	8 bpc	8 bpc	10 bpc
B-frame=NONE Motion Vector Range=LOW	105 MB	291 MB	364 MB
B-frame=STANDARD Motion Vector Range=MEDIUM	395 MB	1,128 MB	1,410 MB
B-frame=HIERARCHICAL Motion Vector Range=HIGH	761 MB	2,250 MB	2,813 MB

Table 3-5: **Encoder Buffer Memory Requirements for 4:2:2 Sampling**

Encoder Configuration	1920×1080	3840×2160	
	8 bpc	8 bpc	10 bpc
B-frame=NONE Motion Vector Range=LOW	139 MB	388 MB	485 MB
B-frame=STANDARD Motion Vector Range=MEDIUM	526 MB	1,504 MB	1,880 MB
B-frame=HIERARCHICAL Motion Vector Range=HIGH	1,014 MB	3,000 MB	3,750 MB

DDR Memory Footprint Requirement

DDR memory buffer size depends on the following:

- Video resolution
- Chroma sub-sampling

- Color depth
- Coding standard

The number of buffers depend on the following:

- Coding standard – H.264 or H.265
- Group of Picture (GOP) pattern

Table 3-6 shows the worst-case memory footprint for various encoding schemes. Use IP Integrator to calculate the memory footprint required for your specific use case.

Table 3-6: **Encoder Block Memory Footprint**

Example with 2 B-frames	720p			1080p			2160p		
	Buffers	Per Buffer	Total	Buffers	Per Buffer	Total	Buffers	Per Buffer	Total
Source frame	5	2.3 MB	12 MB	5	5.3 MB	27 MB	5	21.1 MB	106 MB
Reference frames	3	2.2 MB	7 MB	3	4.9 MB	15 MB	3	19.8 MB	60 MB
Reconstructed frame	1	2.2 MB	3 MB	1	4.9 MB	5 MB	1	19.8 MB	20 MB
Motion vector buffer	4	0.1 MB	1 MB	4	0.1 MB	1 MB	4	0.5 MB	2 MB
Bitstream buffer	2	0.5 MB	2 MB	2	1.2 MB	3 MB	2	4.9 MB	10 MB
Other buffers	1	0.0 MB	1 MB	1	0.0 MB	1 MB	1	0.0 MB	1 MB
Total			**26 MB**			**52 MB**			**199 MB**

Encoder Buffer Memory and Bandwidth

The optional encoder buffer can be used to reduce the memory bandwidth. This option can slightly reduce the video quality. See the CacheLevel2 Encoder setting for more information. Aside from the size, there are no user controls for tuning the encoder buffer usage.

Table 3-7: **Device Memory**

	ZU4EV	ZU5EV	ZU7EV
Block RAM (MB)	4.5	5.1	11.0
UltraRam (MB)	13.5	18.0	27.0

Source Frame Format

The source frame buffer contains the input frame pixels. It contains two parts: luminance pixels (luma) followed by chrominance pixels (chroma). Luma pixels are stored in pixel raster scan order, shown in Figure 3-2. Chroma pixels are stored in U/V-interleaved pixel raster scan order, therefore the chroma portion is half the size of the luma portion when using a

4:2:0 format and the same size as the luma portion when using a 4:2:2 format. The encoder picture buffer must be one contiguous memory region.

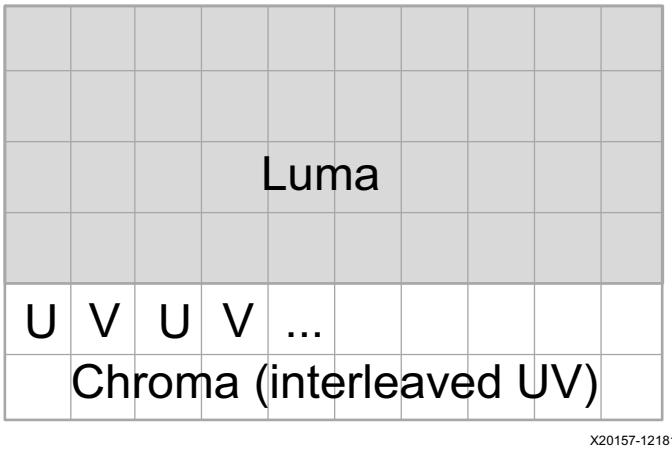


Figure 3-2: **Frame Layout**

Two packing formats are supported in external memory: 8 bits per component or 10 bits per component, shown in Table 3-8 and Table 3-9, respectively. The 8-bit format can only be used for an 8-bit component depth and the 10-bit format can only be used for a 10-bit component depth.

Table 3-8: **Source Frame Buffer Format with 8-bit Component Format**

255 248	...	31 24	23 16	15 8	7 0
$Y_{x+31,y}$	...	$Y_{x+3,y}$	$Y_{x+2,y}$	$Y_{x+1,y}$	$Y_{x,y}$
...(all luma in pixel raster scan order)...					
255 248	247 240	...	31 24	23 16	15 8	7 0
$V_{x+15,y}$	$U_{x+15,y}$	...	$V_{x+1,y}$	$U_{x+1,y}$	$V_{x,y}$	$U_{x,y}$
...(all interleaved chroma in pixel raster scan order)...						

Table 3-9: **Source Frame Buffer Format with 10-bit Component Format**

255	254	253 244							31	30	29 20	19 10	9 0
0	0	$V_{x+23,y}$							0	0	$Y_{x+2,y}$	$Y_{x+1,y}$	$Y_{x,y}$
...(all luma in pixel raster scan order)...													
255	254	253 244	...	63	62	61 52	51 42	41 32	31	30	29 20	19 10	9 0
0	0	$V_{x+11,y}$	...	0	0	$V_{x+2,y}$	$U_{x+2,y}$	$V_{x+1,y}$	0	0	$U_{x+1,y}$	$V_{x,y}$	$U_{x,y}$
...(all interleaved chroma in pixel raster scan order)...													

The encoder buffer must be one contiguous memory region and should be aligned to a 32-byte boundary.

The frame buffer width (pitch) may be larger than the frame width. When the pitch is greater than the frame width, pixels in each line beyond the picture width are ignored, as illustrated in Figure 3-3.

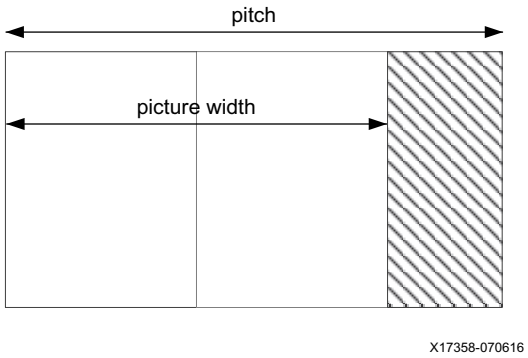


Figure 3-3: **Frame Buffer Pitch**

Encoder Block Register Overview

Table 3-10 lists the Encoder block registers.

Table 3-10: **Encoder Registers**

Register	Address	Width	Type	Reset Value	Description
MCU_RESET	0×A0009000	32	mixed	0x00000000	MCU Subsystem Reset
MCU_RESET_MODE	0xA0009004	32	mixed	0x00000001	MCU Reset Mode
MCU_STA	0xA0009008	32	mixed	0x00000000	MCU Status
MCU_WAKEUP	0xA000900C	32	mixed	0x00000000	MCU Wake-up
MCU_ADDR_OFFSET_IC0	0xA0009010	32	rw	0x00000000	MCU Instruction Cache Address Offset 0
MCU_ADDR_OFFSET_IC1	0xA0009014	32	rw	0x00000000	MCU Instruction Cache Address Offset 1
MCU_ADDR_OFFSET_DC0	0xA0009018	32	rw	0x00000000	MCU Data Cache Address Offset 0
MCU_ADDR_OFFSET_DC1	0xA000901C	32	rw	0x00000000	MCU Data Cache Address Offset 1
ITC_MCU_IRQ	0xA0009100	32	mixed	0x00000000	MCU Interrupt Trigger
ITC_CPU_IRQ_MSK	0xA0009104	32	rw	0x00000000	CPU Interrupt Mask
ITC_CPU_IRQ_CLR	0xA0009108	32	mixed	0x00000000	CPU Interrupt Clear
ITC_CPU_IRQ_STA	0xA000910C	32	mixed	0x00000000	CPU Interrupt Status
AXI_BW	0xA0009204	32	rw	0x00000000	AXI Bandwidth Measurement Window
AXI_ADDR_OFFSET_IP	0xA0009208	32	rw	0x00000000	Video Data Address Offset

Table 3-10: **Encoder Registers**

Register	Address	Width	Type	Reset Value	Description
AXI_RBW0	0xA0009210	32	ro	0x00000000	AXI Read Bandwidth Status 0
AXI_RBW1	0xA0009214	32	ro	0x00000000	AXI Read Bandwidth Status 1
AXI_WBW0	0xA0009218	32	ro	0x00000000	AXI Write Bandwidth Status 0
AXI_WBW1	0xA000921C	32	ro	0x00000000	AXI Write Bandwidth Status 1
AXI_RBL0	0xA0009220	32	rw	0x00000000	AXI Read Bandwidth Limiter 0
AXI_RBL1	0xA0009224	32	rw	0x00000000	AXI Read Bandwidth Limiter 1

Decoder Block Overview

Introduction

The Decoder block is designed to process video streams using the H.265 (HEVC) and H.264 (AVC) standards. It provides a complete support for these standards, including support for 8-bit and 10-bit color depth, 4:0:0, 4:2:0, and 4:2:2 chroma formats, up to 4K UHD at 60 Hz performance.

The Decoder block efficiently performs video decompression.

The IP hardware has a direct access to the system data bus through a high-bandwidth master interface to transfer video data to and from an external memory.

The IP control software is partitioned into two layers. The VCU Control Software runs on the APU while the MCU firmware runs on an MCU, which is embedded in the hardware IP. The APU communicates with the embedded MCU through a slave interface, also connected to the system bus. The IP hardware is controlled by the embedded MCU using a register map to set decoding parameters through an internal peripheral bus.

Features

Table 4-1 describes the Decoder block features.

Table 4-1: **VCU Features**

Video Coding Feature	H.264	H.265
Performance		
Profiles	Baseline (except FMO/ASO/RS), Constrained Baseline, Main, High, High 10, High 4:2:2	Main, Main Intra, Main 10, Main 10 Intra, Main 4:2:2 10, Main 4:2:2 10 Intra
Levels	up to 5.2(2)	up to 5.1 High Tier(1)

Table 4-1: **VCU Features *(Cont'd)***

Video Coding Feature	H.264	H.265
Performance at 667 MHz Four streams at 1920x1080p at 60 Hz Two streams at 3840x2160p at 30 Hz One stream at 3840x2160p at 60 Hz One stream at 7680x4320p at 15 Hz	Supported	Supported
Configurable resolution: picture width and height multiple of 8, maximum width or height: 8192, with maximum picture size of 33.5 MP	Supported • minimum size: 80x96 • maximum width: 8192 • maximum height: 8192	Supported • minimum size: 80x96 • maximum width: 8184 (limited to 4096 in level 4/4.1 or when WPP is enabled) • maximum height: 8192
Configurable frame rate	Supported	Supported
Coding Tools		
Sample bit depth: 8 bpc, 10 bpc	Supported	Supported
Chroma format: YCbCr 4:2:0, YCbCr 4:2:2, Y-only (monochrome)	Supported	Supported
Progressive format only	Supported	Supported

Notes:

1. Support of 8K15 uses a subset of Level 6: maximum luma picture size up to 225 samples, other constraints of Level 5.1 apply (e.g. maximum of 200 slices and 11×10 tiles), WPP is not supported.
2. Support of 8K15 uses a subset of Level 6: maximum luma picture size up to 225 samples, other constraints of Level 5.2 apply, maximum slice size of 65,535 macroblocks so a minimum of two balanced slices must be used above 4K size.

Table 4-2 describes the VCU Decoder block maximum supported bit rates.

Table 4-2: **VCU Decoder Maximum Supported Bit Rates**

Standard	Level	Profile	Max Bit Rate
H.264	4.2 (1080p60)	Baseline, Main	50
		High	62.5
		High 10	150
		High 4:2:2	200
	5.2 (2160p60)	Baseline, Main	240
		High	300
		High 10	720
		High 4:2:2	960 (CAVLC) 720 (CABAC)

Table 4-2: **VCU Decoder Maximum Supported Bit Rates *(Cont'd)***

Standard	Level	Profile	Max Bit Rate
H.264	4.1 (1080p60) High Tier	Main, Main 10	50
		Main 4:2:2 10	84
		Main 4:2:2 10 Intra	167
	5.1 (2160p60) High Tier	Main, Main 10	160
		Main 4:2:2 10	267
		Main 4:2:2 10 Intra	533

Functional Description

Figure 4-1 shows the block diagram of the Decoder block.

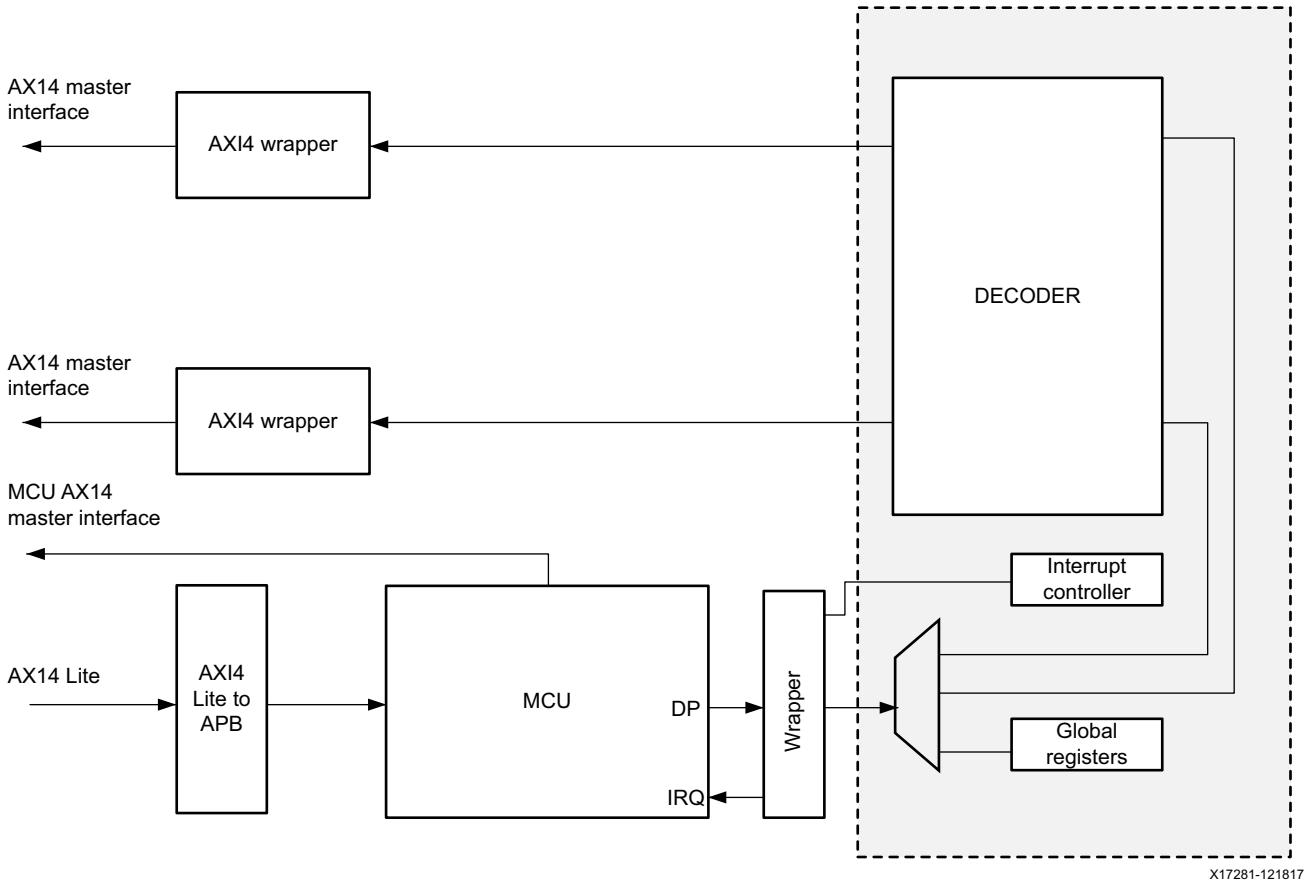


Figure 4-1: **Detailed Architecture of the Decoder Block**

The Decoder block includes the H.265/H.264 decompression engine, control registers, and an interrupt controller block.

The Decoder block is controlled by an MCU subsystem.

A 32-bit AXI-Lite slave interface is used by the system CPU to control the MCU to configure decoder parameters, start processing of video frames and to get status and results.

Two 128-bit AXI-4 master interfaces are used to fetch video input data and store video output data from/to the system memory.

An AXI-4 master interface is used to fetch the MCU software and performs load/store operation on additional MCU data.

Interfaces and Ports

Table 4-3 shows the Decoder block AXI-4 master interface ports.

Table 4-3: **Decoder Ports**

Name	Width	Dir	Description
vcu_pl_dec_araddr0/1	44	Output	AXI-4 ARADDR signal
vcu_pl_dec_arburst0/1	2	Output	AXI-4 ARBURST signal
vcu_pl_dec_arid0/1	4	Output	AXI-4 ARID signal
vcu_pl_dec_arlen0/1	8	Output	AXI-4 ARLEN signal
pl_vcu_dec_arready0/1	1	Input	AXI-4 ARREADY signal
vcu_pl_dec_arsize0/1	3	Output	AXI-4 ARSIZE signal
vcu_pl_dec_arvalid0/1	1	Output	AXI-4 ARVALID signal
vcu_pl_dec_awaddr0/1	44	Output	AXI-4 AWADDR signal
vcu_pl_dec_awburst0/1	2	Output	AXI-4 AWBURST signal
vcu_pl_dec_awid0/1	4	Output	AXI-4 AWID signal
vcu_pl_dec_awlen0/1	8	Output	AXI-4 AWLEN signal
pl_vcu_dec_awready0/1	1	Input	AXI-4 AWREADY signal
vcu_pl_dec_awsize0/1	3	Output	AXI-4 AWSIZE signal
vcu_pl_dec_awvalid0/1	1	Output	AXI-4 AWVALID signal
pl_vcu_dec_bresp0/1	2	Input	AXI-4 BRESP signal
vcu_pl_dec_bready0/1	1	Output	AXI-4 BREADY signal
pl_vcu_dec_bvalid0/1	1	Input	AXI-4 BVALID signal
pl_vcu_dec_bid0/1	4	Input	AXI-4 BID signal
pl_vcu_dec_rdata0/1	128	Input	AXI-4 RDATA signal
pl_vcu_dec_rid0/1	4	Input	AXI-4 RID signal
pl_vcu_dec_rlast0/1	1	Input	AXI-4 RLAST signal
vcu_pl_dec_rready0/1	1	Output	AXI-4 RREADY signal
pl_vcu_dec_rresp0/1	2	Input	AXI-4 RRESP signal
pl_vcu_dec_rvalid0/1	1	Input	AXI-4 RVALID signal

Table 4-3: **Decoder Ports *(Cont'd)***

Name	Width	Dir	Description
vcu_pl_dec_wdata0/1	128	Output	AXI-4 WDATA signal
vcu_pl_dec_wlast0/1	1	Output	AXI-4 WLAST signal
pl_vcu_dec_wready0/1	1	Input	AXI-4 WREADY signal
vcu_pl_dec_wvalid0/1	1	Output	AXI-4 WVALID signal
vcu_pl_dec_awprot0/1	1	Output	AXI-4 AWPROT signal, controlled from System Level Control Register (SLCR)
vcu_pl_dec_arprot0/1	1	Output	AXI-4 ARPROT signal, controlled from SLCR
vcu_pl_dec_awqos0/1	4	Output	AXI-4 AWQOS signal, controlled from SLCR
vcu_pl_dec_arqos0/1	4	Output	AXI-4 ARQOS signal, controlled from SLCR
vcu_pl_dec_awcache0/1	4	Output	AXI-4 AWCACHE signal, controlled from SLCR
vcu_pl_dec_arcache0/1	4	Output	AXI-4 ARCACHE signal, controlled from SLCR

Notes:

1. The MCU AXI interface and AXI Lite interface from VCU is common between encoder and decoder blocks. Refer to Table 3-3 for signal descriptions.

Clocking

Refer to Chapter 6, Clocking and Resets for more information on clocking.

Reset

Refer to Chapter 6, Clocking and Resets for more information on resets.

Data Path

The master interface inputs several types of video data from external memory:

- Bitstream
- Reference frame pixels
- Co-located picture motion vectors
- Headers and residual data

The master interface outputs:

- Decoded frame pixels
- Headers and residual data
- Decoded frame motion vectors, when the picture is later used as co-located picture

Control Path

The VCU slave interface is accessed once per frame by the APU, which sends a frame-level command to the IP. This interface therefore does not require a fast data path. An interrupt is generated on conclusion of each frame. These commands are processed by the embedded MCU, which generates tile and slice-level commands to the Decoder block hardware.

Memory Footprint Requirement

The Decoder block memory footprint depends on the decoding parameters.

The buffer size depends on the following:

- Video resolution
- Chroma sub-sampling
- Color depth
- Coding standard – H.264/H.265

The number of buffers depend on coding standard.

Table 4-4 shows the worst-case memory footprint required for different buffer sizes.

Table 4-4: **Decoder Block Memory Footprint**

Example with 2 B-frames	720p			1080p			2160p		
	Buffers	Per Buffer	Total	Buffers	Per Buffer	Total	Buffers	Per Buffer	Total
Input Bitstream buffer	2	1.4 MB	3 MB	2	3.0 MB	7 MB	2	3.0 MB	7 MB
Circular Bitstream buffer	1	7.2 MB	8 MB	1	15.1 MB	16 MB	1	15.1 MB	16 MB
Reference frames	18	1.3 MB	25 MB	12	3.2 MB	39 MB	22	3.2 MB	71 MB
Reconstructed frame	1	1.3 MB	2 MB	1	3.2 MB	4 MB	1	3.2 MB	4 MB
Intermediate buffers	5	3.8 MB	19 MB	5	8.6 MB	43 MB	5	8.6 MB	43 MB
Motion vector buffer	18	0.1 MB	1 MB	12	0.1 MB	2 MB	22	0.1 MB	3 MB
Slice parameters	5	1.3 MB	7 MB	5	1.3 MB	7 MB	5	1.3 MB	7 MB
Other buffers	1	1.1 MB	2 MB	1	1.1 MB	2 MB	1	1.1 MB	2 MB
Total			**67 MB**			**120 MB**			**153 MB**

Memory Bandwidth

The Decoder memory bandwidth depends on frame rate, resolution, color depth, chroma format and Decoder profile. The LogiCORE™ IP provides an estimate of Decoder bandwidth based on the video parameters selected in the GUI.

Memory Format

The decoded picture buffer contains the decoded pixels. It contains two parts: luminance pixels (luma) followed by chrominance pixels (chroma). Luma pixels are stored in pixel raster scan order. Chroma pixels are stored in U/V-interleaved pixel raster scan order, hence the chroma part is half the size of the luma part when using a 4:2:0 format and the same size as the luma part when using a 4:2:2 format. The decoded picture buffer must be one contiguous memory region.

The Xilinx DisplayPort implementation has 256-byte pitch requirement for frame buffer so the Decoder output is aligned to 256-byte pitch.

Two packing formats are supported in external memory: 8 bits per component or 10 bits per component, shown in Table 4-5 and Table 4-6, respectively. The 8-bit format can only be used for an 8-bit component depth and the 10-bit format can only be used for a 10-bit component depth. Table 4-5 and Table 4-6 show the raster scan format supported by the decoder block for 8-bit and 10-bit color depth.

Table 4-5: **Source Frame Buffer Format with 8-bit Component Format**

255 248	247 240	...	31 24	23 16	15 8	7 0
$Y_{x+31,y}$		...	$Y_{x+3,y}$	$Y_{x+2,y}$	$Y_{x+1,y}$	$Y_{x,y}$
...(all luma in pixel raster scan order)...						
255 248	247 240	...	31 24	23 16	15 8	7 0
$V_{x+15,y}$	$U_{x+15,y}$	...	$V_{x+1,y}$	$U_{x+1,y}$	$V_{x,y}$	$U_{x,y}$
...(all interleaved chroma in pixel raster scan order)...						

Table 4-6: **Source Frame Buffer Format with 10-bit Component Format**

255	254	253 244							31	30	29 20	19 10	9 0
0	0	$V_{x+23,y}$							0	0	$Y_{x+2,y}$	$Y_{x+1,y}$	$Y_{x,y}$
...(all luma in pixel raster scan order)...													
255	254	253 244	...	63	62	61 52	51 42	41 32	31	30	29 20	19 10	9 0
0	0	$V_{x+11,y}$	...	0	0	$V_{x+2,y}$	$U_{x+2,y}$	$V_{x+1,y}$	0	0	$U_{x+1,y}$	$V_{x,y}$	$U_{x,y}$
...(all interleaved chroma in pixel raster scan order)...													

The frame buffer width (pitch) may be larger than the frame width so that there are (pitch - width) ignored values between consecutive pixel lines.

Decoder Block Register Overview

Table 4-7 lists the Decoder block registers.

Table 4-7: **Decoder Registers**

Register Name	Address	Width	Type	Reset Value	Description
MCU_RESET	0xA0029000	32	mixed	0x00000000	MCU Subsystem Reset
MCU_RESET_MODE	0xA0029004	32	mixed	0x00000001	MCU Reset Mode
MCU_STA	0xA0029008	32	mixed	0x00000000	MCU Status
MCU_WAKEUP	0xA002900C	32	mixed	0x00000000	MCU Wake-up
MCU_ADDR_OFFSET_IC0	0xA0029010	32	rw	0x00000000	MCU Instruction Cache Address Offset 0
MCU_ADDR_OFFSET_IC1	0xA0029014	32	rw	0x00000000	MCU Instruction Cache Address Offset 1
MCU_ADDR_OFFSET_DC0	0xA0029018	32	rw	0x00000000	MCU Data Cache Address Offset 0
MCU_ADDR_OFFSET_DC1	0xA002901C	32	rw	0x00000000	MCU Data Cache Address Offset 1
ITC_MCU_IRQ	0xA0029100	32	mixed	0x00000000	MCU Interrupt Trigger
ITC_CPU_IRQ_MSK	0xA0029104	32	rw	0x00000000	CPU Interrupt Mask
ITC_CPU_IRQ_CLR	0xA0029108	32	mixed	0x00000000	CPU Interrupt Clear
ITC_CPU_IRQ_STA	0xA002910C	32	mixed	0x00000000	CPU Interrupt Status
AXI_BW	0xA0029204	32	rw	0x00000000	AXI Bandwidth Measurement Window
AXI_ADDR_OFFSET_IP	0xA0029208	32	rw	0x00000000	Video Data Address Offset
AXI_RBW0	0xA0029210	32	ro	0x00000000	AXI Read Bandwidth Status 0
AXI_RBW1	0xA0029214	32	ro	0x00000000	AXI Read Bandwidth Status 1
AXI_WBW0	0xA0029218	32	ro	0x00000000	AXI Write Bandwidth Status 0
AXI_WBW1	0xA002921C	32	ro	0x00000000	AXI Write Bandwidth Status 1
AXI_RBL0	0xA0029220	32	rw	0x00000000	AXI Read Bandwidth Limiter 0
AXI_RBL1	0xA0029224	32	rw	0x00000000	AXI Read Bandwidth Limiter 1

Microcontroller Unit Overview

Introduction

The Video Codec Unit (VCU) core includes two microcontroller unit (MCU) subsystems that run the MCU firmware and control the Encoder and Decoder blocks. The Encoder and Decoder blocks each have their own MCU to execute the firmware. The MCU has a 32-bit RISC architecture capable of executing pipelined transactions. The MCU has internal instruction, data cache, and AXI master interface to interface with the external memory.

Functional Description

Figure 5-1 shows the top-level interfaces and detailed architecture of the MCU.

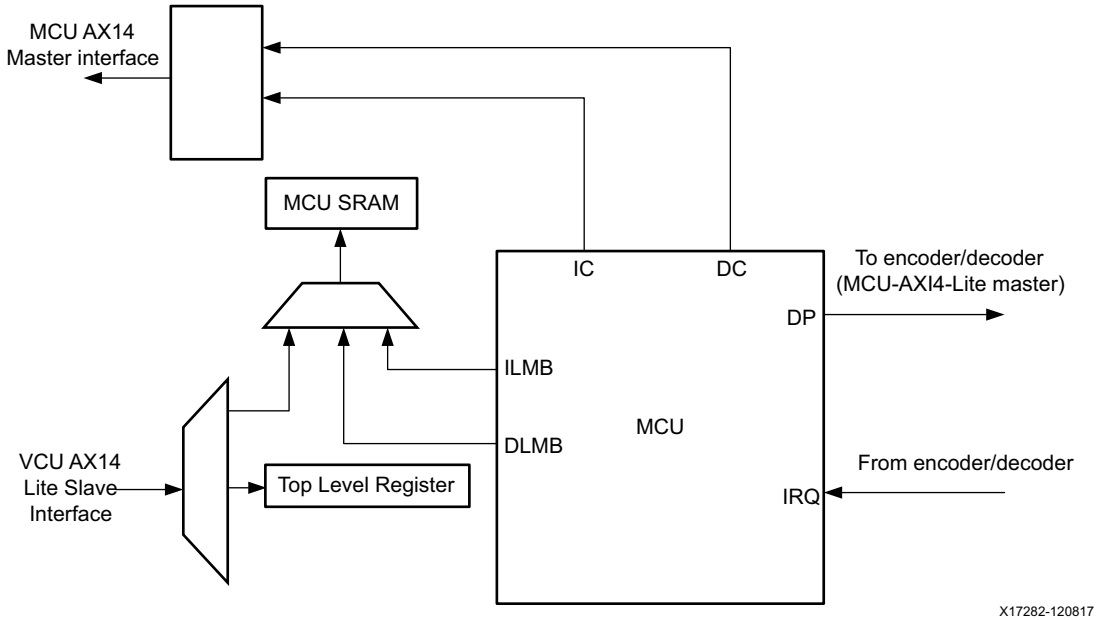


Figure 5-1: **MCU (Top-level)**

The MCU interfaces to peripherals using a 32-bit AXI4-Lite master interface. It has a local memory bus and AXI-4 32-bit instruction and data cache interfaces.

The MCU block has a 32 KB local memory for internal operations that is shared with the CPU for boot and mailbox communication. The MCU has a 32 KB instruction cache with 32-byte cache line width. It has a 4 KB data cache with 16-byte cache line width. The data cache has a write-through cache implementation.

Interfaces and Ports

Table 5-1 shows the AXI4 instruction and data cache interface ports of MCU.

Table 5-1: **MCU Ports**

Port	Size (bits)	Dir	Description
vcu_pl_mcu_m_axi_ic_dc_araddr	44	Output	AXI-4 read address
vcu_pl_mcu_m_axi_ic_dc_arburst	2	Output	AXI-4 read burst type
vcu_pl_mcu_m_axi_ic_dc_arcache	4	Output	AXI-4 ARCACHE value
vcu_pl_mcu_m_axi_ic_dc_arid	3	Output	AXI-4 read master ID
vcu_pl_mcu_m_axi_ic_dc_arlen	8	Output	AXI-4 read burst size
vcu_pl_mcu_m_axi_ic_dc_arlock	1	Output	AXI-4 ARLOCK signal
vcu_pl_mcu_m_axi_ic_dc_arprot	3	Output	AXI-4 ARPROT signal
vcu_pl_mcu_m_axi_ic_dc_arqos	4	Output	AXI-4 ARQOS signal
pl_vcu_mcu_m_axi_ic_dc_arready	1	Input	AXI-4 ARREADY signal
vcu_pl_mcu_m_axi_ic_dc_arsize	3	Output	AXI-4 ARSIZE signal
vcu_pl_mcu_m_axi_ic_dc_arvalid	1	Output	AXI-4 ARVALID signal
vcu_pl_mcu_m_axi_ic_dc_awaddr	44	Output	AXI-4 AWADDR signal
vcu_pl_mcu_m_axi_ic_dc_awburst	2	Output	AXI-4 AWBURST signal
vcu_pl_mcu_m_axi_ic_dc_awcache	4	Output	AXI-4 AWCACHE signal
vcu_pl_mcu_m_axi_ic_dc_awid	3	Output	AXI-4 AWID signal
vcu_pl_mcu_m_axi_ic_dc_awlen	8	Output	AXI-4 AWLEN signal
vcu_pl_mcu_m_axi_ic_dc_awlock	1	Output	AXI-4 AWLOCK signal
vcu_pl_mcu_m_axi_ic_dc_awprot	3	Output	AXI-4 AWPROT signal
vcu_pl_mcu_m_axi_ic_dc_awqos	4	Output	AXI-4 AWQOS signal
pl_vcu_mcu_m_axi_ic_dc_awready	1	Input	AXI-4 AWREADY signal
vcu_pl_mcu_m_axi_ic_dc_awsize	3	Output	AXI-4 AWSIZE signal
vcu_pl_mcu_m_axi_ic_dc_awvalid	1	Output	AXI-4 AWVALID signal
pl_vcu_mcu_m_axi_ic_dc_bid	3	Input	AXI-4 BID signal

Table 5-1: **MCU Ports *(Cont'd)***

Port	Size (bits)	Dir	Description
vcu_pl_mcu_m_axi_ic_dc_bready	1	Output	AXI-4 BREADY signal
pl_vcu_mcu_m_axi_ic_dc_bresp	2	Input	AXI-4 BRESP signal
pl_vcu_mcu_m_axi_ic_dc_bvalid	1	Input	AXI-4 BVALID signal
pl_vcu_mcu_m_axi_ic_dc_rdata	32	Input	AXI-4 RDATA signal
pl_vcu_mcu_m_axi_ic_dc_rid	3	Input	AXI-4 RID signal
pl_vcu_mcu_m_axi_ic_dc_rlast	1	Input	AXI-4 RLAST signal
vcu_pl_mcu_m_axi_ic_dc_rready	1	Output	AXI-4 RREADY signal
pl_vcu_mcu_m_axi_ic_dc_rresp	2	Input	AXI-4 RRESP signal
pl_vcu_mcu_m_axi_ic_dc_rvalid	1	Input	AXI-4 RVALID signal
vcu_pl_mcu_m_axi_ic_dc_wdata	32	Output	AXI-4 WDATA signal
vcu_pl_mcu_m_axi_ic_dc_wlast	1	Output	AXI-4 WLAST signal
pl_vcu_mcu_m_axi_ic_dc_wready	1	Input	AXI-4 WREADY signal
vcu_pl_mcu_m_axi_ic_dc_wstrb	4	Output	AXI-4 WSTRB signal
vcu_pl_mcu_m_axi_ic_dc_wvalid	1	Output	AXI-4 WVALID signal

Table 5-2 summarizes the AXI4-Lite slave interface ports of the MCU subsystem.

Table 5-2: **AXI4-Lite Slave Ports**

Port	Width	Direction	Description
pl_vcu_awaddr_axi_lite_apb	20	Input	AXI-4 AWADDR signal
pl_vcu_awprot_axi_lite_apb	3	Input	AXI-4 AWPROT signal
pl_vcu_awvalid_axi_lite_apb	1	Input	AXI-4 AWVALID signal
vcu_pl_awready_axi_lite_apb	1	Output	AXI-4 AWREADY signal
pl_vcu_wdata_axi_lite_apb	32	Input	AXI-4 WDATA signal
pl_vcu_wstrb_axi_lite_apb	4	Input	AXI-4 WSTRB signal
pl_vcu_wvalid_axi_lite_apb	1	Input	AXI-4 WVALID signal
vcu_pl_wready_axi_lite_apb	1	Output	AXI-4 WREADY signal
vcu_pl_bresp_axi_lite_apb	2	Output	AXI-4 BRESP signal
vcu_pl_bvalid_axi_lite_apb	1	Output	AXI-4 BVALID signal
pl_vcu_bready_axi_lite_apb	1	Input	AXI-4 BREADY signal
pl_vcu_araddr_axi_lite_apb	20	Input	AXI-4 ARADDR signal
pl_vcu_arprot_axi_lite_apb	3	Input	AXI-4 ARPROT signal
pl_vcu_arvalid_axi_lite_apb	1	Input	AXI-4 ARVALID signal
vcu_pl_arready_axi_lite_apb	1	Output	AXI-4 ARREADY signal
vcu_pl_rdata_axi_lite_apb	32	Output	AXI-4 RDATA signal

Table 5-2: **AXI4-Lite Slave Ports *(Cont'd)***

Port	Width	Direction	Description
vcu_pl_rresp_axi_lite_apb	2	Output	AXI-4 RRESP signal
vcu_pl_rvalid_axi_lite_apb	1	Output	AXI-4 RVALID signal
pl_vcu_rready_axi_lite_apb	1	Input	AXI-4 RREADY signal

Control Flow

The MCU is kept in sleep mode after applying the reset until the firmware boot code is downloaded by the kernel device driver into the internal memory of the MCU. After downloading the boot code and completing the MCU initialization sequence, the control software communicates with the MCU using a mailbox mechanism implemented in the internal SRAM memory of the MCU. The MCU sends an acknowledgment to the control software and performs the encoding/ decoding operation. When the requested operation is complete, the MCU communicates the status to the control software.

For more details about control software and MCU firmware, refer to Chapter 11, Application Software Development.

MCU Register Overview

Table 5-3 lists the MCU registers.

Table 5-3: **Encoder MCU Registers**

Register	Address	Width	Type	Reset Value	Description
MCU_RESET	0xA0009000	32	mixed	0x00000000	MCU Subsystem Reset
MCU_RESET_MODE	0xA0009004	32	mixed	0x00000001	MCU Reset Mode
MCU_STA	0xA0009008	32	mixed	0x00000000	MCU Status
MCU_WAKEUP	0xA000900C	32	mixed	0x00000000	MCU Wake-up
MCU_ADDR_OFFSET_IC0	0xA0009010	32	rw	0x00000000	MCU Instruction Cache Address Offset 0
MCU_ADDR_OFFSET_IC1	0xA0009014	32	rw	0x00000000	MCU Instruction Cache Address Offset 1
MCU_ADDR_OFFSET_DC0	0xA0009018	32	rw	0x00000000	MCU Data Cache Address Offset 0
MCU_ADDR_OFFSET_DC1	0xA000901C	32	rw	0x00000000	MCU Data Cache Address Offset 1

Clocking and Resets

Introduction

The Video Codec Unit (VCU) core supports one clocking topology:

- Internal PLL: An internal VCU phase locked loop (PLL) drives the high frequency core (667 MHz) and MCU (444 MHz) clocks based on an input reference clock from the programmable logic (PL). The internal PLL generates a clock for the Encoder and Decoder blocks.

Note: All AXI clocks are supplied with clocks from external PL sources. These clocks are asynchronous to core Encoder and Decoder block clocks. The Encoder and Decoder blocks handle asynchronous clocking in the AXI ports.

The VCU core is reset under the following conditions:

- Initially while PL is in powerup/configuration mode, the VCU core is held in reset.
- After PL is fully configured, a PL based reset signal can be used to reset the VCU for initialization and bring-up. Platform Management Unit (PMU) in the processing system (PS) can drive this reset signal to control VCU's reset state.
- During Partial Reconfiguration (PR), the VCU block is kept under reset if it is part of the dynamically reconfigurable module.

Functional Description

Clocking

The Decoder (VDEC) and Encoder (VENC) blocks work independently as separate units without any dependency on each other. Table 6-1 describes the clock domains in VCU core.

Table 6-1: **VCU Clock Domains**

Domain	Min freq [MHz]	Max freq [MHz]	Description
Core clock	N/A	667	Processing core, most of the logic and memories
MCU clock	N/A	444	Internal micro controllers
AXI Master Port clock	N/A	333	m_axi_enc_aclk, m_axi_dec_aclk, enc_buffer_clk, pl_vcu_axi_mcu_clk AXI master port for memory access, 128 bit, typically connected to PS AFI-FM (HP) port or to a soft memory controller in the PL.
AXI-Lite slave port clock	N/A	167	s_axi_lite_aclk, AXI-lite slave port (32-bit) for register programming

The Encoder and Decoder cores are fully synchronous and have single clock domain respectively. This clock drives all flops and RAMs in the Encoder and Decoder blocks.

Note: All AXI clocks are supplied with clocks from external PL sources. These clocks are asynchronous to core Encoder, Decoder, and MCU clocks). The VENC and VDEC cores are designed to handle asynchronous clocking in the AXI ports. The `m_axi_mcu_aclk` is asynchronous to all clocks used in VCU.

See The Encoder Buffer in Chapter 3 for more information.

Figure 6-1 shows the clock generation options inside VCU block. Note that the following blocks work on a single clock domain:

- `pll_ref_clk` is sourced externally to the device, typically by a programmable clock integrated circuit.
- Video encoder and decoder blocks work under the `VENC_core_clk` domain generated by the VCU PLL.
- MCU for encoder and decoder work under the `VENC_MCU_clk` domain generated by the VCU PLL.
- `m_axi_enc_aclk` is the AXI clock input from the PL for the 128-bit AXI master interfaces for the Encoder.
- `m_axi_dec_aclk` is the AXI clock input from the PL for the 128-bit AXI master interfaces for the Decoder.
- `s_axi_lite` is the AXI-Lite clock from the PL.
- `m_axi_mcu_aclk` is the MCU AXI master clock from the PL.

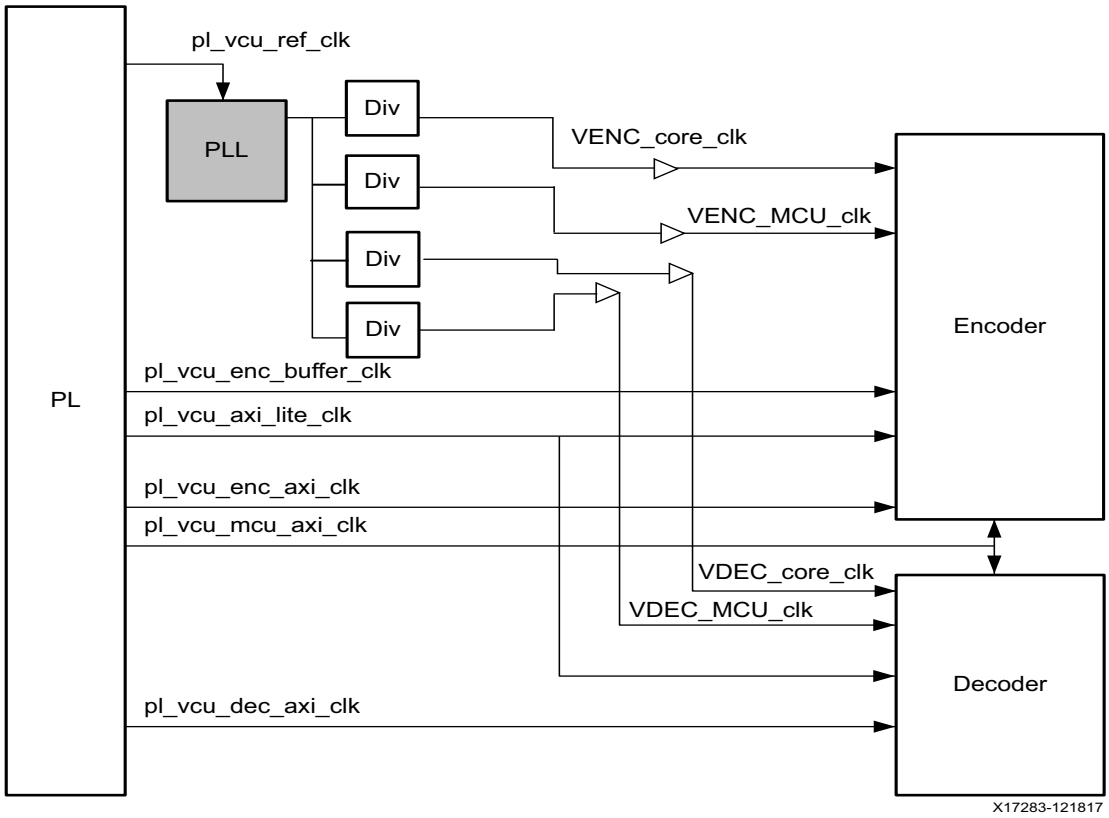


Figure 6-1: **Clock Generation Options**

The following clock frequency requirements must be met while providing clocks from PL:

- The AXI clock for encoder and decoder interface is limited to 333 MHz.
- The following ratio requirements need to be met:
  - $VENC\_core\_clk / m\_axi\_enc\_aclk \leq 2$

Refer to Chapter 5, Microcontroller Unit Overview for more information on the MCU.

The `VENC_core_clk` is generated based on VCU PLL.

The `VENC_MCU_clk` is generated based on VCU PLL.

The `VDEC_core_clk` is generated based on VCU PLL.

PLL Overview

The VCU core has a PLL for generating Encoder/Decoder block clocks. The PLL has a reference clock provided by a crystal oscillator on the board which is routed through a clock pin in the PL. The range of the PLL reference clock is 27 to 60 MHz. The PLL generates a high frequency clock that can be divided down to generate various output clock frequencies. The divided clock can be supplied to the Encoder block, Decoder block, and MCU (separate MCU for video encoder and decoder).

Generation of Primary Clock

The PLL has a Voltage Controlled Oscillator (VCO) block which generates an output clock based on the input reference clock. The output clock from VCO is generated based on a frequency multiplier value. The VCO's output clock is divided by an output divider to generate the final clock.

VCO Frequency and MF Value

The VCO operating frequency can be determined by using the following relationship:

$$f_{vco} = f_{refclk} \times M$$

and

$$f_{clkout} = f_{vco} / O$$

where *M* corresponds to the integer feedback divide value and *O* corresponds to the value of output divide. Note that the PLL does not support fractional divider values.

IMPORTANT: *Select the PLL feedback multiplier value based on the supported VCO frequency range (f_{vco}).*

Refer to *Zynq UltraScale+ MPSoC Data Sheet: DC and AC Switching Characteristics* [Ref 13] for more information on the operating range of f_{vco}.

IMPORTANT: *Select the output divider (O) based on the required core clock or MCU clock frequency.*

IMPORTANT: *Sharing VCU clock inputs with other IP can result in clock jitter that may degrade VCU performance or image quality.*

Reset Sequence

The state of VCU during PL power up and the initialization sequence for VCU are described as follows:

- The PL is not yet configured. In this condition VCU is held in reset.
- The VCU core is held in reset when the power supplies ramp up. A voltage detector present in the VCU core to PL interface keeps the core under reset while supplies ramp up.
- PL is fully configured. The PL is configured with AXI connectivity between a CPU in the PS or PL and the AXI slave port of the VCU core. The VCU core reset can be released so that the core is in a known state.

After VCU core reset is de-asserted, use the software to program the VCU PLL for generating the clocks for VCU core and MCU blocks. When programming the VCU PLL, follow the steps described in PLL Integer Divider Programming for programming PLL configuration parameters. The PLL lock status is indicated by VCU_SLCR.

Note: The VCU core clocks are available while reset is released. The PL should be configured before releasing the raw reset, which can be controlled by the PMU from outside of the VCU core.

Additional initialization is done by software through programming the VCU core registers after the PL is configured and core is in a reset release state.

PLL Integer Divider Programming

To operate the VCU PLL, configure the VCU_SLCR.VCU_PLL_CFG register using the values in Table 6-2. The following fields must be programmed.

- VCU_SLCR.VCU_PLL_CFG[LOCK_DLY]
- VCU_SLCR.VCU_PLL_CFG[LOCK_CNT]
- VCU_SLCR.VCU_PLL_CFG[LFHF]
- VCU_SLCR.VCU_PLL_CFG[CP]
- VCU_SLCR.VCU_PLL_CFG[RES]

The FBDIV value (or the PLL feedback multiplier value, M) depends on the output VCO frequency (fvco). You must program VCU_SRCR.VCU_PLL_CFG based on the calculated FBDIV values in Table 6-2.

Table 6-2: **PLL Programming for Integer Feedback Divider Values**

FBDIV	CP	RES	LFHF	LOCK_DLY	LOCK_CNT
25	3	10	3	63	1000
26	3	10	3	63	1000
27	4	6	3	63	1000
28	4	6	3	63	1000
29	4	6	3	63	1000
30	4	6	3	63	1000
31	6	1	3	63	1000
32	6	1	3	63	1000
33	4	10	3	63	1000
34	5	6	3	63	1000
35	5	6	3	63	1000
36	5	6	3	63	1000
37	5	6	3	63	1000
38	5	6	3	63	975

Table 6-2: **PLL Programming for Integer Feedback Divider Values *(Cont'd)***

FBDIV	CP	RES	LFHF	LOCK_DLY	LOCK_CNT
39	3	12	3	63	950
40	3	12	3	63	925
41	3	12	3	63	900
42	3	12	3	63	875
43	3	12	3	63	850
44	3	12	3	63	850
45	3	12	3	63	825
46	3	12	3	63	800
47	3	12	3	63	775
48	3	12	3	63	775
49	3	12	3	63	750
50	3	12	3	63	750
51	3	2	3	63	725
52	3	2	3	63	700
53	3	2	3	63	700
54	3	2	3	63	675
55	3	2	3	63	675
56	3	2	3	63	650
57	3	2	3	63	650
58	3	2	3	63	625
59	3	2	3	63	625
60	3	2	3	63	625
61	3	2	3	63	600
62	3	2	3	63	600
63	3	2	3	63	600
64	3	2	3	63	600
65	3	2	3	63	600
66	3	2	3	63	600
67	3	2	3	63	600
68	3	2	3	63	600
69	3	2	3	63	600
70	3	2	3	63	600
71	3	2	3	63	600
72	3	2	3	63	600
73	3	2	3	63	600

Table 6-2: **PLL Programming for Integer Feedback Divider Values *(Cont'd)***

FBDIV	CP	RES	LFHF	LOCK_DLY	LOCK_CNT
74	3	2	3	63	600
75	3	2	3	63	600
76	3	2	3	63	600
77	3	2	3	63	600
78	3	2	3	63	600
79	3	2	3	63	600
80	3	2	3	63	600
81	3	2	3	63	600
82	3	2	3	63	600
83	4	2	3	63	600
84	4	2	3	63	600
85	4	2	3	63	600
86	4	2	3	63	600
87	4	2	3	63	600
88	4	2	3	63	600
89	4	2	3	63	600
90	4	2	3	63	600
91	4	2	3	63	600
92	4	2	3	63	600
93	4	2	3	63	600
94	4	2	3	63	600
95	4	2	3	63	600
96	4	2	3	63	600
97	4	2	3	63	600
98	4	2	3	63	600
99	4	2	3	63	600
100	4	2	3	63	600
101	4	2	3	63	600
102	4	2	3	63	600
103	5	2	3	63	600
104	5	2	3	63	600
105	5	2	3	63	600
106	5	2	3	63	600
107	3	4	3	63	600
108	3	4	3	63	600

Table 6-2: **PLL Programming for Integer Feedback Divider Values *(Cont'd)***

FBDIV	CP	RES	LFHF	LOCK_DLY	LOCK_CNT
109	3	4	3	63	600
110	3	4	3	63	600
111	3	4	3	63	600
112	3	4	3	63	600
113	3	4	3	63	600
114	3	4	3	63	600
115	3	4	3	63	600
116	3	4	3	63	600
117	3	4	3	63	600
118	3	4	3	63	600
119	3	4	3	63	600
120	3	4	3	63	600
121	3	4	3	63	600
122	3	4	3	63	600
123	3	4	3	63	600
124	3	4	3	63	600
125	3	4	3	63	600

Reset

The VCU hard block can be held under reset under the following conditions:

- When external reset input `vcu_resetn` signal is asserted.
- During PL configuration.
- When the VCU to PL isolation is not removed.

The VCU reset signal must be asserted at least for two clock cycles of the VCU PLL reference clock (the slowest clock input to the VCU). The VCU registers can be accessed after the reset signal is de-asserted.

Note: If software resets the VCU block in the middle of a frame, use the software to clear the physical memory allocated for the VCU.

Note: The reset does not need to be asserted between changes to the VCU configuration during run-time via the VCU Control Software.

The software can program the `VCU_GASKET_INIT` register at offset 0x41074 in the `VCU_SLCR` to assert a reset pulse to the VCU block. Writing a 0 asserts reset, 1 de-asserts the reset pulse. Reset VCU using reset register using the following procedure.

1. Write 0 to VCU_GASKET_INIT[1].
2. Write 1 to VCU_GASKET_INIT[0].

The PLL in the VCU core can be reset through VCU_SLCR register which is accessible through AXI-Lite interface.

Each of the Encoder and Decoder blocks have register-based soft reset.

Clocking and Reset Registers

Table 6-3 lists the clocking and reset registers.

Table 6-3: **Clocking and Reset Registers**

Register	Address	Width	Type	Reset Value	Description
CRL_WPROT	0xA0040020	1	rw	0x00000000	CRL SLCR Write protection register
VCU_PLL_CTRL	0xA0040024	32	mixed	0x0000510F	PLL Basic Control
VCU_PLL_CFG	0xA0040028	32	mixed	0x00000000	Helper data
PLL_STATUS	0xA0040060	32	mixed	0x00000008	Status of the PLLs

Latency in the VCU Pipeline

The Video Codec Unit (VCU) is designed to support video streams at resolutions up to 3840×2160 pixels at 60 frames per second (4K UHD at 60Hz) with group of pictures (GOP) B-frames (hardest case). Latency is defined between frame boundaries and all GOP types are allowed. Some GOP types require display reordering buffers.

Glass-to-Glass Latency

As illustrated in Figure 7-1, glass-to-glass latency (L) is the sum of the following:

- Camera latency
- On-chip latency (L1)
  - Source frame buffer DMA latency
  - Encoder latency
  - Transmission bitstream buffer latency
- Network or storage latency
- On-chip latency (L2)
  - CPB/jitter buffer latency
  - Decoder latency
  - DPB latency
  - Display frame buffer DMA latency
- Display monitor latency

When B-frames are enabled, one frame of latency is incurred for each B-frame due to the usage of the reordering buffer. To optimize the CPB latency, a handshaking mechanism in PL is required between decoder and the display DMA. It is assumed that both capture side and display side works on a common VSYNC timing.

VSYNC timing can be asynchronous and a clock recovery mechanism is needed to synchronize source timing with sync.

With independent VSYNC timing and without clock recovery mechanism, it requires one additional frame latency to synchronize with the display devices.

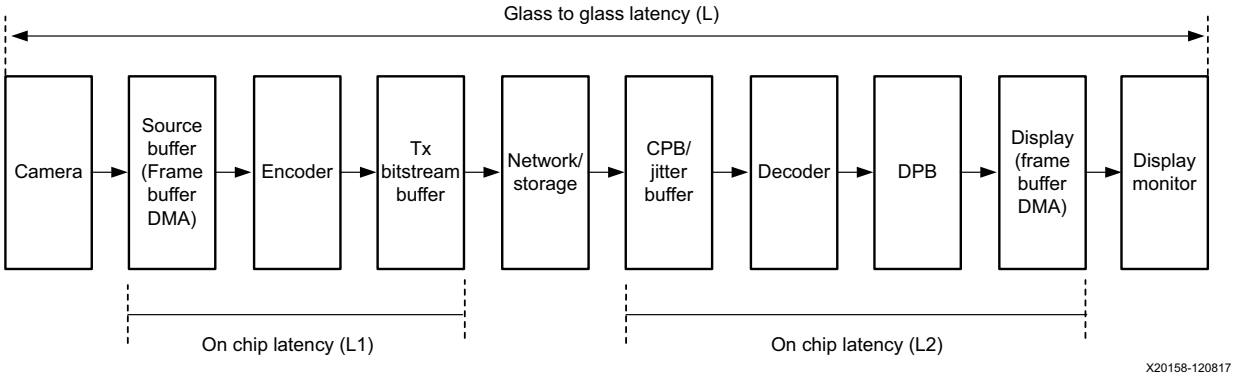


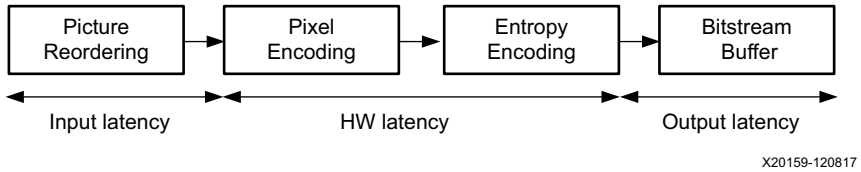
Figure 7-1: **Glass to Glass Latency**

Table 7-1 shows the latency for VCU pipeline stages.

Table 7-1: **VCU Latency**

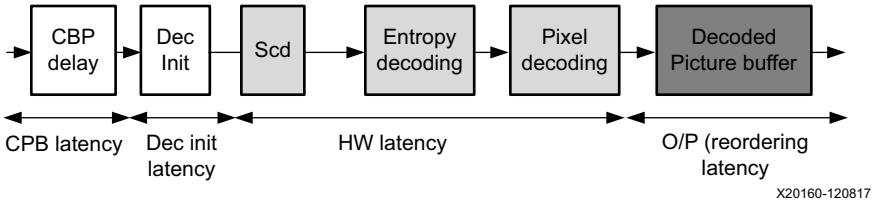
Use Case	Capture	Encode	Decode	Display
Latency	16.6 ms	16.6 ms	83.33 ms	16.6 ms

VCU Encoder Latency



The overall latency of the Encoder is the steady state latency, equal to the sum of the input latency, the hardware latency and the output latency. Bitstream buffer latency is application dependent. Picture reordering latency equals one frame duration per B-frame.

VCU Decoder Latency



The overall latency of the Decoder is the steady state latency, equal to the sum of the hardware latency and the output latency. Initialization latency is the sum of the CPB latency and the Dec Init latency.

AXI Performance Monitor

Overview

The AXI Performance Monitor (APM) is implemented inside the embedded Video Codec Unit (VCU). The VCU AXI Performance Monitor (VAPM) allows access to system level behavior in a non-invasive way and without burdening the design with additional soft IP.

The APM block is capable of measuring the number of read/write bytes and address based transactions within a measurement window on the AXI master bus from Encoder/Decoder blocks. The APM can additionally measure master ID based read and write latency within a measurement window. The APM supports cumulative latency value along with number of outstanding transfers being considered for latency measurement. The APM has the ability to interrupt the host processor when the status registers are ready to be read.

Functional Description

Figure 8-1 shows the VAPM.

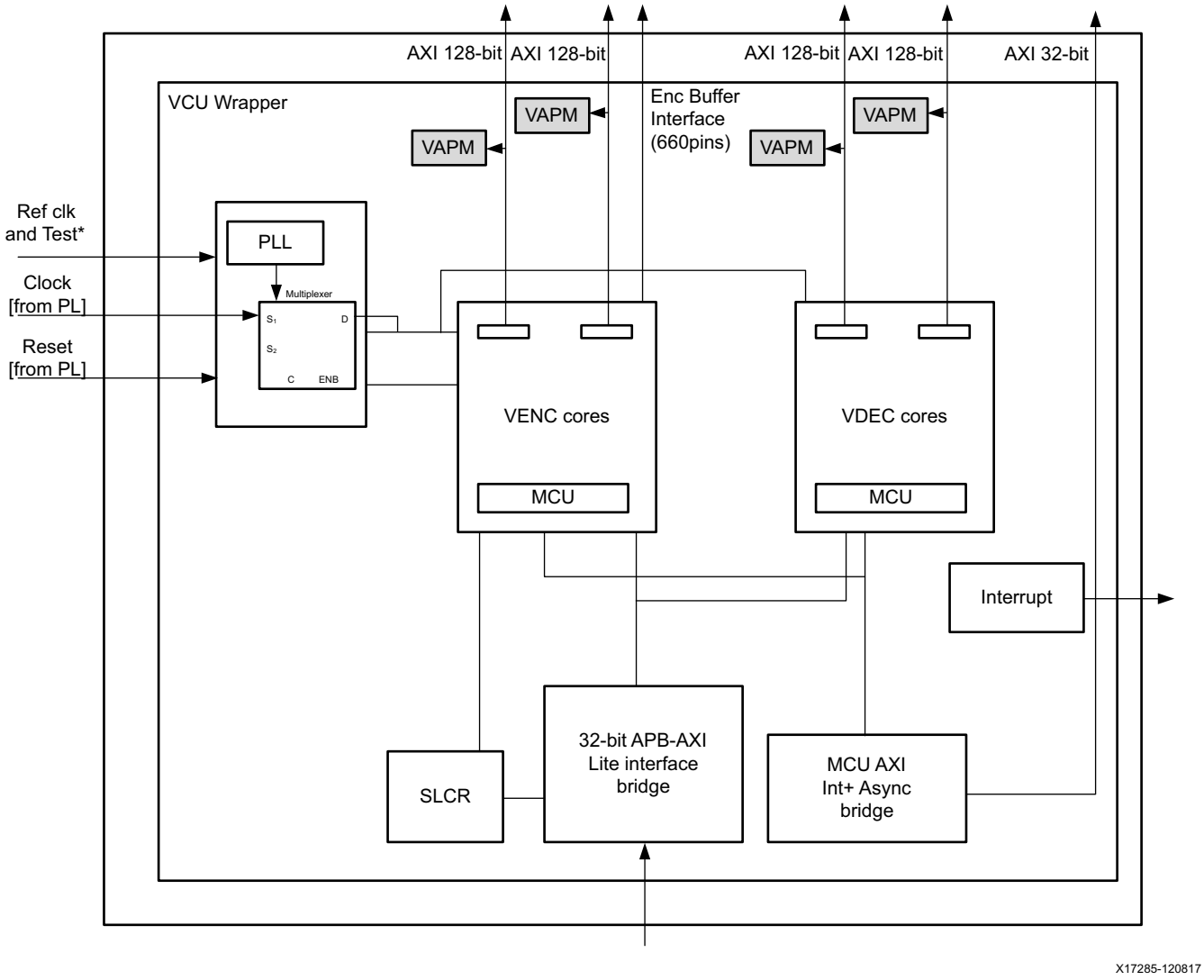


Figure 8-1: **VCU AXI Performance Monitor (VAPM)**

The following sections describe the different operating modes of VAPM.

Operating Timing Window Generation

The VAPM generates measurement parameters based on two user-selected operating modes.

Start/Stop Mode

In this mode, the measurement window is determined by the start/stop bit in `VCU_SLCR.APMn_TRIG[start_stop] (n=0,1,2,3)` bit. A measurement is triggered when start bit is set from 0 to 1 in this register and measurement is stopped when this bit is reset from 1 to 0.

Fixed Duration Timing Window

In this mode, a 32-bit counter is used to generate fixed length measurement window. When the counter reaches maximum value, it resets to a value specified in the `VCU_SLCR.APMn_TIMER (n=0,1,2,3)` register. The measurement is continued until the 32-bit counter reaches the value set in `APMn_TIMER` register and a capture pulse is generated to store the measured values in `VCU_SLCR` result registers.

The VAPM is capable of doing the following measurements:

- AXI Read and Write Transaction Measurement
- AXI Read and Write Byte Count Measurement
- AXI Transaction Latency Measurement

AXI Read and Write Transaction Measurement

Two 32-bit registers count number of read and write 128-bit AXI bus cycles transferred in a given timing window. The measured value is transferred to the `VCU_SLCR` result register when a capture pulse is generated based on start/stop mode or fixed duration timing window mode. To compute the number of bytes transferred, `VCU_SLCR` must be multiplied by 16.

AXI Read and Write Byte Count Measurement

Two 32-bit registers are implemented to count number of read and write bytes transferred in a given timing window. The register content has to be multiplied by 16 to know the actual byte count transferred across AXI 128-bit master interface. The measured value is transferred to the `VCU_SLCR` result register when a capture pulse is generated based on start/stop mode or fixed duration timing window mode.

AXI Transaction Latency Measurement

Read and write latency can be measured based on AXI master ID. Read latency is defined as AXI read address acknowledged to last read data cycle. Write latency is defined as AXI write address acknowledged to write response handshaking between master and slave. A 13-bit counter is implemented to measure the latency on read and write bus. The timer is used to timestamp an event. The difference in the timestamp between two events is used to calculate the latency.

Latency can be calculated on transaction ID basis. It is possible to select single ID or all IDs for latency calculation.

Table 8-1: **APM Registers**

Register	Address	Width	Type	Reset Value	Description
APM_InputT_GBL_CNTL	0×A0040090	32	mixed	0x00000001	This register controls APM timing window completion interrupt.
APM0_CFG	0xA0040100	32	mixed	0x00000002	APM0_CFG
APM0_TIMER	0xA0040104	32	rw	0x00000000	APM0_TIMER
APM0_TRG	0xA0040108	32	mixed	0x00000000	APM0_TRG
APM0_RESULT0	0xA004010C	32	ro	0x00000000	APM0_RESULT0
APM0_RESULT1	0xA0040110	32	ro	0x00000000	APM0_RESULT1
APM0_RESULT2	0xA0040114	32	ro	0x00000000	APM0_RESULT2
APM0_RESULT3	0xA0040118	32	ro	0x00000000	APM0_RESULT3
APM0_RESULT4	0xA004011C	32	mixed	0x00000000	APM0_RESULT4
APM0_RESULT5	0xA0040120	32	mixed	0x00000000	APM0_RESULT5
APM0_RESULT6	0xA0040124	32	mixed	0x00000000	APM0_RESULT6
APM0_RESULT7	0xA0040128	32	mixed	0x00000000	APM0_RESULT7
APM0_RESULT8	0xA004012C	32	mixed	0x00000000	APM0_RESULT8
APM0_RESULT9	0xA0040130	32	mixed	0x00000000	APM0_RESULT9
APM0_RESULT10	0xA0040134	32	mixed	0x00000000	APM0_RESULT10
APM0_RESULT11	0xA0040138	32	mixed	0x00000000	APM0_RESULT11
APM0_RESULT12	0xA004013C	32	mixed	0x00000000	APM0_RESULT12
APM0_RESULT13	0xA0040140	32	mixed	0x00000000	APM0_RESULT13
APM0_RESULT14	0xA0040144	32	mixed	0x00000000	APM0_RESULT14
APM0_RESULT15	0xA0040148	32	mixed	0x00000000	APM0_RESULT15
APM0_RESULT16	0xA004014C	32	mixed	0x00000000	APM0_RESULT16
APM0_RESULT17	0xA0040150	32	mixed	0x00000000	APM0_RESULT17
APM0_RESULT18	0xA0040154	32	mixed	0x00000000	APM0_RESULT18
APM0_RESULT19	0xA0040158	32	mixed	0x00000000	APM0_RESULT19
APM0_RESULT20	0xA004015C	32	mixed	0x1FFF0000	APM0_RESULT20
APM0_RESULT21	0xA0040160	32	mixed	0x1FFF0000	APM0_RESULT21
APM0_RESULT22	0xA0040164	32	mixed	0x1FFF0000	APM0_RESULT22
APM0_RESULT23	0xA0040168	32	mixed	0x1FFF0000	APM0_RESULT23
APM0_RESULT24	0xA004016C	32	mixed	0x00000000	APM0_RESULT24
APM1_CFG	0xA0040200	32	mixed	0x00000002	APM1_CFG
APM1_TIMER	0xA0040204	32	rw	0x00000000	APM1_TIMER
APM1_TRG	0xA0040208	32	mixed	0x00000000	APM1_TRG

Table 8-1: **APM Registers *(Cont'd)***

Register	Address	Width	Type	Reset Value	Description
APM1_RESULT0	0xA004020C	32	ro	0x00000000	APM1_RESULT0
APM1_RESULT1	0xA0040210	32	ro	0x00000000	APM1_RESULT1
APM1_RESULT2	0xA0040214	32	ro	0x00000000	APM1_RESULT2
APM1_RESULT3	0xA0040218	32	ro	0x00000000	APM1_RESULT3
APM1_RESULT4	0xA004021C	32	mixed	0x00000000	APM1_RESULT4
APM1_RESULT5	0xA0040220	32	mixed	0x00000000	APM1_RESULT5
APM1_RESULT6	0xA0040224	32	mixed	0x00000000	APM1_RESULT6
APM1_RESULT7	0xA0040228	32	mixed	0x00000000	APM1_RESULT7
APM1_RESULT8	0xA004022C	32	mixed	0x00000000	APM1_RESULT8
APM1_RESULT9	0xA0040230	32	mixed	0x00000000	APM1_RESULT9
APM1_RESULT10	0xA0040234	32	mixed	0x00000000	APM1_RESULT10
APM1_RESULT11	0xA0040238	32	mixed	0x00000000	APM1_RESULT11
APM1_RESULT12	0xA004023C	32	mixed	0x00000000	APM1_RESULT12
APM1_RESULT13	0xA0040240	32	mixed	0x00000000	APM1_RESULT13
APM1_RESULT14	0xA0040244	32	mixed	0x00000000	APM1_RESULT14
APM1_RESULT15	0xA0040248	32	mixed	0x00000000	APM1_RESULT15
APM1_RESULT16	0xA004024C	32	mixed	0x00000000	APM1_RESULT16
APM1_RESULT17	0xA0040250	32	mixed	0x00000000	APM1_RESULT17
APM1_RESULT18	0xA0040254	32	mixed	0x00000000	APM1_RESULT18
APM1_RESULT19	0xA0040258	32	mixed	0x00000000	APM1_RESULT19
APM1_RESULT20	0xA004025C	32	mixed	0x1FFF0000	APM1_RESULT20
APM1_RESULT21	0xA0040260	32	mixed	0x1FFF0000	APM1_RESULT21
APM1_RESULT22	0xA0040264	32	mixed	0x1FFF0000	APM1_RESULT22
APM1_RESULT23	0xA0040268	32	mixed	0x1FFF0000	APM1_RESULT23
APM1_RESULT24	0xA004026C	32	mixed	0x00000000	APM1_RESULT24
APM2_CFG	0xA0040300	32	mixed	0x00000002	APM2_CFG
APM2_TIMER	0xA0040304	32	rw	0x00000000	APM2_TIMER
APM2_TRG	0xA0040308	32	mixed	0x00000000	APM2_TRG
APM2_RESULT0	0xA004030C	32	ro	0x00000000	APM2_RESULT0
APM2_RESULT1	0xA0040310	32	ro	0x00000000	APM2_RESULT1
APM2_RESULT2	0xA0040314	32	ro	0x00000000	APM2_RESULT2
APM2_RESULT3	0xA0040318	32	ro	0x00000000	APM2_RESULT3
APM2_RESULT4	0xA004031C	32	mixed	0x00000000	APM2_RESULT4
APM2_RESULT5	0xA0040320	32	mixed	0x00000000	APM2_RESULT5
APM2_RESULT6	0xA0040324	32	mixed	0x00000000	APM2_RESULT6

Table 8-1: **APM Registers *(Cont'd)***

Register	Address	Width	Type	Reset Value	Description
APM2_RESULT7	0xA0040328	32	mixed	0x00000000	APM2_RESULT7
APM2_RESULT8	0xA004032C	32	mixed	0x00000000	APM2_RESULT8
APM2_RESULT9	0xA0040330	32	mixed	0x00000000	APM2_RESULT9
APM2_RESULT10	0xA0040334	32	mixed	0x00000000	APM2_RESULT10
APM2_RESULT11	0xA0040338	32	mixed	0x00000000	APM2_RESULT11
APM2_RESULT12	0xA004033C	32	mixed	0x00000000	APM2_RESULT12
APM2_RESULT13	0xA0040340	32	mixed	0x00000000	APM2_RESULT13
APM2_RESULT14	0xA0040344	32	mixed	0x00000000	APM2_RESULT14
APM2_RESULT15	0xA0040348	32	mixed	0x00000000	APM2_RESULT15
APM2_RESULT16	0xA004034C	32	mixed	0x00000000	APM2_RESULT16
APM2_RESULT17	0xA0040350	32	mixed	0x00000000	APM2_RESULT17
APM2_RESULT18	0xA0040354	32	mixed	0x00000000	APM2_RESULT18
APM2_RESULT19	0xA0040358	32	mixed	0x00000000	APM2_RESULT19
APM2_RESULT20	0xA004035C	32	mixed	0x1FFF0000	APM2_RESULT20
APM2_RESULT21	0xA0040360	32	mixed	0x1FFF0000	APM2_RESULT21
APM2_RESULT22	0xA0040364	32	mixed	0x1FFF0000	APM2_RESULT22
APM2_RESULT23	0xA0040368	32	mixed	0x1FFF0000	APM2_RESULT23
APM2_RESULT24	0xA004036C	32	mixed	0x00000000	APM2_RESULT24
APM3_CFG	0xA0040400	32	mixed	0x00000002	APM3_CFG
APM3_TIMER	0xA0040404	32	rw	0x00000000	APM3_TIMER
APM3_TRG	0xA0040408	32	mixed	0x00000000	APM3_TRG
APM3_RESULT0	0xA004040C	32	ro	0x00000000	APM3_RESULT0
APM3_RESULT1	0xA0040410	32	ro	0x00000000	APM3_RESULT1
APM3_RESULT2	0xA0040414	32	ro	0x00000000	APM3_RESULT2
APM3_RESULT3	0xA0040418	32	ro	0x00000000	APM3_RESULT3
APM3_RESULT4	0xA004041C	32	mixed	0x00000000	APM3_RESULT4
APM3_RESULT5	0xA0040420	32	mixed	0x00000000	APM3_RESULT5
APM3_RESULT6	0xA0040424	32	mixed	0x00000000	APM3_RESULT6
APM3_RESULT7	0xA0040428	32	mixed	0x00000000	APM3_RESULT7
APM3_RESULT8	0xA004042C	32	mixed	0x00000000	APM3_RESULT8
APM3_RESULT9	0xA0040430	32	mixed	0x00000000	APM3_RESULT9
APM3_RESULT10	0xA0040434	32	mixed	0x00000000	APM3_RESULT10
APM3_RESULT11	0xA0040438	32	mixed	0x00000000	APM3_RESULT11
APM3_RESULT12	0xA004043C	32	mixed	0x00000000	APM3_RESULT12
APM3_RESULT13	0xA0040440	32	mixed	0x00000000	APM3_RESULT13

Table 8-1: **APM Registers *(Cont'd)***

Register	Address	Width	Type	Reset Value	Description
APM3_RESULT14	0xA0040444	32	mixed	0x00000000	APM3_RESULT14
APM3_RESULT15	0xA0040448	32	mixed	0x00000000	APM3_RESULT15
APM3_RESULT16	0xA004044C	32	mixed	0x00000000	APM3_RESULT16
APM3_RESULT17	0xA0040450	32	mixed	0x00000000	APM3_RESULT17
APM3_RESULT18	0xA0040454	32	mixed	0x00000000	APM3_RESULT18
APM3_RESULT19	0xA0040458	32	mixed	0x00000000	APM3_RESULT19
APM3_RESULT20	0xA004045C	32	mixed	0x1FFF0000	APM3_RESULT20
APM3_RESULT21	0xA0040460	32	mixed	0x1FFF0000	APM3_RESULT21
APM3_RESULT22	0xA0040464	32	mixed	0x1FFF0000	APM3_RESULT22
APM3_RESULT23	0xA0040468	32	mixed	0x1FFF0000	APM3_RESULT23
APM3_RESULT24	0xA004046C	32	mixed	0x00000000	APM3_RESULT24

Designing with the Core

This chapter includes guidelines and additional information to facilitate designing with the core.

General Design Guidelines

The Video Codec Unit (VCU) core is a dedicated hardware block in the programming logic (PL). All interfaces are connected through AXI interconnect blocks in the PL. The VCU core is AXI-4 compliant on its AXI master interfaces. It can be connected to the `S_AXI_HP_FPD` (or `S_AXI_LPD`, `S_AXI_HPC_FPD`) ports of the PS or AXI compliant interface of the PL memory controller. There are no direct (hardwired) connections from the VCU to the processing system (PS).

The register programming interface of the VCU core connects to PS General Purpose (GP) ports. The clock can be used from PL or through an internal PLL inside the VCU core.

The core is delivered through the Vivado® Design Suite with an example design built around the core, allowing the functionality of the core to be demonstrated in hardware, if placed on a suitable board. For details about the Vivado Design Suite example design, see Chapter 10, Example Design.

Interrupts

There is one interrupt line from the VCU core to the PS (`vcu_host_interrupt`). This interrupt has to be connected to either `PL-PS-IRQ0[7:0]` or `PL-PS-IRQ1[7:0]`. If there are other interrupts in the design, the interrupt has to be concatenated along with the other interrupts and then connected to the PS.

Design Flow Steps

This chapter describes customizing and generating the core, constraining the core, and the simulation, synthesis and implementation steps that are specific to this IP core. More detailed information about the standard Vivado® design flows and the IP integrator can be found in the following Vivado Design Suite user guides:

- *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994) [Ref 1]
- *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 2]
- *Vivado Design Suite User Guide: Getting Started* (UG910) [Ref 3]
- *Vivado Design Suite User Guide: Logic Simulation* (UG900) [Ref 4]

Vivado Integrated Design Environment

You can customize the IP for use in your design by specifying values for the various parameters associated with the IP core using the following steps:

1. Select the IP from the IP catalog.
2. Double-click the selected IP or select the Customize IP command from the toolbar or right-click menu.

For details, see the *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 2] and the *Vivado Design Suite User Guide: Getting Started* (UG910) [Ref 3].

Note: Figures in this chapter are illustrations of the Vivado IDE. The layout depicted here might vary from the current version.

Note: The LogiCORE™ IP is available through the Vivado IP Integrator (IPI) flow only. It is not available as a standalone IP.

The Basic configuration tab, shown in Figure 10-1, allows for the selection of video parameters used to calculate the Encoder buffer size and total dynamic power used by Encoder or Decoder blocks.

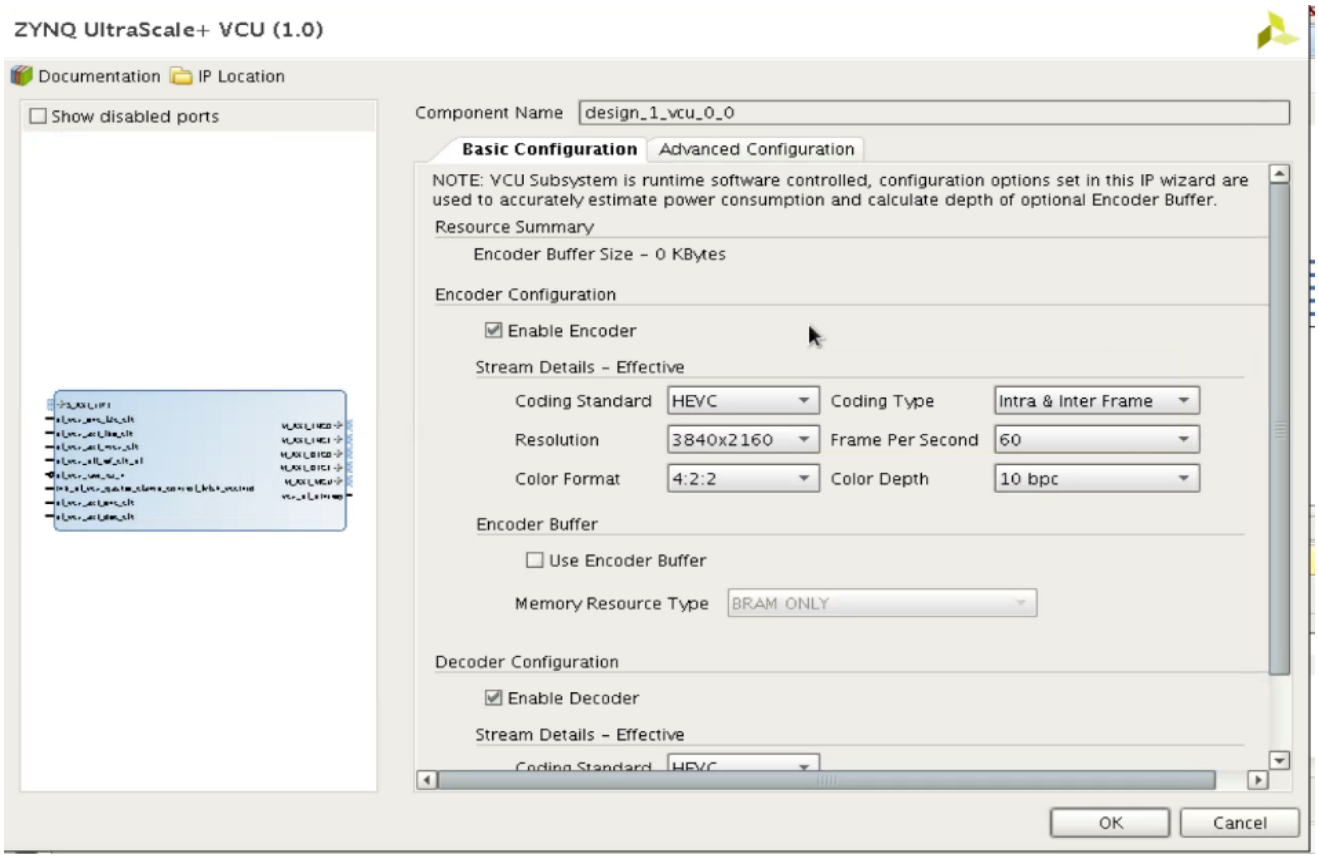


Figure 10-1: **Basic Configuration Tab**

Note:

- The resolution field represents the aggregate resolution used for multiple streams. (For example, select 3840x2160 to use four 1920x1080 streams.)
- You must input the upper range of video parameters (for color format or color depth) if multiple streams use different color formats and color depth.
- Encoder buffer option is enabled for Intra and Inter frame coding only. The Encoder buffer is used for motion estimation.

The Advanced Configuration tab, shown in Figure 10-2, optionally allows you to manually override the encoder buffer depth.

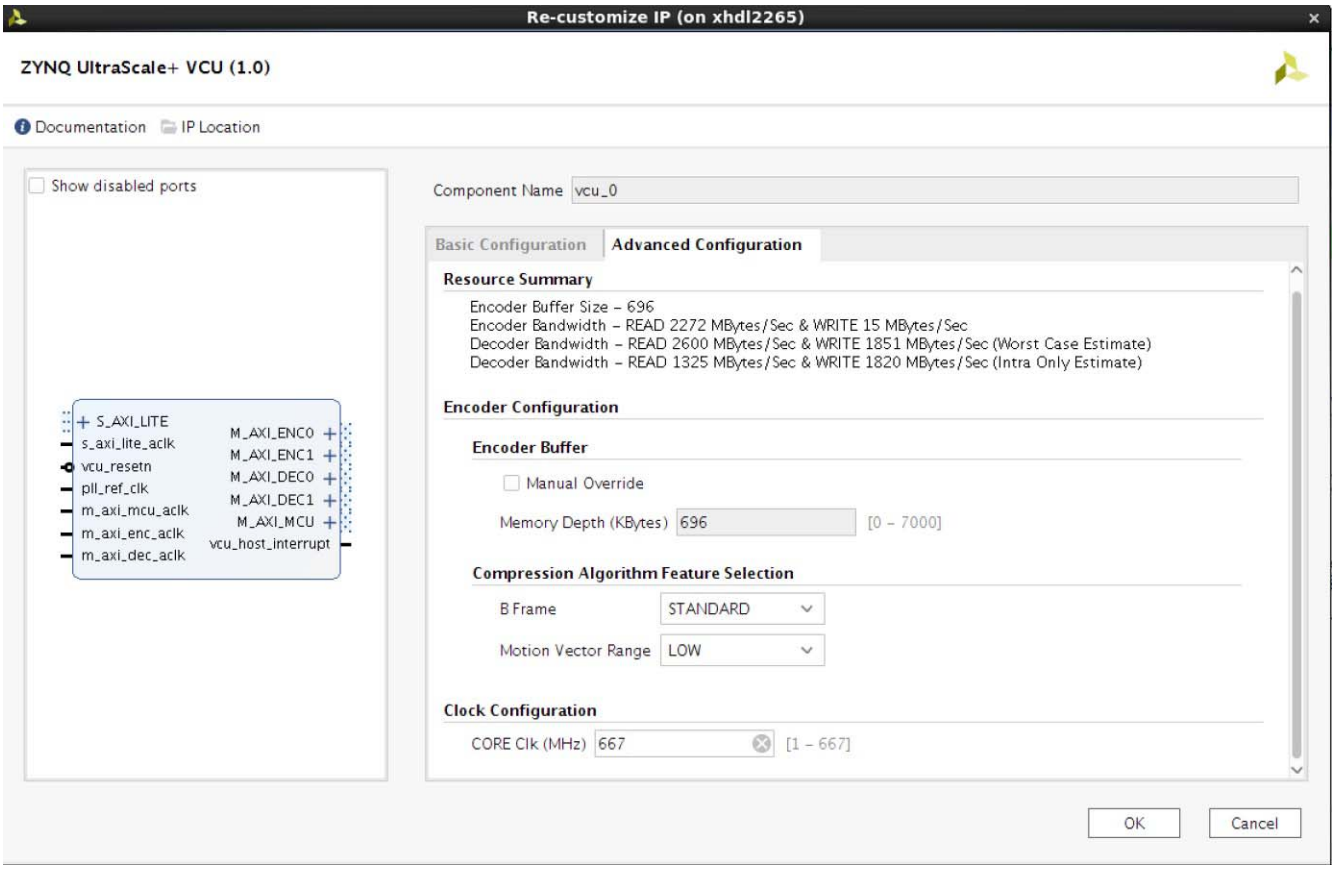


Figure 10-2: **Advanced Configuration Tab**

Interfacing the Core with Zynq UltraScale+ MPSoC Devices

To integrate the VCU core into an IP Integrator (IPI) block design, perform the following steps:

1. Launch the Vivado IDE and create a new project. (Figure 10-3)

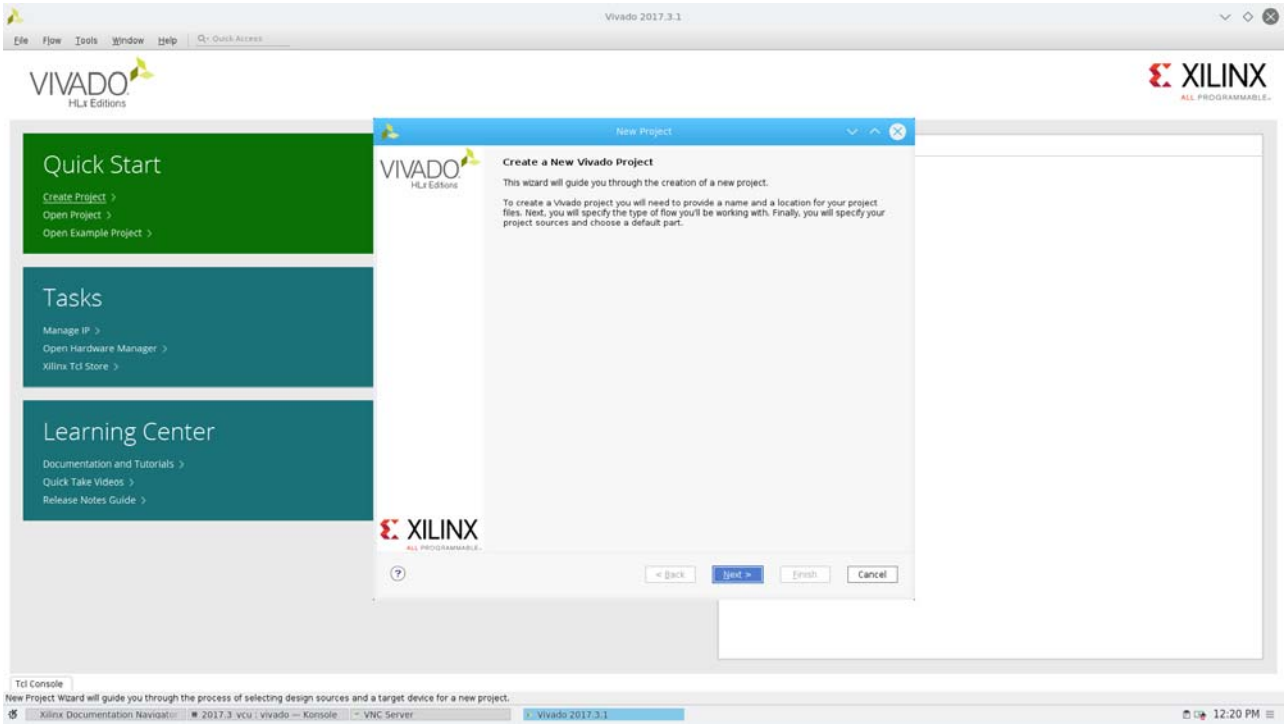


Figure 10-3: **New Vivado Project**

2. Click **Next** on **New Project** wizard until you reach the **Family Selection** window.
3. Select a target device for the VCU core. (Figure 10-4)

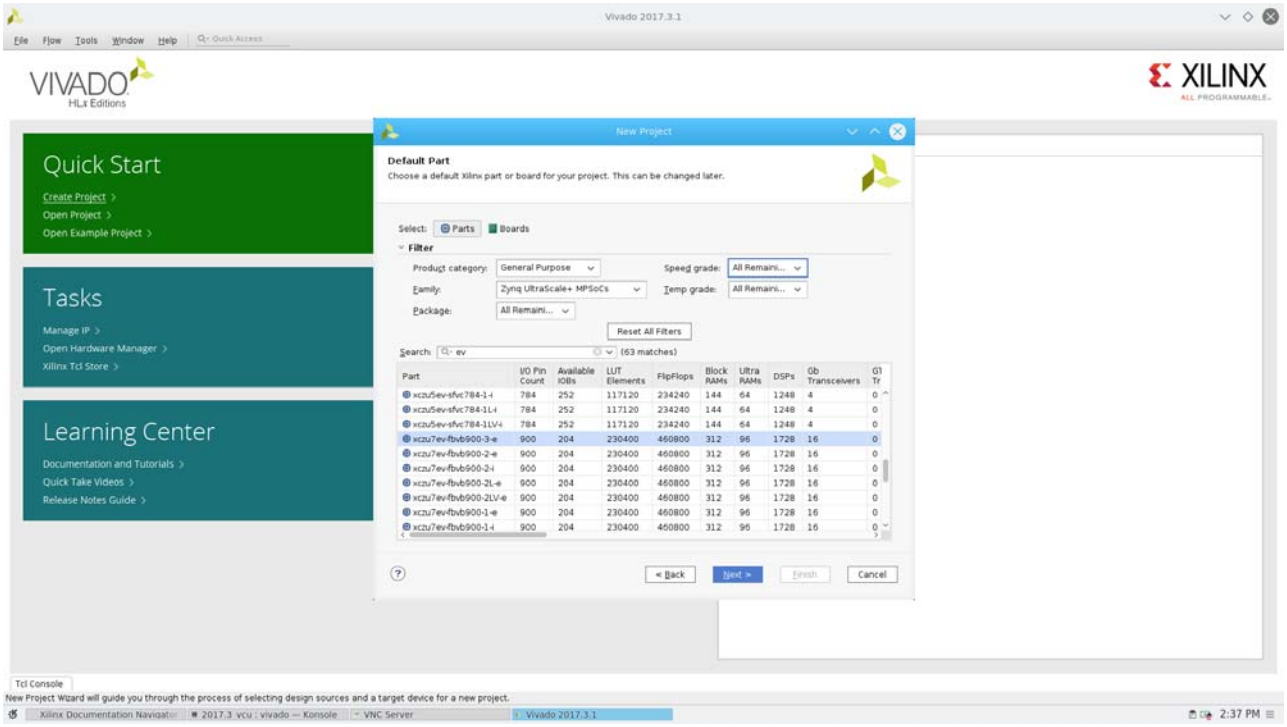


Figure 10-4: **Family Selection Tab**

4. Click on the **Project Settings** window. Click on the Implementation, as shown in Figure 10-5.

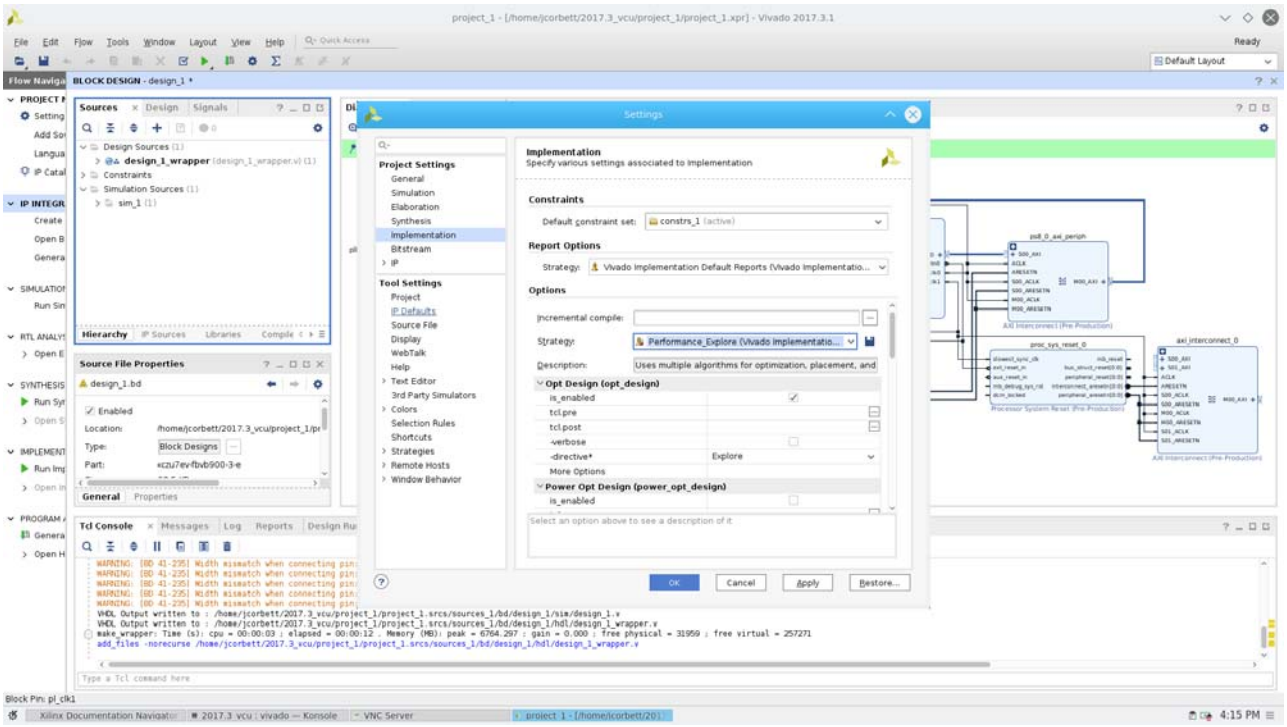


Figure 10-5: **Project Settings Tab**

5. In the **Settings** window, enable the **Performance_Explore** option.
   Settings > Implementation > Options > Strategy: Performance_Explore
   See *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* [Ref 7] for more information.
6. Click on **Create Block Design** option.
7. Click on **Add IP** option and type VCU. The following IP appears.

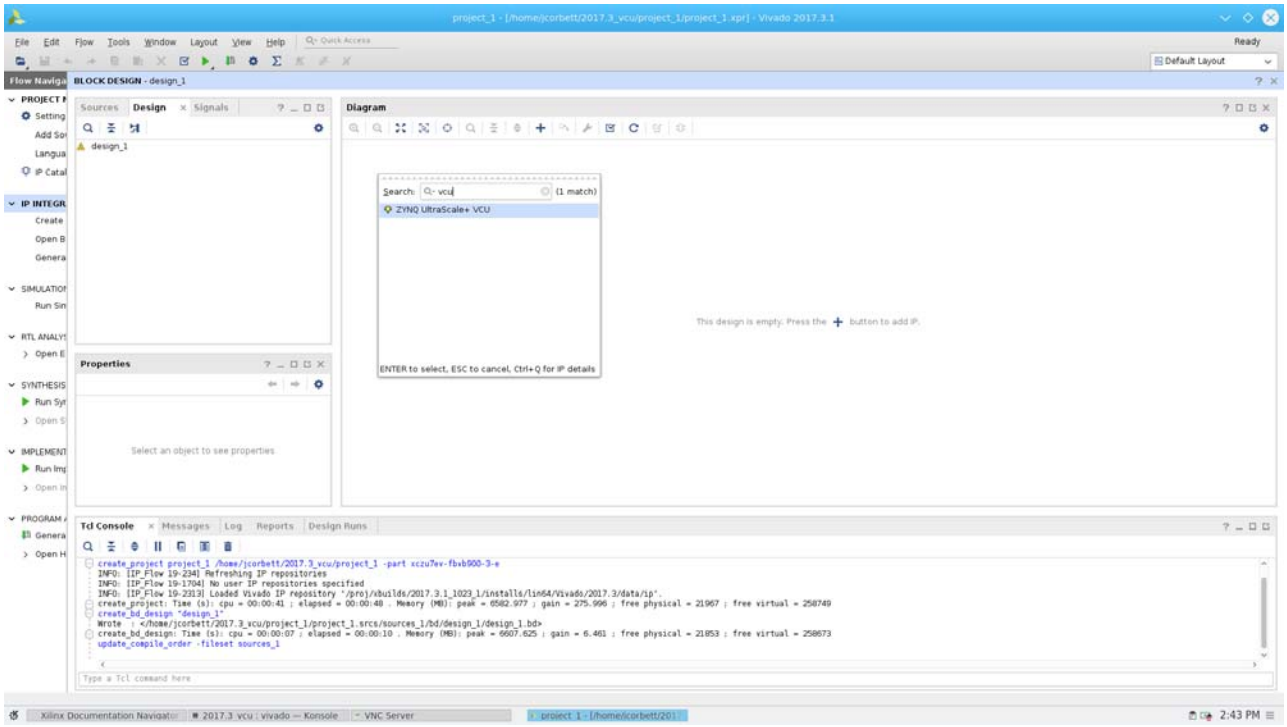


Figure 10-6: **Zynq UltraScale+ VCU**

8. Add Zynq UltraScale+ VCU to the block design.
9. Add Zynq UltraScale+ MPSoC IP to the block design as shown in Figure 10-7.

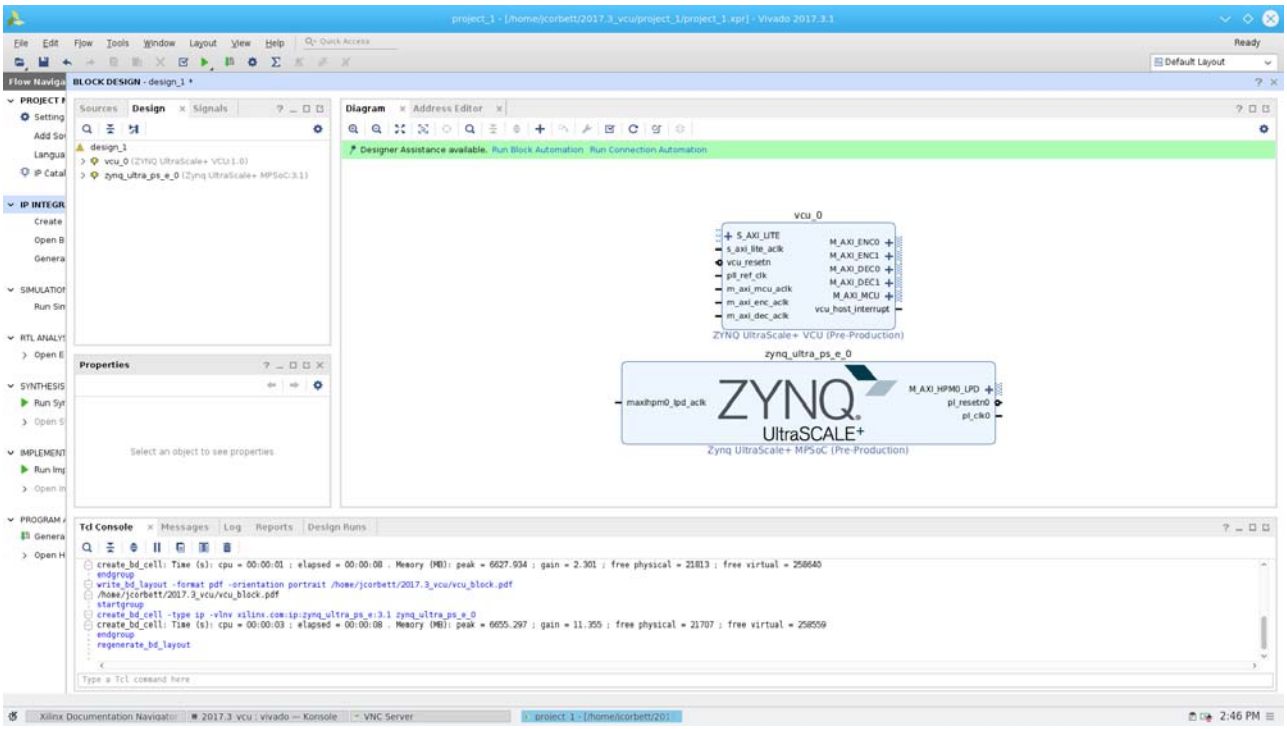


Figure 10-7: **Zynq UltraScale+ MPSoC**

10. Configure Zynq UltraScale+ MPSoC to enable AXI slave interfaces, clocking, and PL-PS interrupt signal per your design requirements. Refer to *Zynq UltraScale+ MPSoC Processing System Product Guide* [Ref 14] for configuration options of the Zynq UltraScale+ MPSoC IP.

    Figure 10-8 shows an example of configuring the PS-PL interface signals.

11. Select PL1 clock frequency as 333 MHz as shown in Figure 10-8.

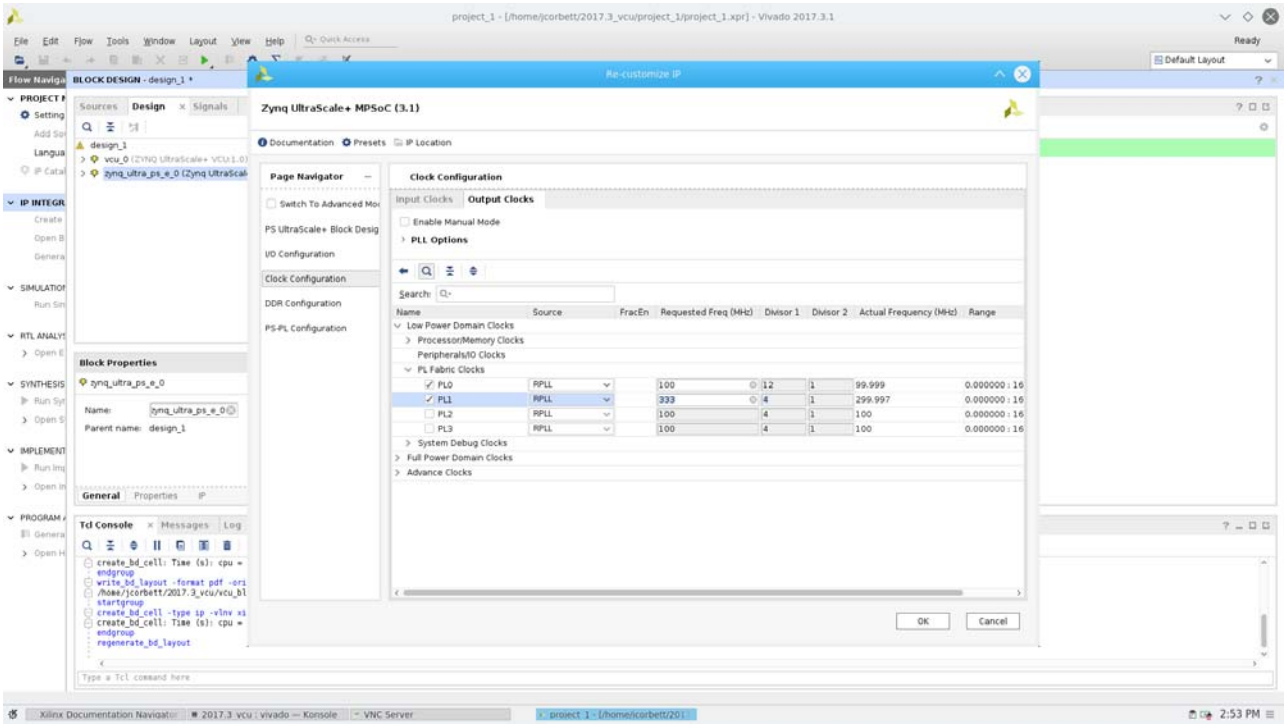


Figure 10-8: **Re-customize IP**

12. Enable IRQ0 [0-7] and HP0-3 ports as shown in Figure 10-9.

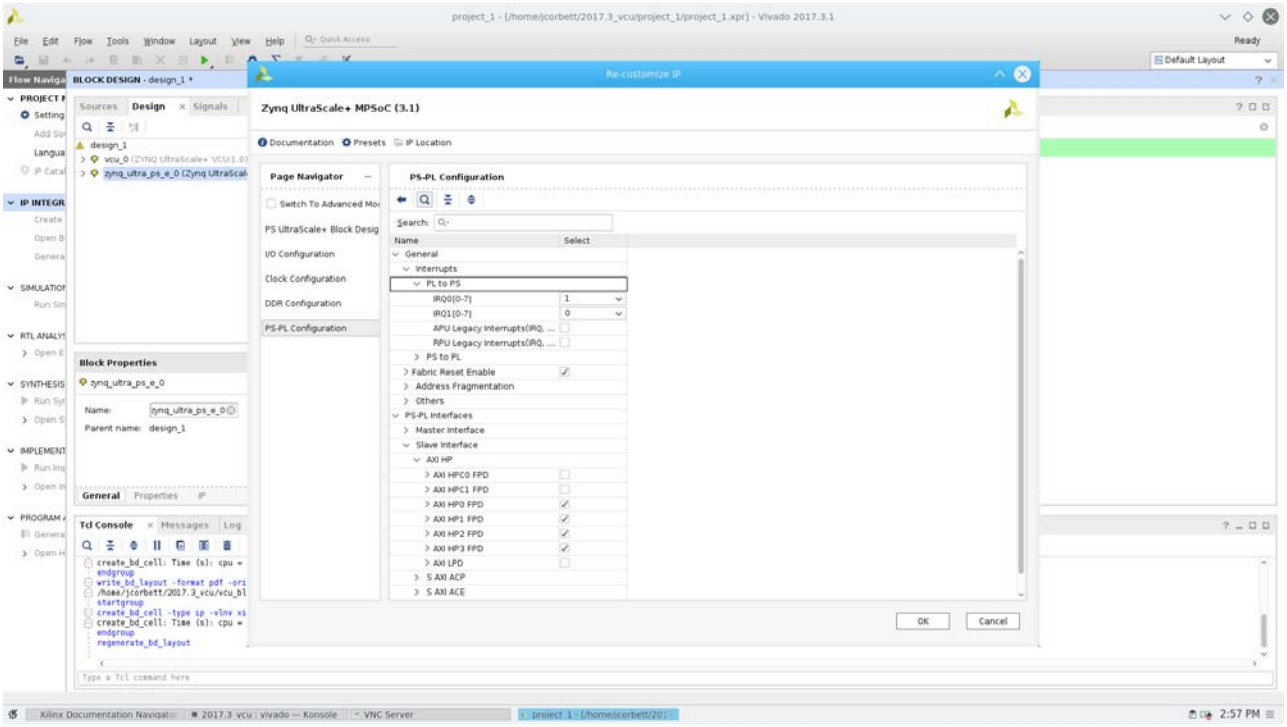


Figure 10-9: **Output PL-PS Configuration**

13. Use connection automation to connect the S_AXI_LITE interface of VCU IP to the M_AXI_HPM0_LPD interface.

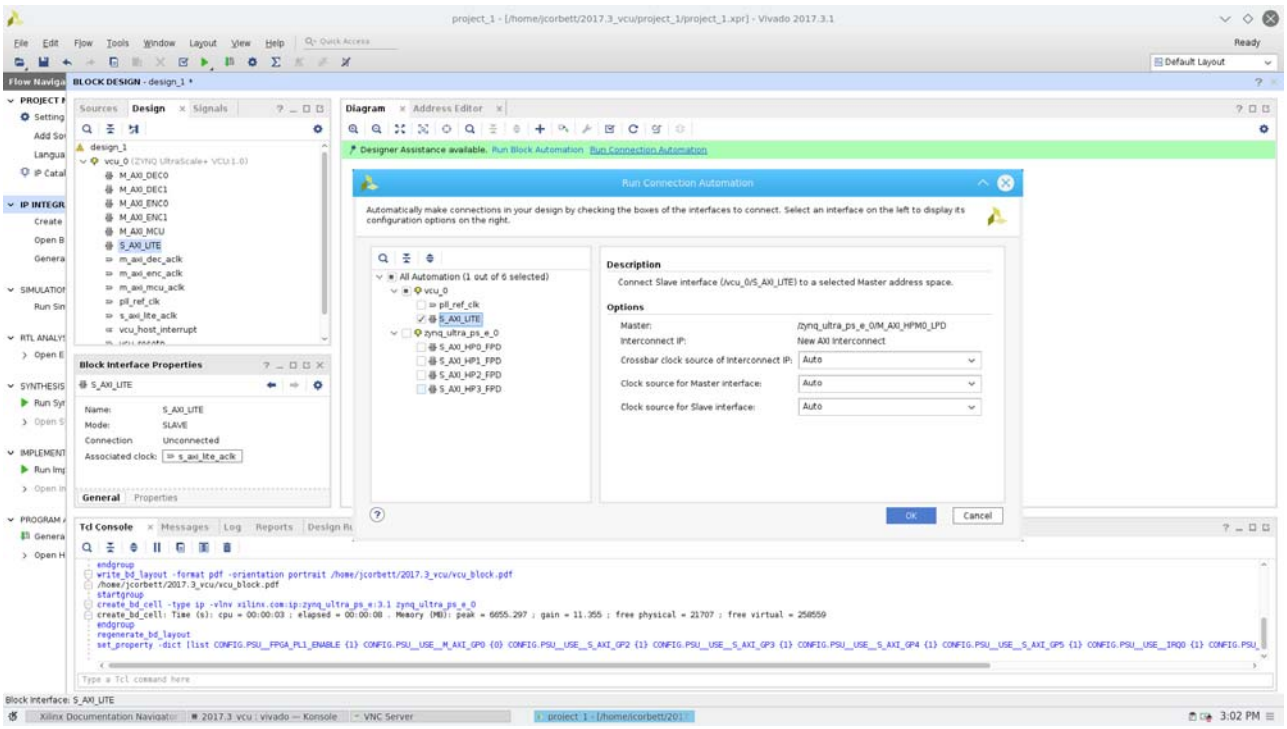


Figure 10-10: **Connection Wizard**

14. Connect the following interfaces manually:
    - Zynq UltraScale+ VCU.M_AXI_ENC1 to Zynq UltraScale+ MPSoC.S_AXI_HP1_FPD
    - Zynq UltraScale+ VCU.M_AXI_DEC0 to Zynq UltraScale+ MPSoC.S_AXI_HP2_FPD
    - Zynq UltraScale+ VCU.M_AXI_DEC1 to Zynq UltraScale+ MPSoC.S_AXI_HP3_FPD

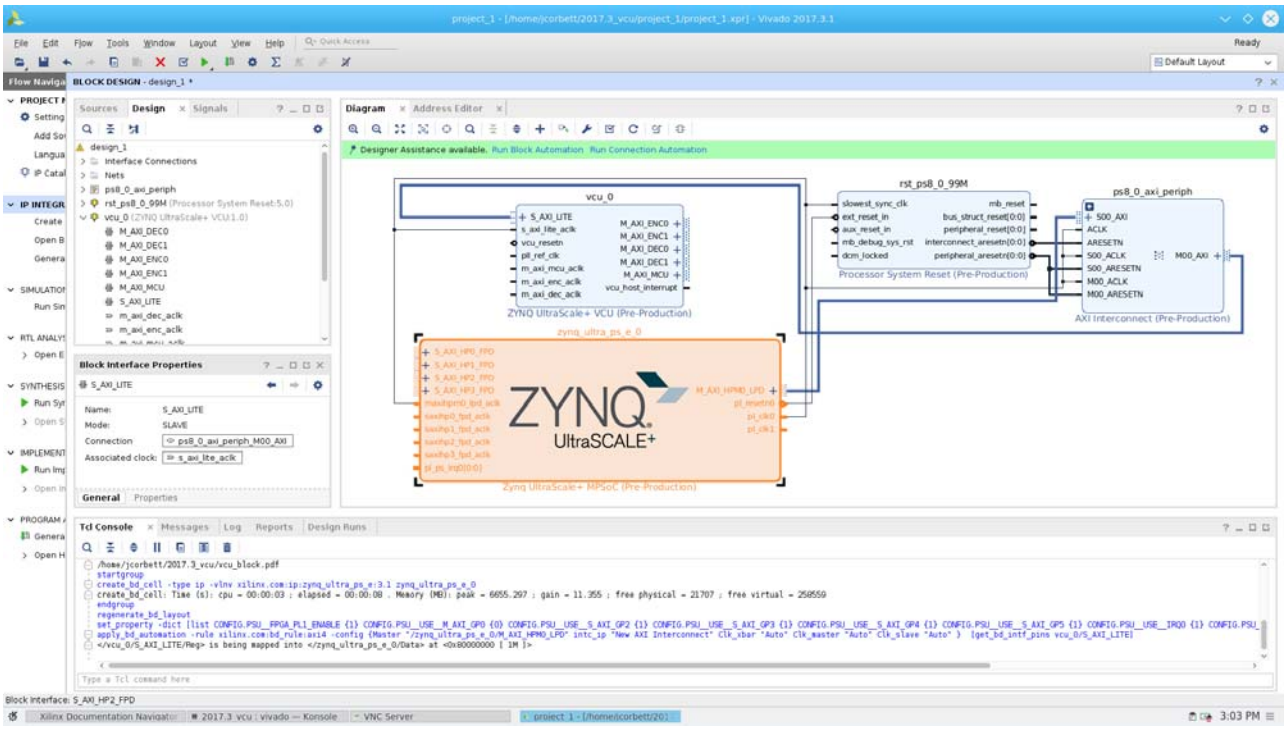


Figure 10-11: **Connection Wizard**

15. Add the AXI Interconnect IP and set number of slave interfaces to **2** and master interface to **1** as shown in Figure 10-12.

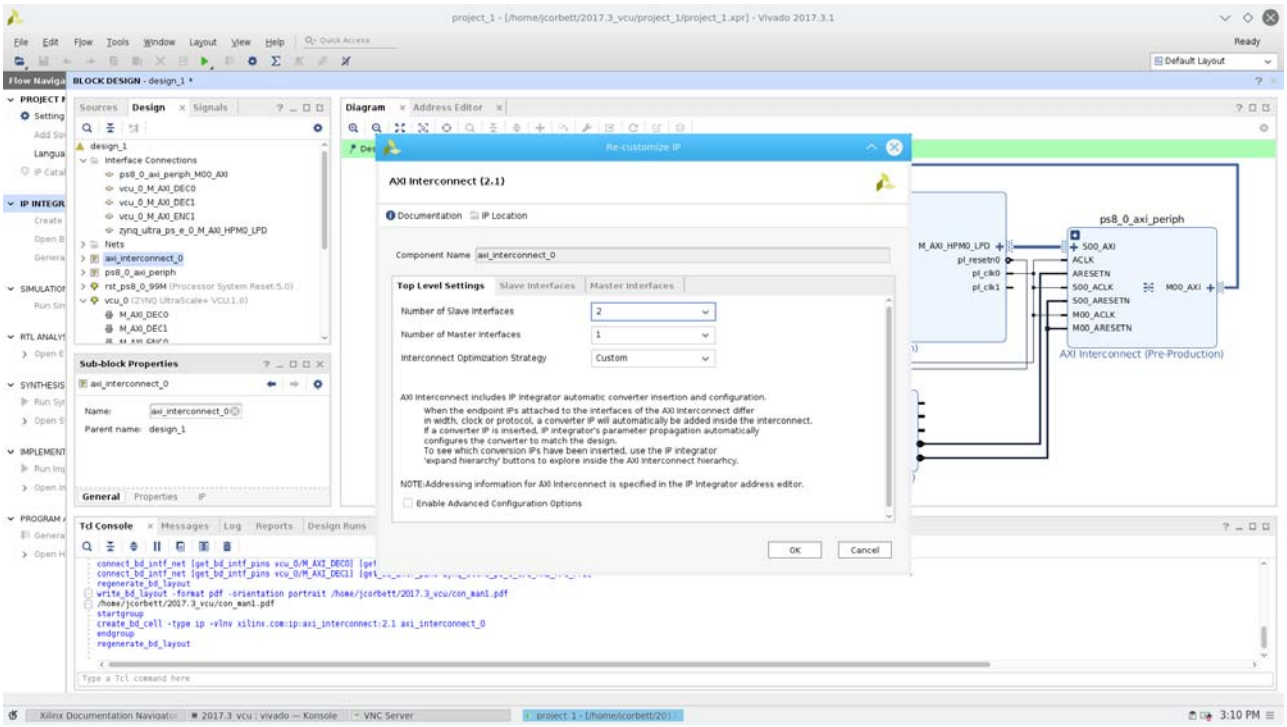


Figure 10-12: **AXI Interconnect**

16. Perform the following connections manually:
    - Instantiate the processor system reset IP. A second reset block is needed for the `pl_clk1` clock domain.
    - Connect the slowest sync clock to the `pl_clk1` port.
    - Use the `interconnect_aresetn` port as the `ARESETN` input to the AXI Interconnect IP core.
    - Use `peripheral_aresetn` port as a reset input to the `S00_ARESETN`, `S01_ARESETN`, and `M00_ARESETN` ports as shown in Figure 10-13.
    - Connect the `ext_reset_n` signal to the `pl_resetn0` signal of Zynq UltraScale+ MPSoC.
    - Connect the `vcu_host_interrupt` to the `pl_ps_irq` port of Zynq UltraScale+ MPSoC IP.

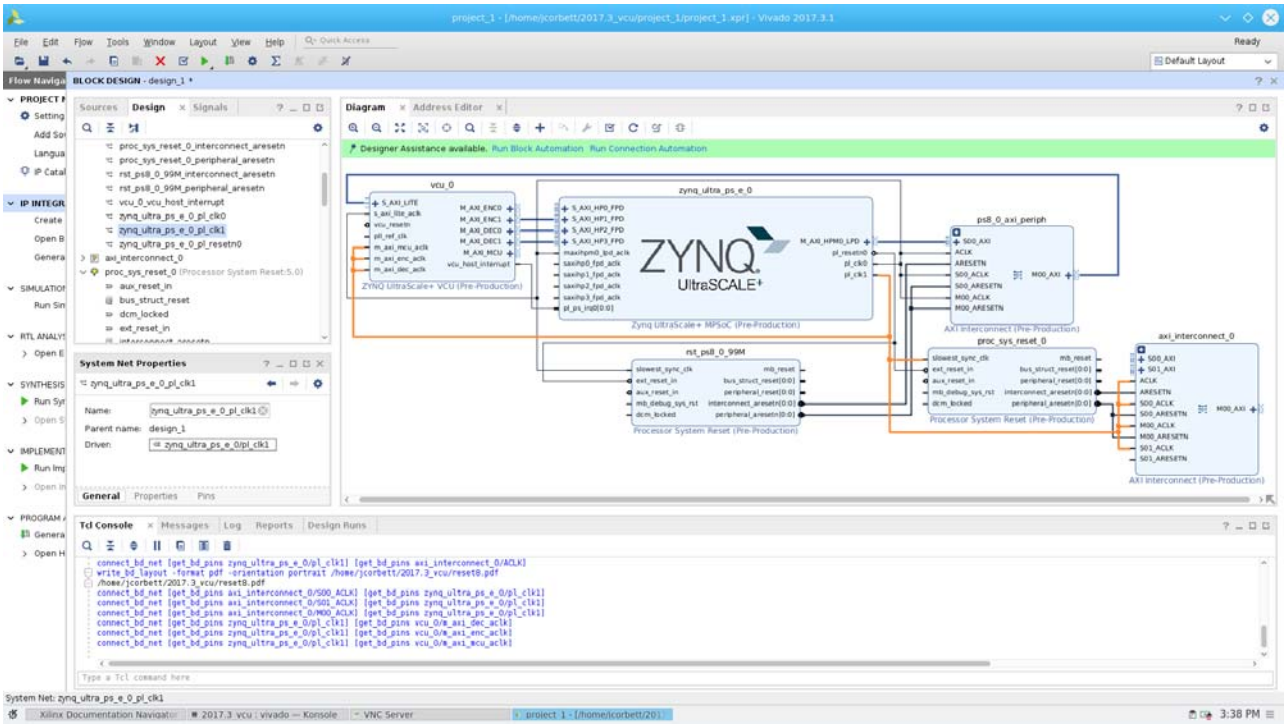


Figure 10-13: **Diagram**

17. Connect up the following clocks to the `pl_clk1` output of Zynq UltraScale+ MPSoC core:
    - AXI Interconnect - aclk
    - AXI Interconnect - s00_aclk
    - AXI Interconnect - s01_aclk
    - AXI Interconnect - m01_aclk

- VCU - m_axi_mcu_aclk
- VCU - m_axi_enc_aclk
- VCU - m_axi_dec_aclk
- Zynq UltraScale+ MPSoC - saxihp0_fpd_aclk
- Zynq UltraScale+ MPSoC - saxihp1_fpd_aclk
- Zynq UltraScale+ MPSoC - saxihp2_fpd_aclk
- Zynq UltraScale+ MPSoC - saxihp3_fpd_aclk

18. Connect `saxihp0_fpd_aclk`, `saxihp1_fpd_aclk`, `saxihp2_fpd_aclk` and `saxihp3_fpd_aclk` to `pl_clk1` output of Zynq UltraScale+ MPSoC core.
19. Tie off `vcu_resetn` signal of Zynq UltraScale+ VCU constant 1 using LogiCORE IP.
20. Make `pll_ref_clk` signal as external.
21. In the **Address Editor** tab, expand EncData address segment and auto assign the addresses. Figure 10-14 shows an example address map.

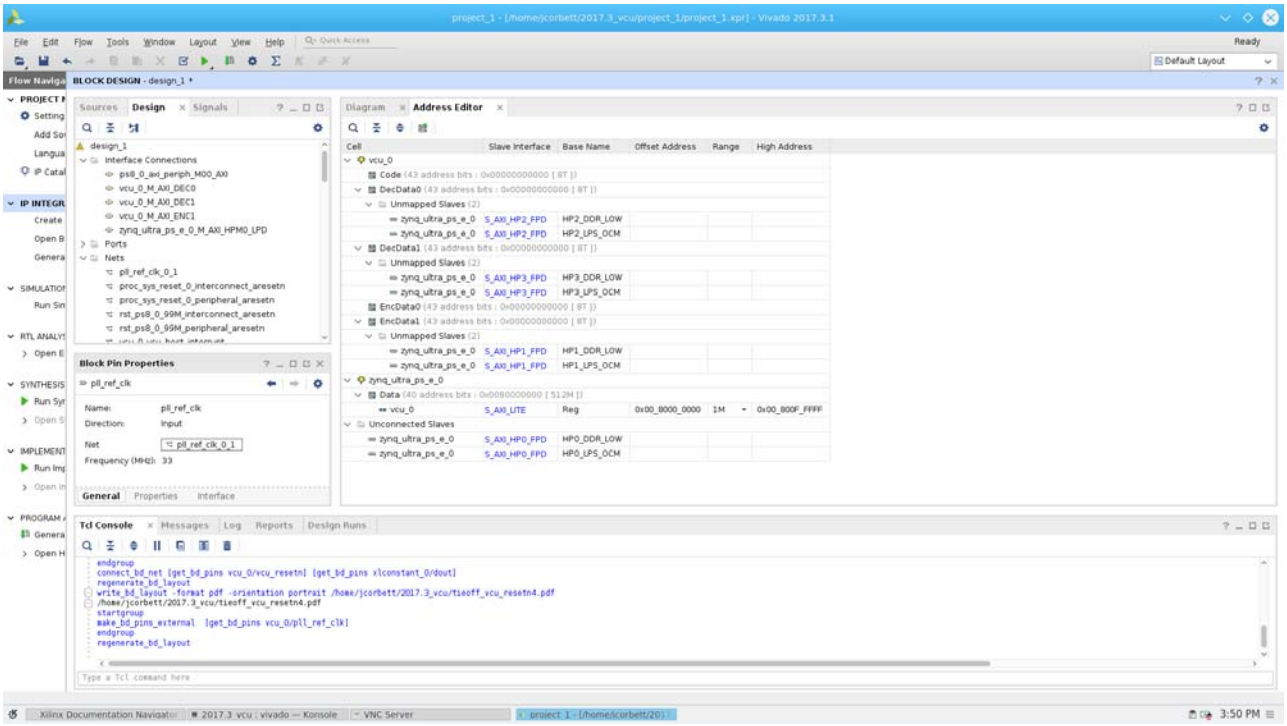


Figure 10-14: **Address Editor**

22. Click on **Validate Block Design** to validate the connections.

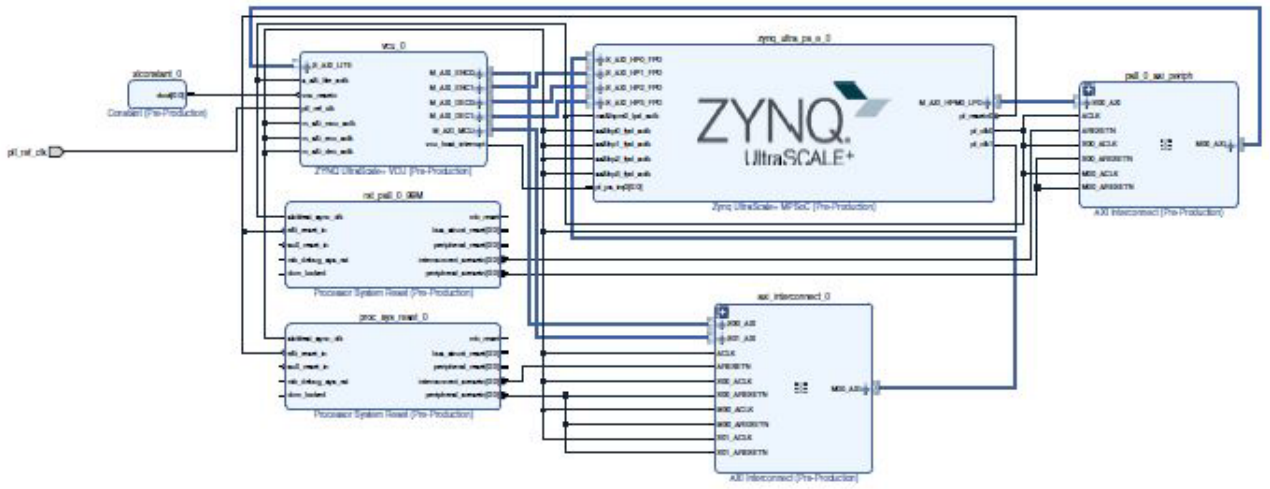


Figure 10-15: **Validate Block Design**

23. Create a top-level Vivado wrapper by right-clicking on Block Design and selecting **Create HDL Wrapper** option as shown in Figure 10-16.

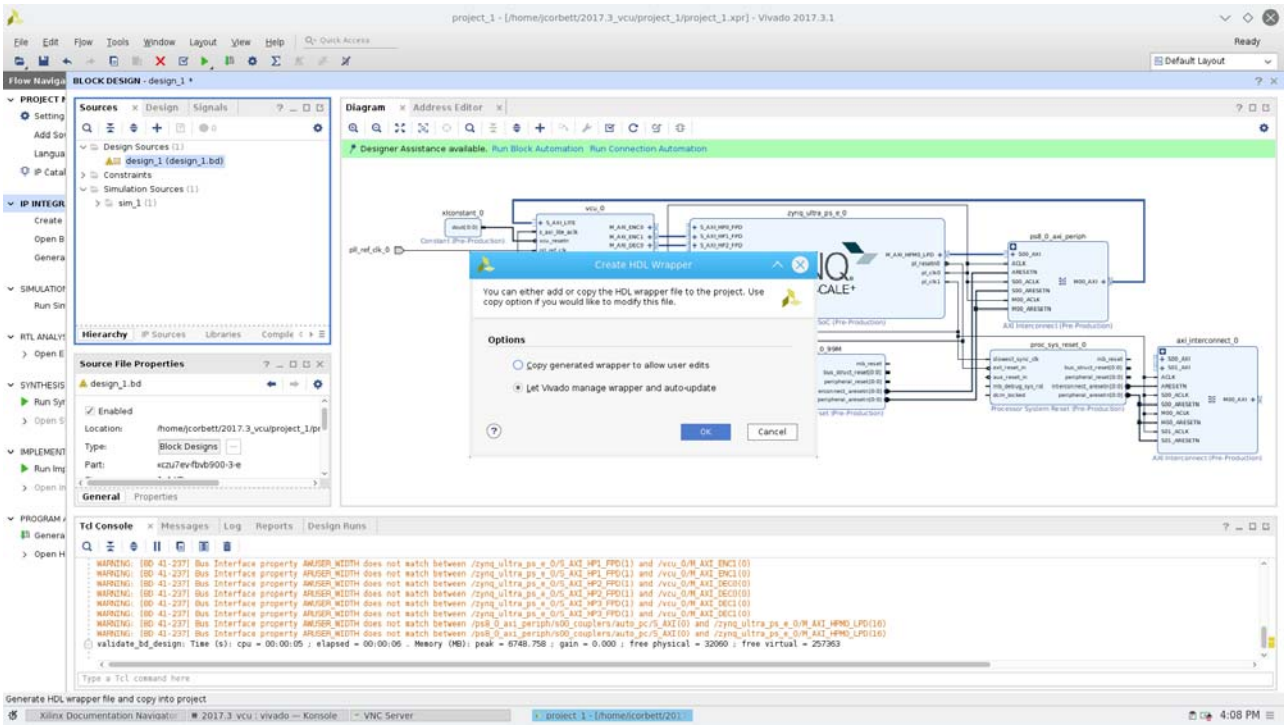


Figure 10-16: **Create HDL Wrapper**

24. Add constraints file to the project.

25. Add the constraints file (**.xdc**) from the board support package if available. If no constraints file is available, in the I/O Ports window, for `pll_ref_clk_0`, the I/O Std must be changed from **LVCMOS18 (Default)** to **LVCMOS18**.

And on the same row, the Fixed checkbox must be checked. This corresponds to an XDC file containing the following:

- `set_property IOSTANDARD LVCMOS18 [get_ports pll_ref_clk_0].`
- `set_property PACKAGE_PIN AA2 [get_ports pll_ref_clk_0]`

26. Click on the **Run Synthesis**, **Run Implementation**, or **Generate Bitstream** option.

Constraining the Core

The necessary XDC constraints are delivered with the core generation in the Vivado Design Suite.

Required Constraints

This section is not applicable for this IP core.

Device, Package, and Speed Grade Selections

This section is not applicable for this IP core.

Clock Frequencies

There is no restriction for speed grade. All speed grades support the max frequency of operation.

Clock Management

This section is not applicable for this IP core.

Clock Placement

This section is not applicable for this IP core.

Banking

This section is not applicable for this IP core.

Transceiver Placement

This section is not applicable for this IP core.

I/O Standard and Placement

This section is not applicable for this IP core.

Synthesis and Implementation

For details about synthesis and implementation, see the *Vivado Design Suite User Guide: Designing with IP* (UG896) [Ref 2].

Application Software Development

Overview

The Video Code Unit (VCU) software stack has a layered architecture programmable at several levels of abstraction available to software developers, as shown in Figure. The application interfaces from high level to low level are:

- GStreamer
- OpenMAX Integrator Level
- VCU Control Software

The GStreamer is the cross-platform open source multimedia framework. GStreamer provides the infrastructure to integrate multiple multimedia components and create pipelines. The GStreamer framework is implemented on the OpenMAX™ Integration Layer API-supported GStreamer version is 1.8.3. [Ref 17].

The OpenMAX Integration Layer API [Ref 16] defines a royalty-free standardized media component interface to enable developers and platform providers to integrate and communicate with multimedia codecs implemented in hardware or software.

The VCU Control Software is the lowest level visible software visible to VCU application developers. All VCU applications must use a Xilinx-provided VCU Control Software. The VCU Control Software includes custom kernel modules, custom user space library, and the AI_Encode and AI_Decode applications. The OpenMAX IL (OMX) layer is integrated on top of the VCU Control Software.

User applications can use the layer or layers of the VCU software stack are most appropriate to their requirements.

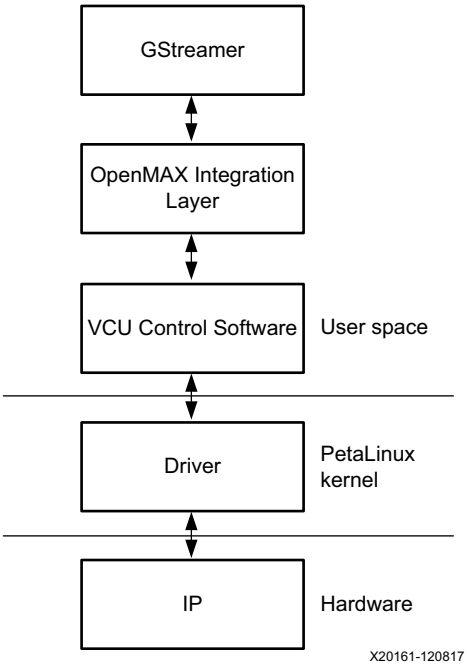


Figure 11-1: **VCU Software Stack**

Software Prerequisites

The application software using the VCU is written on top of the following libraries and modules, shown in Table 11-1. All of these are included in Xilinx® PetaLinux included in Xilinx Software Development Kit (SDK) 2017.4. Refer to AR69952 for updates to this information.

Table 11-1: **Application Software**

Software	Version	Source
GStreamer library	1.8.3	https://gstreamer.freedesktop.org
OpenMAX™ Integration Layer API	1.1.2	https://github.com/Xilinx/vcu-omx-il
VCU Control Software	1.0.35	https://github.com/Xilinx/vcu-ctrl-sw
VCU firmware	1.0.0	https://github.com/Xilinx/vcu-firmware
VCU binaries	1.0.0	https://github.com/Xilinx/vcu-binaries
VCU recipe files		https://github.com/Xilinx/meta-petalinux/tree/master/recipes-multimedia
VCU kernel modules		https://github.com/Xilinx/vcu-modules

Encoder Features

The VCU supports multi standard video encoding, shown in Table 11-2.

Table 11-2: **Encoder Supported Features**

Video Coding Parameter	H.265 (HEVC)	H.264 (AVC)
Profiles	Main Main Intra Main10 Main10 Intra Main 4:2:2 10 Main 4:2:2 10 Intra	Baseline Main High High10 High 4:2:2
Levels	Up to 5.1 High Tier	Up to 5.2
Resolution and Frame Rate (1)(2)	3840×2160p60 3840×2160p30 1920×1080p60 1920×1080p30 1280×720p60 1280×720p30	3840×2160p60 3840×2160p30 1920×1080p60 1920×1080p30 1280×720p60 1280×720p30
Bit Depth		
GStreamer	8-bit	8-bit
OMX	10-bit	10-bit
VCU Control Software	10-bit	10-bit
Chroma Format		
GStreamer	4:2:0	4:2:0
OMX	4:2:0, 4:2:2	4:2:0, 4:2:2
VCU Control Software	4:2:0, 4:2:2	4:2:0, 4:2:2
Slices Types	I, P, and B	I, P and B
Bit Rate	Limited by level and profile	Limited by level and profile

Note:

1. The H.265 (HEVC) minimum picture resolution is 128×128.
2. The H.254 (AVC) minimum picture resolution is 80×96.

Encoder Configurable GStreamer Parameters

The Encoder GStreamer parameters are shown in Table 11-3.

Table 11-3: **Decoder Supported Features**

VCU Parameter	GStreamer Property	Description
Bit Rate	target-bitrate	The number of bits required for video compression per second. Specify bitrate in bits per second.
Rate Control Mode	control-rate	Available bitrate control modes, Constant Bitrate (CBR) and Variable Bitrate(VBR) Encoding 0=Disable (CONST_QP, Constant QP) 1=Variable (VBR) 2=Constant (CBR)
Target Bit Rate	target-bitrate	Target bitrate in Kbps
Maximum Bit Rate	max-bitrate	Max bitrate in Kbps, used in VBR rate control mode.
Profile	Through caps	Supported profiles for corresponding codec are mentioned in Table 1
Level	Through caps	Supported Levels for corresponding codec are mentioned in Table 1
Tier	Through caps	Supported Tier for H.265 (HEVC) codec is mentioned in Table 1
Slice QP / I-frame QP	quant-i-frames	Quantization parameter for I-frames in CONST_QP mode, also used as initial QP in other rate control modes. Range 0–51.
P-frame QP	quant-p-frames	Quantization parameter for P-frames in CONST_QP mode. Range 0–51.
B-frame QP	quant-b-frames	Quantization parameter for B-frames in CONST_QP mode. Range 0–51.
GOP Length	gop-length	Distance between two consecutive Intra frames. Specify integer value between 0 and 1000. Value 0 and 1 corresponds to Intra only encoding
Number of B-frames	b-frames	Number of B frames between two consecutive P frames. Specify value in integer between 0 and 4
Number of Slices	num-slices	Specifies the number of slices used for each frame. Each slice contains one or more full LCU row or rows and are spread over the frame as regularly as possible. Specify integer value between 1 and max supported. Max supported slices: H.264 (AVC): picture_height/16 H.265 (HEVC): the minimum of picture_height/32 and 22
Minimum QP	min-qp	Minimum QP value allowed in encoding session. Range 0–51.

Table 11-3: **Decoder Supported Features *(Cont'd)***

VCU Parameter	GStreamer Property	Description
Maximum QP	max-qp	Maximum QP value allowed in encoding session. Range 0–51.
GOP Configuration	gop-mode	Specifies Group Of Pictures configuration =default (IPPP mode with Intra period of 30) =low_delay_p (IPPPPP....) =low_delay_b (IBBBBB...) =pyramidal (Advanced Gop pattern with hierarchical B frame, works with B=3, 5 and 7)
Gradual Decoder Refresh	gdr-mode	Specifies which Gradual Decoder Refresh scheme should be used when gop-mode = low_delay_p =disable =vertical (Gradual refresh using a vertical bar moving from left to right) =horizontal (Gradual refresh using a horizontal bar moving from top to bottom)
QP Control Mode	qp-mode	Specifies how to generate the QP per MB/CU =uniform - All MBs uses same QP =auto - The QP is chosen according to the CU/MB content.
ssFiller data	filler-data	Enable/Disable filler data adding functionality in CBR rate control mode. True or False
Entropy Mode	entropy-mode	Specifies the entropy mode for H.264 (AVC) encoding process = 0 CAVLC = 1 CABAC
Constrained Intra Prediction	constrained-intra-pred	Enables/Disable Constrained Intra prediction feature in Encoding session. Boolean: true or false
Deblocking Filter	loop-filter	Enables/Disables the disable deblocking filter option 0 = Enables deblocking filter 1 = Disables deblocking filter 2 = DisableSliceBoundary, excludes slice boundaries from filtering.
IDR picture frequency	gop-freq-idr	Specifies the number of frames between consecutive IDR (instantaneous decoder refresh) pictures
Initial Removal Delay	initial-delay	Specifies the initial removal delay as specified in the HRD model in milliseconds
Coded Picture buffer size	cpb-size	Specifies the Coded Picture Buffer as specified in the HRD model in milliseconds

Table 11-3: **Decoder Supported Features *(Cont'd)***

VCU Parameter	GStreamer Property	Description
Dependent slice	dependent-slice	Specify whether the additional slices are Dependent other slice segments or regular slices in Multiple slices encoding session. Used in H.265 (HEVC) encoding only. Boolean: true or false
Target slice size	slice-size	Target Slice Size, If set to 0, slices are defined by the num-slices parameter, else it specifies the target slice size in bytes. Encoder uses it to automatically split the bitstream into approximately equally-sized slices. Range 0–65535.
Scaling Matrix	scaling-list	specifies the scaling list mode =0 (flat) FLAT scaling list mode =1 (default) Default scaling list mode
Input Port Mode	ip-mode	Specifies encoder input port mode. Takes integer between 0 and 3 0: Default GStreamer implementation 1: Zero-copy between OMX and GStreamer 2: DMA import 3: DMA export
Encoder Buffer Size	prefetch-buffer-size	Value of encoder buffer size is in KB, It helps in reducing memory bandwidth, also can slightly reduce video quality. Specify value in KB.
Vertical search range	low-bandwidth	Specifies low bandwidth mode. It decreases the vertical search range used for P-frame motion estimation. True or false.
SliceHeight	sliceHeight	Specify input buffer height alignment of upstream element if any.
Stride	stride	Specify input buffer stride alignment of upstream element if any.
Aspect-ratio	aspect-ratio	Selects the display aspect ratio of the video sequence to be written in SPS/VUI. = auto - 4:3 for SD video,16:9 for HD video, unspecified for unknown format =aspect_ratio_4_3 - 4:3 aspect ratio =aspect_ratio_16_9 - 16:9 aspect ratio =none - Aspect ratio information is not present in the stream

Table 11-4: **Encoder Configuration Parameters at GStreamer**

Video Coding Parameter	H.265 (HEVC)	H.264 (AVC)
Profiles	Main Main Intra Main 10 Main 10 Intra Main 422 10 Main 422 10 Intra	Baseline (Except FMO/ASO) Main High High10 High 422
Levels	Up to 5.1 High Tier	Up to 5.2
Resolutions	3840×2160p60 3840×2160p30 1920×1080p60 1920×1080p30 1280×720p60 1280×720p30	3840×2160p60 3840×2160p30 1920×1080p60 1920×1080p30 1280×720p60 1280×720p30
Chroma format 4:2:0, 8bit	Yes	Yes

Decoder Features

Note: YUV422 Chroma format and 10-bit is supported only at VCU Control Software level.

GStreamer Decoder Parameters

Table 11-5: **Decoder Configurable Parameters at GStreamer**

Parameter	Gst-property	Options
Input Port mode	ip-mode	Specifies decoder input port mode. Takes integer between 0 and 1 0: Default GStreamer implementation 1: Zero-copy between OMX and GStreamer
Output Port Mode	op-mode	Specifies decoder output port mode. Takes integer between 0 and 1 0: Default GStreamer implementation 1: DMA exporter
Entropy Buffers	internal-entropy-buffers	Specifies decoder internal entropy buffers, used to smooth out entropy decoding performance. Specify values in integer between 2 and 16. Default value is 5. Increasing buffering-count increases Decoder memory foot print. User can set this value to 10 for higher bitrate use-cases, e.g >100Mbps.
Latency	latency-mode	Specifies decoder latency mode. 0: Default mode 1: Low-latency. Currently both modes has similar latency performance, it is reserved for future usage.

Decoder Maximum Bitrate Tests

Pipeline used for measuring maximum bit rate for which decoder produces 30 fps.

Table 11-6: **Decoder Maximum Bitrate Tests**

Codec	IPPP	IPBB	Intra only
H.264 (AVC)	350 Mbps	335 Mbps	400 Mbps
H.265 (HEVC)	275 Mbps	275 Mbps	270 Mbps

Example pipeline used for measurement:

```
gst-launch-1.0 filesrc location="/run/test_file.mp4 " ! qtdemux ! h264parse !
omxh264dec ip-mode=1 op-mode=1 internal-entropy-buffers=9 ! queue max-size-bytes=0 !
fpsdisplaysink name=fpssink text-overlay=false video-sink=kmssink sync=true
fps-update-interval=3000 -v
```

GStreamer

Sample GStreamer Pipelines Using gst-launch-1.0

Each of the samples below consists of a single command line. Line breaks have been inserted for presentation only.

H264 Decoding

```
gst-launch-1.0 filesrc location="input-file.mp4" ! qtdemux name=demux demux.video_0
! h264parse ! omxh264dec ip-mode=1 op-mode=1 ! queue max-size-bytes=0 ! kmssink
```

H265 Decoding

```
gst-launch-1.0 filesrc location="input-file.mp4" ! qtdemux name=demux demux.video_0
! h265parse ! omxh265dec ip-mode=1 op-mode=1 ! queue max-size-bytes=0 ! kmssink
```

Higher Bitrate Bitstream Decoding

The Decoder generally might take higher than one frame period time for high bitrate stream decoding (> 100Mbps, 4kP30). Use the below options to get better decoder performance.

Increase internal decoder buffers (buffering-count parameter) to 9 or 10.

Add a queue at decoder input side.

```
gst-launch-1.0 filesrc location="input-file.mp4" ! qtdemux name=demux demux.video_0
! h264parse ! queue max-size-bytes=0 ! omxh264dec ip-mode=1 op-mode=1
internal-entropy-buffers=10 ! queue max-size-bytes=0 ! kmssink
```

H264 Encoding

```
gst-launch-1.0 filesrc location="input-file.yuv" ! videoparse format=nv12 width=3840
height=2160 framerate=30/1 ! omxh264enc ip-mode=1 ! filesink location=”output.h264”
```

H265 Encoding

```
gst-launch-1.0 filesrc location="input-file.yuv" ! videoparse format=nv12 width=3840
height=2160 framerate=30/1 ! omxh265enc ip-mode=1 ! filesink location=”output.h265”
```

Note: The input YUV should be in NV12 format.

Transcoding from H.264 to H.265

```
gst-launch-1.0 filesrc location="input-h264-file.mp4" ! qtdemux name=demux
demux.video_0 ! h264parse ! omxh264dec ip-mode=1 op-mode=1 ! omxh265enc ip-mode=2
stride=256 sliceHeight=64 ! filesink location="output.h265"
```

Transcoding from H.265 to H.264

```
gst-launch-1.0 filesrc location="input-h265-file.mp4" ! qtdemux name=demux
demux.video_0 ! h265parse ! omxh265dec ip-mode=1 op-mode=1 ! omxh264enc ip-mode=2
stride=256 sliceHeight=64 ! filesink location="output.h264"
```

Transcoding and Streaming via Ethernet

```
gst-launch-1.0 filesrc location="test_0003.mp4" ! qtdemux ! h264parse ! omxh264dec
ip-mode=1 op-mode=1 ! omxh265enc ip-mode=2 control-rate=2 target-bitrate=20000
stride=256 sliceHeight=64 ! h265parse ! queue ! rtph265pay ! udpsink host=192.168.1.1
port=50000 buffer-size=20000000 async=false max-lateness=-1 qos-dscp=60
```

Note: 192.168.1.1 is an example client IP address. You may need to modify this with actual client IP address.

Streaming via Ethernet and Decoding to the Display Pipeline

```
gst-launch-1.0 udpsrc port=50000 caps="application/x-rtp, media=video,
clock-rate=90000, payload=96, encoding-name=H265" buffer-size=20000000 !
rtpjitterbuffer latency=1000 ! rtph265depay ! h265parse ! omxh265dec ip-mode=1
op-mode=1 ! queue ! kmssink sync=false
```

Multistream Decoding

```
gst-launch-1.0 filesrc location=input_1920x1080.mp4 ! qtdemux ! h265parse ! tee
name=t t. ! queue ! omxh265dec ip-mode=1 op-mode=1 ! queue max-size-bytes=0 !
filesink location=output_0_1920x1080.yuv t. ! queue ! omxh265dec ip-mode=1 op-mode=1
! queue max-size-bytes=0 ! filesink location=output_1_1920x1080.yuv t. ! queue !
omxh265dec ip-mode=1 op-mode=1 ! queue max-size-bytes=0 ! filesink
```

```
location=output_2_1920x1080.yuv t. ! queue ! omxh265dec ip-mode=1 op-mode=1 ! queue
max-size-bytes=0 ! filesink location=output_3_1920x1080.yuv
```

Note: The element is used to feed same input file into 4 decoder instances. You can use separate gst-launch-1.0 application to fed different inputs.)

Multistream Encoding

```
gst-launch-1.0 filesrc location=input_nv12_1920x1080.yuv ! videoparse width=1920
height=1080 format=nv12 framerate=30/1 ! tee name=t t. ! queue ! omxh264enc
control-rate=2 target-bitrate=8000 ! video/x-h264, profile=high ! filesink location=
ouput_0.h264 t. ! queue ! omxh264enc control-rate=2 target-bitrate=8000 ! video/
x-h264, profile=high ! filesink location=ouput_1.h264 t. ! queue ! omxh264enc
control-rate=2 target-bitrate=8000 ! video/x-h264, profile=high ! filesink location=
ouput_2.h264 t. ! queue ! omxh264enc control-rate=2 target-bitrate=8000 ! video/
x-h264, profile=high ! filesink location= ouput_3.h264 t. ! queue ! omxh264enc
control-rate=2 target-bitrate=8000 ! video/x-h264, profile=high ! filesink location=
ouput_4.h264 t. ! queue ! omxh264enc control-rate=2 target-bitrate=8000 ! video/
x-h264, profile=high ! filesink location= ouput_5.h264 t. ! queue ! omxh264enc
control-rate=2 target-bitrate=8000 ! video/x-h264, profile=high ! filesink location=
ouput_6.h264 t. ! queue ! omxh264enc control-rate=2 target-bitrate=8000 ! video/
x-h264, profile=high ! filesink location= ouput_7.h264
```

Note: The element is used to feed same input file into 8 encoder instances. You can use separate gst-launch-1.0 application to fed different inputs.

Verified GStreamer Elements

Table 11-7: **Verified GStreamer Elements**

Element	Description
filesink	Writes incoming data to a file in the local file system
filesrc	Reads data from a file in the local file system
h264parse	Parses a H.264 encoded stream
h265parse	Parses a H.265 encoded stream
kmssink	Renders video frames directly in a plane of a DRM device
omxh264dec	Decodes OpenMAX H.264 video
omxh264enc	Encodes OpenMAX H.264 video
omxh265dec	Decodes OpenMAX H.265 video
omxh265enc	Encodes OpenMAX H.265 video
qtdemux	Demuxes a .mov file into raw or compressed audio and/or video streams.
queue	Queues data until one of the limits specified by the “max-size-buffers”, “max-size-bytes” or “max-size-time” properties has been reached
rtph264depay	Extracts an H.264 video payload from an RTP packet stream
rtph264pay	Encapsulates an H.264 video in an RTP packet stream
rtph265depay	Extracts an H.265 video payload from an RTP packet stream

Table 11-7: **Verified GStreamer Elements *(Cont'd)***

Element	Description
rtph265pay	Encapsulates an H.265 video in an RTP packet stream
rtpjitterbuffer	Reorders and removes duplicate RTP packets as they are received from a network source
tee	Splits data to multiple pads
udpsink	Sinks UDP packets to the network
updsrc	Reads UDP packets from the network
v4l2src	Captures video from v4l2 devices, like webcams and television tuner cards
videoparse	Converts a byte stream into video frames

OpenMax Sample OMX Test Application Pipelines Decoding File to File

H265 Decoding

```
omx_decoder.exe input-file.h265 -hevc -o out.yuv
```

The `omx_decoder.exe –help` command shows all the options.

Encoding File to File

```
omx_encoder.exe inputfile.yuv -w 352 -h 288 -r 30 -avc -o out.h264
```

The `omx_encoder.exe –help` command shows all the options.

Note: Input YUV file should be in NV12 or NV16 format for 8 bit.

Sample VCU Control Software Test Application Pipelines Decoding File to File

H264 Decoding

```
AL_Decoder -avc -in input-avc-file.h264 -out ouput.yuv
```

H265 Decoding

```
AL_Decoder -hevc -in input-hevc-file.h265 -out ouput.yuv
```

Encode File to File

```
AL_Encoder –cfg encode_simple.cfg
```

Note: You can use this as a demo example to run the encode and decode applications at control software level with different configurations. Sample config files and input yuv file mentioned in cfg can be found in the control software source tree test/cfg folder.

Note: Use notepad++ or Linux machine for editing encode_simple.cfg file.

VCU Control Software

The VCU Control Software operates on the frame or slice levels. Its responsibilities are:

- Generating NAL (Network Abstraction Layer) units for encoder.
- Parsing NAL units for decoder.
- Composing and queuing commands for each frame to the MCU Firmware.
- Retrieves status of each frame.
- Concatenates video bit stream generated by hardware and software.

Driver

There are three kernel drivers in PetaLinux associated with the VCU. The decoder driver is called al5d. The encoder driver is called al5e. The VCU kernel driver is called allegro. The allegro driver has the following responsibilities:

- Loading the MCU firmware.
- Initiating the MCU boot sequence.
- Writing mailbox messages into memory shared between APU and MCU.
- Providing notification of new mailbox messages.

MCU Firmware

The MCU Firmware running on the MCU has the following responsibilities:

- Transforming frame-level commands from VCU Control Software to slice level commands for the hardware IP core.
- Configuring hardware registers for each command.
- Performing rate control between each frame.

Encoding Stack

Figure 11-2 shows the Encoding software stack.

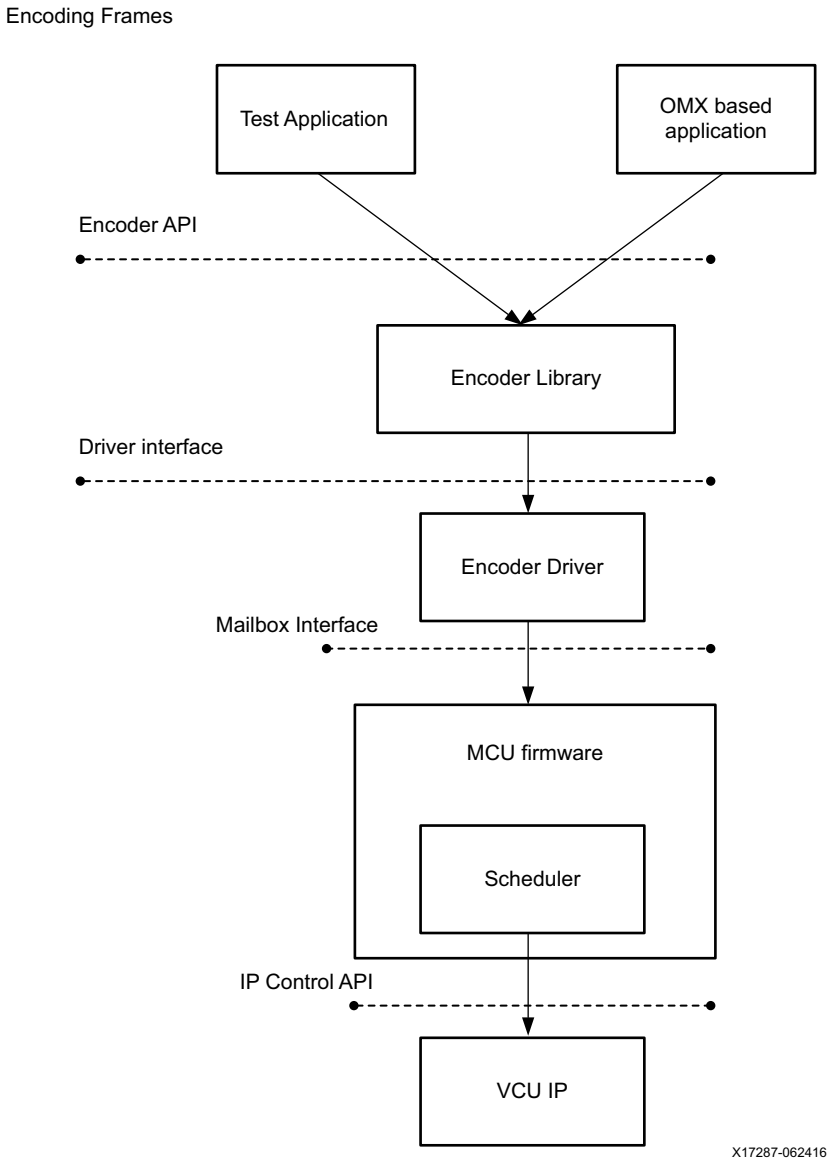


Figure 11-2: **Encoder Overview**

Application

Application refers to any OpenMAX based or standalone application that uses the VCU's underlying encoder capabilities.

Encoder Library

Encoder library provides the entry points for configuring the Encoder and sending frames to the Encoder.

Encoder Driver

The Encoder driver passes control information and buffer pointers of the video bit stream on which VCU encoder has to operate to the MCU Firmware. The Encoder driver uses mailbox communication technique to pass this information to MCU firmware.

MCU Firmware

Firmware receives control and buffer information through mailbox. Appropriate action is taken and status is communicated back to Encoder driver.

Scheduler

The Scheduler directs the activity of the hardware, handles interrupts, and manages the multi-channel and multi-slice aspects of the encoding.

Decoder Stack

Figure 11-3 shows the Decoder software stack.

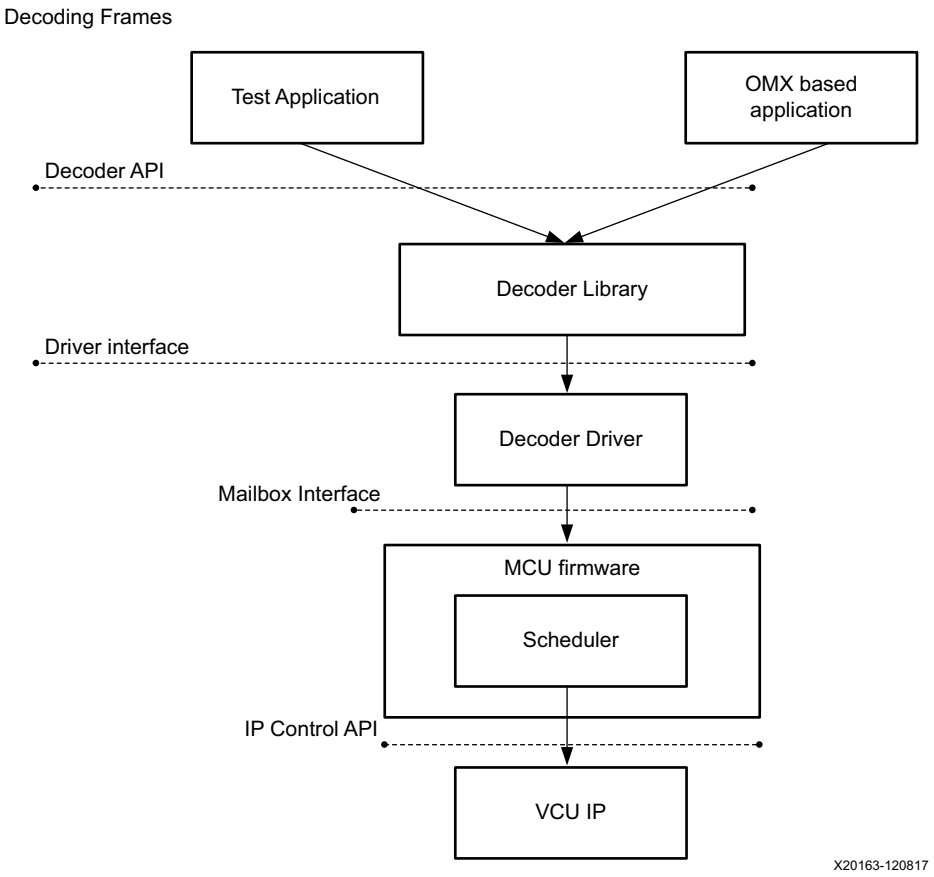


Figure 11-3: **Decoder Overview**

Application

The application can either be test pattern generator or an OpenMAX-based application that uses the VCU Decoder.

Decoder Library

The Decoder library enables applications to communicate with the MCU firmware through Decoder driver.

Decoder Driver

The Decoder driver passes control information as well as buffer pointers of the video to the MCU Firmware. The Decoder driver uses a mailbox communication technique to pass this information to the MCU firmware.

MCU Firmware

The firmware receives control and buffer information through mailbox. Appropriate action is taken and status is communicated back to Decoder driver.

Scheduler

The Scheduler, which is part of MCU firmware, programs the HW IP, handles interrupts and manages the multi-channel and multi-slice aspects of the decoding.

VCU Control Software Encoder Parameters

The Encoder reads a configuration file as an input parameter. The configuration file contains below mentioned parameters to configure the encoder, some parameter also can be set via command line, and they can be viewed with –help option of the encoder application. Parameters inside the configuration file are divided into several sections.

Input Parameters

Table 11-8: **Encoder Input Parameters**

Parameter	Description and Possible Values
YUVFile	YUV File name
Width	Frame Width in Pixels
Height	Frame Height in Pixels

Table 11-8: **Encoder Input Parameters *(Cont'd)***

Parameter	Description and Possible Values
Format	I420: YUV file contains 4:2:0 8-bit video samples stored in planar format with all picture luma (Y) samples followed by chroma samples (all U samples then all V samples) IYUV: Same as I420. YV12: Same as I420 with inverted U and V order NV12: YUV file contains 4:2:0 8-bit video samples stored in planar format with all picture luma (Y) samples followed by interleaved U and V chroma samples. I0AL: YUV file contains 4:2:0 10-bit video samples each stored in a 16-bit word in planar format with all picture luma (Y) samples followed by chroma samples (all U samples then all V samples). P010: YUV file contains 4:2:0 10-bit video samples each stored in a 16-bit word in planar format with all picture luma (Y) samples followed by interleaved U and V chroma samples. I422: YUV file contains 4:2:2 8-bit video samples stored in planar format with all picture luma (Y) samples followed by chroma samples (all U samples then all V samples). YV16: Same as I422 I2AL: YUV file contains 4:2:2 10-bit video samples each stored in a 16-bit word in planar format with all picture luma (Y) samples followed by chroma samples (all U samples then all V samples). P210: YUV file contains 4:2:2 10-bit video samples each stored in a 16-bit word in planar format with all picture luma (Y) samples followed by interleaved U and V chroma samples.
Framerate	Number of frames per second of the YUV input file. When this parameter is not present, its value is set equal to the FrameRate specified in RATE_CONTROL section. When this parameter is greater than the frame rate specified in the rate control section the rate specified in the rate control section, the encoder repeats some frames encoder drops some frames; when this parameter is lower than the frame.

Output Parameters

Table 11-9: **Encoder Output Parameters**

Parameter	Description and Possible Values
BitstreamFile	Elementary stream output file name
RecFile	Optional output file name for reconstructed picture (in YUV format). When this parameter is not present, the reconstructed YUV file is not saved.
Format	FOURCC code of the reconstructed YUV file format, see Input section for possible values

Rate Control Parameters

Table 11-10: **Encoder Rate Control Section Parameters**

Parameter	Description and Possible Values
RateCtrlMode:	selects the way the bit rate is controlled **CONST_QP**: No rate control, all pictures use the same QP defined by the SliceQP parameter. **CBR**: Use constant bit rate control. **VBR**: Use variable bit rate control. **LOW_LATENCY**: Use variable bit rate for low latency application. The IP should include the optional hardware rate control block.
BitRate	Target bit rate in Kbits/s. Not used when RateCtrlMode = CONST_QP Default value: 4000
MaxBitRate	Target bit rate in Kbits/s. Not used when RateCtrlMode = CONST_QP Default value: 4000 Notes: When **RateCtrlMode** is CBR, **MaxBitRate** shall be set to the same value as **BitRate**
FrameRate	Number of frames per second Default value: 30
SliceQP	Quantization parameter. When RateCtrlMode = CONST_QP the specified QP is applied to all slices. When RateCtrlMode = CBR the specified QP is used as initial QP. Allowed values: from 0 to 51 Default value: 30
MinQP	Minimum QP value allowed. This parameter is especially useful when using VBR rate control. In VBR rate control the value AUTO can be used to let the encoder select the MinQP according to SliceQP. Allowed values: from 0 to SliceQP Default value: 10
MaxQP	Maximum QP value allowed. Allowed values: from SliceQP to 51 Default value: 51
InitialDelay	Specifies the initial removal delay as specified in the HRD model, in seconds. Not used when RateCtrlMode = CONST_QP. Default value: 1.5
CPBSize	Specifies the size of the Coded Picture Buffer as specified in the HRD model, in seconds. Not used when RateCtrlMode = CONST_QP. Default value: 3.0
IPDelta	Specifies the QP difference between I frames and P frames. Allowed values: AUTO or positive value (≥ 0)
PBDelta	Specifies the QP difference between P frames and B frames. Allowed values: AUTO or positive value (≥ 0)
ScnChgResilience	Specifies the scene change resilience handling during encode process. Improves quality during scene changes. ENABLE/DISABLE.

Group of Pictures Parameters

Table 11-11: **Encoder GOP Section Parameters**

Parameter	Description and Possible Values
GopCtrlMode	Specifies the Group Of Pictures configuration mode. Allowed values: **DEFAULT_GOP**: IBBPBBP... (Display order) **LOW_DELAY_P**: GopPattern with a single I-Picture at the beginning followed with P-pictures only. Each P-Picture uses the picture just before as reference. IPPPP.... **LOW_DELAY_B**: GopPattern with a single I-Picture at the beginning followed with B-pictures only. Each B-Picture use the picture just before as first reference; the second reference depends on the Gop.Length parameter. IBBB...
Gop.Length	GOP length in frames including the I picture. Used only when GopCtrlMode is set to DEFAULT_GOP. Should be set to 0 for Intra only Range 0–1000. Default value: 30
Gop.NumB	Maximum number of consecutive B Frames in a GOP. Used only when GopCtrlMode is set to DEFAULT_GOP Allowed values: When GopCtrlMode is set to DEFAULT_GOP, Gop.NumB shall be in range 0 to 4.
Gop.GdrMode	When GopCtrlMode is set to LOW_DELAY_P or LOW_DELAY_B this parameter specifies whether a Gradual Decoder Refresh (GDR) scheme should be used or not. **DISABLE**: no GDR **GDR_VERTICAL**: Gradual refresh using a vertical bar moving from left to right. **GDR_HORIZONTAL**: Gradual refresh using a horizontal bar moving from top to bottom. Notes: When GDR is enabled, the Gop.Length specifies the frequency at which the refresh pattern should happen. To allow full picture refreshing, this parameter should be greater than the number of CTB/MB rows (GDR_HORIZONTAL) or columns (GDR_VERTICAL).

Settings Parameters

Table 11-12: **Encoder Settings Parameters**

Parameter	Description and Possible Values
Profile	Specifies the Profile to which the bitstream conforms Allowed values: HEVC_MAIN, HEVC_MAIN10, HEVC_MAIN_422, AVC_BASELINE, AVC_MAIN, AVC_HIGH, AVC_HIGH10, AVC_HIGH_422
Level	Specifies the Level to which the bitstream conforms Allowed values: from 1.0 to 5.1 for H.265 (HEVC), from 1.0 to 5.2 for H.264 (AVC)
Tier	Specifies the tier to which the bitstream conforms (H.265 (HEVC) only) Allowed values: MAIN_TIER, HIGH_TIER

Table 11-12: **Encoder Settings Parameters *(Cont'd)***

Parameter	Description and Possible Values
ChromaMode	Selects the chroma subsampling mode used to encode the stream Allowed values: CHROMA_MONO: The stream is encoded in Monochrome (4:0:0) CHROMA_4_2_0: The stream is encoded with 4:2:0 chroma subsampling CHROMA_4_2_2: The stream is encoded with 4:2:2 chroma subsampling
BitDepth	Specifies the bit depth of the luma and chroma samples in the encoded stream. Allowed values: 8 or 10
NumSlices	Specifies the number of slices used for each frame. Each slice contains one or more full LCU row(s) and are spread over the frame as regularly as possible. Allowed values: from 1 up to number of Coding unit rows in the frame.
SliceSize	Target Slice Size (not supported in H.264 (AVC) encoding using several cores). If set to 0, slices are defined by the NumSlices parameter. Otherwise it specifies the target slice size, in bytes, that the encoder uses to automatically split the bitstream into approximately equally-sized slices, with a granularity of one LCU. This impacts performance, adding an overhead of one LCU per slice to the processing time of a command. Allowed values: <positive value> from 100 to 65535 or 0 to disable the automatic slice splitting. Notes: This parameter is directly sent to the Encoder IP and specifies only the size of the Slice Data. It doesn't include any margin for the Slice header. So it is recommended to set the SliceSize parameter with the target value lowered by 5%. For example if your target value is 1500 bytes per slice, you should set "SliceSize = 1425" in the cfg file.
DependentSlice	When there are several slices per frame (e.g. NumSlices is greater than 1 or SliceSize is greater than 0), this parameter specifies whether the additional slice are Dependent slice segments or regular slices (H.265 (HEVC) only). Allowed values: FALSE, TRUE
EntropyMode	selects the entropy coding mode if Profile is set to AVC_MAIN, AVC_HIGH, AVC_HIGH10 or AVC_HIGH_422 (AVC only) Allowed values: MODE_CABAC: the stream is encoded with CABAC MODE_CAVLC: the stream is encoded with CAVLC
CabacInit	Specifies the CABAC initialization table index (H.264 (AVC)) / initialization flag (H.265 (HEVC)). Allowed values: from 0 to 2 (H.264 (AVC)), from 0 to 1 (H.265 (HEVC))
PicCbQpOffset	Specifies the QP offset for the first chroma channel (Cb) at picture level. (H.265 (HEVC) only) Allowed values: from -12 to +12 Default value: 0
PicCrQpOffset	specifies the QP offset for the second chroma channel (Cr) at picture level (H.265 (HEVC) only) Allowed values: from -12 to +12 Default value: 0

Table 11-12: **Encoder Settings Parameters *(Cont'd)***

Parameter	Description and Possible Values
SliceCbQpOffset	Specifies the QP offset for the first chroma channel (Cb) at slice level. Allowed values: from -12 to +12 Default value: 0
SliceCrQpOffset	specifies the QP offset for the second chroma channel (Cr) at slice level Allowed values: from -12 to +12 Default value: 0 Notes (PicCbQPOffset + SliceCbQPOffset) shall be in range -12 to +12 (PicCrQPOffset + SliceCrQPOffset) shall be in range -12 to +12
ScalingList	Specifies the scaling list mode (H.264 (AVC) and H.265 (HEVC) only). Allowed values: FLAT, DEFAULT
QpCtrlMode	Specifies how to generate the QP per CU. UNIFORM_QP: All CUs of the slice use the same QP. AUTO_QP: The QP is chosen according to the CU content using a pre-programmed lookup table. LOAD_QP: the QPs of each LCU come from an external buffer loaded from a file. The file shall be named QPs.hex and located in the working directory. The file format is described in 3.1.3
CuQpDeltaDepth	Specifies the Qp per CU granularity (H.265 (HEVC) only). Used only when QpCtrlMode is set to AUTO_QP or ADAPTIVE_AUTO_QP 0: down to 32×32, 1: down to 16×16 2: down to 8×8
ConstrainedIntraPred	Specifies the value of constrained_intra_pred_flag syntax element. Allowed values: ENABLE, DISABLE
VrtRange_P	Specifies the vertical search range used for P-frame motion estimation: Allowed values for H.265 (HEVC): 16 or 32: using 16 allows to reduce the memory bandwidth (Low Bandwidth mode) Allowed values for H.264 (AVC): 8 or 16: using 8 allows to reduce the memory bandwidth (Low Bandwidth mode)
LoopFilter	Enables/disables the deblocking filter. Allowed values: ENABLE, DISABLE
LoopFilter.CrossSlice	Enables/disables in-loop filtering across the left and upper boundaries of each slice of the frame. Used only when LoopFilter is set to ENABLE. Allowed values: ENABLE, DISABLE
LoopFilter.CrossTile	Enables/disables in-loop filtering across the left and upper boundaries of each tile of the frame. (H.265 (HEVC) only) Used only when LoopFilter is set to ENABLE. Allowed values: ENABLE, DISABLE
LoopFilter.BetaOffset	Specifies the beta offset for the deblocking filter. Used only when LoopFilter is set to ENABLE. Allowed values: from -6 to +6 Default value: -1

Table 11-12: **Encoder Settings Parameters *(Cont'd)***

Parameter	Description and Possible Values
LoopFilter.TcOffset	Specifies the Alpha_c0 offset (H.264 (AVC)) or Tc offset (H.265 (HEVC)) for the deblocking filter. Used only when Loop Filter is set to ENABLE. Allowed values: from -6 to +6 Default value: -1
CacheLevel2	The optional Encoder buffer can be used to reduce the memory bandwidth. This option can slightly reduce the video quality. This parameter specifies the maximum size of the memory used for the Encoder buffer in KB. When the value is 0 or DISABLE, the Encoder buffer is not used. When the value is too low to handle the minimum motion estimation range, the encoder fails and displays an error message. Allowed values: DISABLE or <positive value> default value: DISABLE
AspectRatio	selects the display aspect ratio of the video sequence to be written in SPS/VUI Allowed values: ASPECT_RATIO_AUTO 4:3 for common SD video, 16:9 for common HD video, unspecified for unknown format. ASPECT_RATIO_4_3 4:3 aspect ratio ASPECT_RATIO_16_9 16:9 aspect ratio ASPECT_RATIO_NONE Aspect ratio information is not present in the stream

Run Parameters

Table 11-13: **Encoder Run Section Parameters**

Parameter	Description and Possible Values
Loop	Specifies whether the encoder should loop back to the beginning of the YUV input stream when it reaches the end of the file. Allowed values: TRUE, FALSE
MaxPicture	Number of frames to encode Allowed values: integer value greater than or equal to 1, or ALL to reach the end of the YUV input stream. (ALL should not be used with Loop = TRUE, otherwise the encoder never ends).
FirstPicture	Specifies the first frame to encode. Allowed values: integer value between 0 and the number of pictures in the input YUV file.

Quantization Parameter (QP) File Format

The QP table modes are enabled by using QpCtrlMode = LOAD_QP or QpCtrlMode = LOAD_QP | RELATIVE_QP.

In this case the reference model uses the file "QPs.hex" in working directory to specify the QP values at LCU level. Each line of QPs.hex contains one 32-bit hexadecimal value, using the little endian order i.e. the LSB is the first value.

For H.264 (AVC), there is one byte per 16×16 MB (in raster scan format): absolute QP in [0;51] or relative QP in [-32;31].

For H.265 (HEVC), there is one byte per 32×32 LCU (in raster scan format): absolute QP in [0;51] or relative QP in [-32;31].

Note: Only the 6 LSBs of each byte are used for QP or Segment ID, the 2 MSBs are reserved.

For example, to specify the following relative QP table,

-1	-2	0	1	1	4
-4	-1	1	2	2	1
-1	0	-1	1	1	4
0	0	-2	2	2	6
1	-2	0	1	4	2

The QPs.hex file for H.265 (HEVC) is:

```
01003E3F
3F3C0401
01020201
013F003F
00000401
0602023E
01003E01
00000204
```

Relative QP per LCU

Xilinx VCU Control Software API

The VCU Control Software API is comprised of an encoder library and a decoder library, both in user space, which user space applications link with to control VCU hardware.

Encoder Flow

Figure 11-4 shows he typical flow of control using the VCU Control Software API.

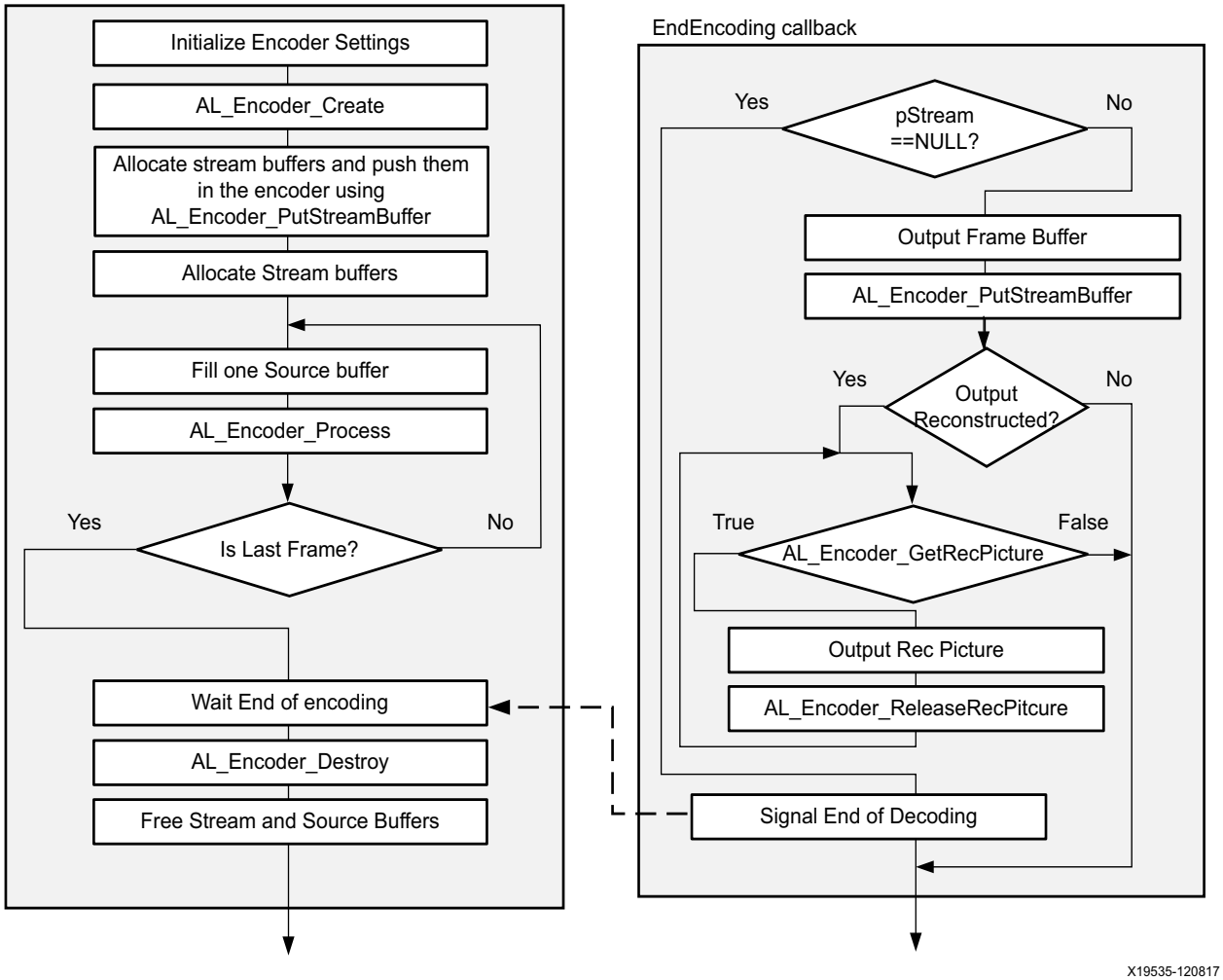


Figure 11-4: **Encoder API Workflow Example**

Encoder Block API

The Encoder block API is defined in the `lib_encoder.h` in https://github.com/Xilinx/allegro-vcu-ctrl-sw/tree/master/include/lib_encode. The API is documented below.

The example application Al_Encoder demonstrates how to use the API.

Structure AL_CB_EndEncoding

```
typedef struct
{
void (* func)(void* pUserParam,
AL_TBuffer* pStream,
AL_TBuffer const* pSrc);
void* userParam;
}AL_CB_EndEncoding;
```

This callback is called when one of the following occurs:

- A frame is encoded, i.e. ((pStream != NULL) && (pSrc != NULL))
- End of stream (EOS) reached, i.e. ((pStream == NULL) && (pSrc == NULL))
- A stream buffer us released, i.e. ((pStream != NULL && (pSrc == NULL))

Parameters

Out	pUserParam	User parameter
Out	pStream	The stream buffer if any.
Out	pSrc	The source buffer associated with the stream, if any.

Function AL_Encoder_Create

The AL_Encoder_Create function creates a new instance of the encoder and returns a handle that can be used to access the object.

```
AL_ERR AL_Encoder_Create (  AL_HEncoder *              hEnc,
                            TScheduler *               pScheduler,
                            AL_TAllocator *            pAlloc,
                            AL_TEncSettings const *    pSettings,
                            AL_CB_EndEncoding          callback )
```

Parameters

Out	AL_HEncoder *	Handle to the newly created encoder
In	pScheduler	Pointer to the scheduler object.
In	pAlloc	Pointer to an AL_TAllocator interface.
In	pSettings	Pointer to AL_TEncSettings structure specifying the encoder parameters.
In	callback	Callback called when the encoding of a frame is finished.

Returns

Returns AL_SUCCESS or an error code indicating why the encoder could not be created.

See Also

AL_Encoder_Destroy

Function AL_Encoder_Destroy

The AL_Encoder_Destroy function releases all allocated resources.

```
void AL_Encoder_Destroy (  AL_HEncoder                hEnc )
```

The AL_Encoder_Destroy function releases all allocated resources.

Parameters

Out	AL_HEncoder *	Handle to the newly created encoder
In	hEnc	Handle to Encoder object previously created with CreateEncoder.

See Also

AL_Encoder_Create

Function AL_Encoder_GetLastError

The AL_Encoder_GetLastError function returns an error code when an error has occurred during encoding, otherwise the function returns AL_SUCCESS.

```
AL_ERR AL_Encoder_GetLastError ( AL_HEncoder hEnc )
```

Parameters

In	hEnc	Handle to an encoder object

Function AL_Encoder_GetRecPicture

```
bool AL_Encoder_GetRecPicture ( AL_HEncoder hEnc,
                                TRecPic *   pRecPic )
```

Function AL_Encoder_NotifyLongTerm

The Encoder_NotifyLongTerm function informs the encoder that a long-term reference picture is used.

```
void AL_Encoder_NotifyLongTerm ( AL_HEncoder hEnc )
```

Parameters

In	hEnc	Handle to an encoder object

Function AL_Encoder_NotifySceneChange

The EncoderNotifySceneChange function informs the encoder that a scene change will happen soon.

```
void AL_Encoder_NotifySceneChange ( AL_HEncoder hEnc,
                                    int         iAhead )
```

Parameters

In	hEnc	Handle to an encoder object
In	iAhead	Number of frame until the scene change will happen. Allowed range is [0..31]

Function AL_Encoder_Process

The AL_Encoder_Process function pushes a frame buffer to the H.264 encoder. The GOP pattern determines whether or not this frame buffer can be encoded immediately.

```
bool AL_Encoder_Process (  AL_HEncoder      hEnc,
                           AL_TBuffer *     pFrame,
                           AL_TBuffer *     pQpTable )
```

Parameters

In	hEnc	hEnc Handle to an encoder object
In	pFrame	Pointer to the frame buffer to encode
In	pQpTable	Pointer to an optional qp table used if the external qp table mode is enabled

CAUTION! *The pData member of each AL_TBuffer struct pointed to by pFrame shall not be altered.*

Returns

If the function succeeds the return value is nonzero (true). If the function fails the return value is zero (false).

See Also

AL_Encoder_ReleaseFrameBuffer

Function AL_Encoder_PutStreamBuffer

The AL_Encoder_PutStreamBuffer function pushes a stream buffer in the encoder stream buffer queue. This buffer is used by the encoder to store the encoded bitstream. The buffer needs to have an associated AL_TStreamMetaData created with AL_StreamMetaData_Create (sectionNumber, uMaxSize) where uMaxSize shall be aligned on a 32-bit boundary and the sectionNumber should be set to AL_MAX_SECTION or fewer, if the number of sections expected is known.

```
bool AL_Encoder_PutStreamBuffer ( AL_HEncoder      hEnc,
                                  AL_TBuffer *     pStream )
```

Parameters

In	hEnc	Handle to an encoder object
In	pStream	Pointer to the stream buffer given to the encoder

Returns

Returns true if the buffer was successfully pushed or false if an error occurred.

Function AL_Encoder_ReleaseRecPicture

```
void AL_Encoder_ReleaseRecPicture ( AL_HEncoder      hEnc,
                                    TRecPic *        pRecPic )
```

Function AL_HEVC_Encoder_Create

```
AL_TEncCtx* AL_HEVC_Encoder_Create ( TScheduler *          pScheduler,
                                     AL_TAllocator *       pAllocator,
                                     AL_TEncSettings const * pSettings
```

Function AL_HEVC_PreprocessScalingList

```
void AL_HEVC_PreprocessScalingList ( AL_TSCLParam const *  pSclLst,
                                     TBufferEP *           pBufEP
```

Function AL_HEVC_SelectScalingList

```
void AL_HEVC_SelectScalingList ( AL_THevcSps *      pSPS,
                                 AL_TEncSettings *  pSettings
```

Function AL_HEVC_UpdatePPS

```
void AL_HEVC_UpdatePPS (  AL_THevcPps *                pPPS,
                          AL_TEncSettings const *      pSettings,
                          int                          iMaxRef,
                          int16_t                      iPpsQP
```

Function AL_HEVC_UpdateSPS

```
void AL_HEVC_UpdateSPS (  AL_THevcSps *                pSPS,
                          int                          iMaxRef,
                          int                          iCpbSize,
                          AL_TEncSettings const *      pSettings
```

Function AL_ISchedulerEnc_Destroy

```
bool AL_ISchedulerEnc_Destroy (  TScheduler *          pScheduler
```

Function AL_UpdateVPS

```
void AL_UpdateVPS (  AL_THevcVps  *                    pVPS,
                     int                               iMaxRef,
                     AL_TEncSettings  const *          pSettings
```

Function AntiEmul

```
void AntiEmu (  TStream *                              pStream,
                uint8_t const *                        pData,
                int                                    iNumBytes
```

Function FlushNAL

```
void FlushNAL (  TStream *                             pStream,
                 uint8_t                               uNUT,
                 uint8_t                               uTempID,
                 uint8_t                               uLayerID,
                 uint8_t *                             pDataInNAL,
                 int                                   iBitsInNAL,
                 bool                                  bCheckEmul,
                 uint8_t                               uNalIdc,
```

```
void FlushNAL (  TStream *          pStream,
                 bool               bAvc
```

Function InitHwRateCtrl

```
void InitHwRateCtrl (  AL_TEncSettings *          pSettings,
                       int                        iNumLCU
```

Function StreamCopyBytes

```
void StreamCopyBytes (  TStream *          pStream,
                        uint8_t *          pBuf,
                        int                iNumber )
```

Function StreamGetNumBytes

```
uint32_t StreamGetNumBytes (  TStream *          pStream
```

Function StreamInitBuffer

```
void StreamInitBuffer (  TStream *          pStream,
                         uint8_t *          pBuf,
                         uint32_t           uSize
```

Function StreamReset

```
void StreamReset (  TStream *          pStream )
```

Function StreamWriteByte

```
void StreamWriteByte (  TStream *          pStream,
                        uint8_t            uByte )
```

Decoder Flow

Figure 11-5 shows an example of using the VCU Control Software API.

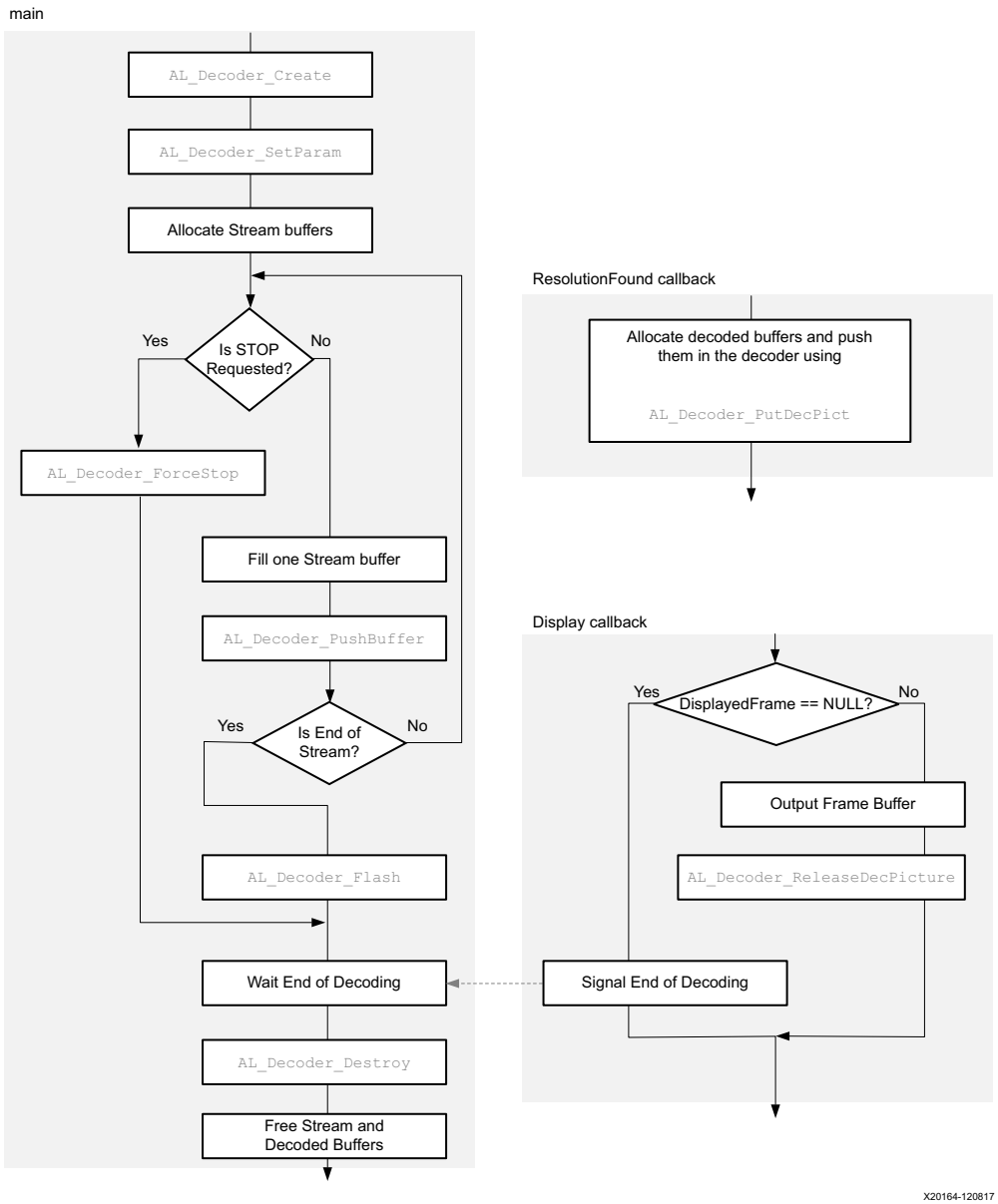


Figure 11-5: **Decoder API Workflow Example**

Decoder Block API

The Decoder block API is defined in the `lib_decoder.h` file.

Structure AL_CB_EndDecoding

```
typedef struct
{
void(*func )(AL_TBuffer *pDecodedFrame, void *pUserParam);
void *userParam;
} AL_CB_EndDecoding;
```

Callback invoked after a frame is decoded. A null frame indicates an error occurred.

See Also

AL_Decoder_GetLastError

Function AL_AVC_DecodeOneNAL

The AL_AVC_DecodeOneNAL function prepares the buffers for the hardware decoding process.

```
bool AL_AVC_DecodeOneNAL (  AL_TAvcAup *        pAUP,
                            AL_TDecCtx *        pCtx,
                            AL_ENut             eNUT,
                            bool                bIsLastAUNal,
                            bool *              bFirstIsValid,
                            bool *              bValidFirstSliceInFrame,
                            int *               iNumSlice )
```

Parameters

In	pAUP	pAUP Pointer to the current Access Unit
In	pCtx	Pointer to a decoder context object
In	eNUT	Nal Unit Type of the current NAL
In	bIsLastAUNal	Specifies if this is the last NAL of the current access unit
In, Out	bFirstIsValid	Specifies if a previous consistent slice has already been decoded
In, Out	bValidFirstSliceInFrame	Specifies if a previous consistent slice has already been decoded in the current frame
In, Out	iNumSlice	Add the number of slice in the NAL to iNumSlice

Function AL_AVC_FillPictParameters

```
void AL_AVC_FillPictParameters ( const AL_TAvcSliceHdr *    pSlice,
                                 const AL_TDecCtx *         pCtx,
                                 AL_TDecPicParam *          pPP )
```

Function AL_AVC_FillSliceParameters

```
void AL_AVC_FillSliceParameters ( const AL_TAvcSliceHdr *    pSlice,
                                  const AL_TDecCtx *         pCtx,
                                  AL_TDecSliceParam *        pSP,
                                  AL_TDecPicParam *          pPP,
                                  bool                       bConceal )
```

Function AL_AVC_FillSlicePicIdRegister

This function initializes an AVC Access Unit instance.

```
void AL_AVC_FillSlicePicIdRegister ( AL_TDecCtx *          pCtx,
                                     AL_TDecSliceParam *   pSP )
```

Function AL_AVC_InitAUP

```
void AL_AVC_InitAUP ( AL_TAvcAup *    pAUP )
```

Parameters

Out	pAUP	Pointer to the Access Unit object to be initialized

Function AL_AVC_IsSupportedProfile

```
bool AL_AVC_IsSupportedProfile ( uint8_t    profile_idc )
```

Function AL_AVC_IsVideoConfigurationCompatible

```
bool AL_AVC_IsVideoConfigurationCompatible ( AL_TVideoConfiguration *    pCfg,
                                             AL_TAvcSps *                pSPS )
```

Function AL_AVC_ParsePPS

The ParsePPS function parses a PPS NAL.

```
AL_PARSE_RESULT AL_AVC_ParsePPS (  AL_TAvcPps          pPPSTable[],
                                   AL_TRbspParser *    pRP,
                                   AL_TAvcSps          pSPSTable[]
```

Parameters

Out	pPPSTable	Pointer to the table where the parsed PPS is stored
In	pRP	Pointer to NAL parser
In	pSPSTable	Pointer to the table where the parsed SPS is stored

Function AL_AVC_ParseSEI

The ParseSEI function parses a SEI NAL.

```
bool AL_AVC_ParseSEI (  AL_TAvcSei *        pSEI,
                        AL_TRbspParser *    pRP,
                        AL_TAvcSps *        pSpsTable,
                        AL_TAvcSps **       pActiveSps
```

Parameters

Out	pSEI	Pointer to the SEI message structure that is to be filled in
In	pRP	Pointer to NAL parser
In	pSpsTable	Pointer to the SPS table
Out	pActiveSps	Pointer receiving the active sps

Returns

Returns true when all SEI messages had been parsed false otherwise.

Function AL_AVC_ParseSPS

The ParseSPS function parses an SPS NAL.

```
AL_PARSE_RESULT AL_AVC_ParseSPS (  AL_TAvcSps          pSPSTable[],
                                   AL_TRbspParser *    pRP
```

Parameters

Out	pSPSTable	Pointer to the table holding the parsed SPS
In	pRP	Pointer to NAL parser
Out	pSPSTable	Pointer to the table where the parsed are stored
In	pRP	Pointer to NAL parser

Function AL_AVC_PictMngr_CleanDPB

This function clears the decoded picture buffer (DPB).

```
void AL_AVC_PictMngr_CleanDPB (  AL_TPictMngrCtx *          pCtx )
```

Parameters

Out	pSPSTable	Pointer to the table holding the parsed SPS
In	pCtx	Pointer to a Picture Manager context object

Function AL_AVC_PictMngr_EndParsing

This function updates the Picture Manager context after a picture has been parsed.

```
void AL_AVC_PictMngr_EndParsing )  AL_TPictMngrCtx *    pCtx,
                                   bool                 bClearRef,
                                   AL_EMarkingRef       eMarkingFlag )
```

Parameters

In	pCtx	Pointer to a Picture Manager context object
In	bClearRef	Specifies if the reference pool picture is cleared
In	eMarkingFlag	Reference status of the current picture

Function AL_AVC_PictMngr_Fill_Gap_In_FrameNum

Initializes fill gap in the frame number.

```
void AL_AVC_PictMngr_Fill_Gap_In_FrameNum (  AL_TPictMngrCtx *    pCtx,
                                             AL_TAvcSliceHdr *    pSlice )
```

Parameters

In	pCtx	Pointer to a Picture Manager context object
In	pSlice	Current slice header

Function AL_AVC_PictMngr_GetBuffers

Retrieves all buffers (input and output) required to decode the current slice.

```
bool AL_AVC_PictMngr_GetBuffers (  AL_TPictMngrCtx *          pCtx,
                                   AL_TDecPicParam *          pPP,
                                   AL_TDecSliceParam *        pSP,
                                   AL_TAvcSliceHdr *          pSlice,
                                   TBufferListRef *           pListRef,
                                   TBuffer *                  pListAddr,
                                   TBufferPOC **              ppPOC,
                                   TBufferMV **               ppMV,
                                   TBuffer *                  pWP,
                                   AL_TBuffer **              ppRec,
                                   AL_EFbStorageMode          eFBStorageMode )
```

Parameters

In	pCtx	Pointer to a Picture Manager context object
In	pPP	Pointer to the current picture parameters
In	pSP	Pointer to the current slice parameters
In	pSlice	Pointer to the slice header of the current slice
In	pListRef	Pointer to the current picture reference lists
Out	pListAddr	Pointer to the buffer that receives the references, colocated POC and colocated motion vectors address list
Out	ppPOC	Receives pointer to the POC buffer where reference Pictures order count are stored.
Out	ppMV	Receives pointer to the MV buffer where Motion Vectors should be stored.
Out	pWP	Receives slices Weighted Prediction tables
Out	ppRec	Receives pointer to the frame buffer where reconstructed picture should be stored.
In	eFBStorageMode	The way frame buffer is stored

Returns

Returns true if successful and false otherwise.

Function AL_AVC_PictMngr_InitPictList

Initializes the reference picture list for the current slice.

```
void AL_AVC_PictMngr_InitPictList ( AL_TPictMngrCtx *     pCtx,
                                    AL_TAvcSliceHdr *     pSlice,
                                    TBufferListRef *      pListRef
```

Parameters

In	pCtx	Pointer to a Picture Manager context object
In	pSlice	Current slice header
Out	pListRef	Receives the reference list of the current slice

Function AL_AVC_PictMngr_ReorderPictList

Reorders the reference picture list of the current slice.

```
void AL_AVC_PictMngr_ReorderPictList ( AL_TPictMngrCtx *     pCtx,
                                       AL_TAvcSliceHdr *     pSlice,
                                       TBufferListRef *      pListRef )
```

Parameters

In	pCtx	Pointer to a Picture Manager context object
In	pSlice	Current slice header
In, Out	pListRef	Receives the modified reference list of the current slice

Function AL_AVC_PictMngr_SetCurrentPOC

Sets the picture order count (POC) of the current decoded frame.

```
bool AL_AVC_PictMngr_SetCurrentPOC ( AL_TPictMngrCtx *     pCtx,
                                     AL_TAvcSliceHdr *     pSlice )
```

Parameters

In	pCtx	Pointer to a Picture Manager context object
In	pSlice	slice header of the current decoded slice

Function AL_AVC_PictMngr_UpdateRecInfo

This function updates the reconstructed resolution information.

```
void AL_AVC_PictMngr_UpdateRecInfo (  AL_TPictMngrCtx *        pCtx,
                                      AL_TAvcSps const *       pSPS,
                                      AL_TDecPicParam *        pPP )
```

Parameters

In	pCtx	Pointer to a Picture Manager context object
In	pSPS	Pointer to a ACV SPS structure
In	pPP	Pointer to the current picture parameters

Function AL_AVC_PrepareCommand

The AL_AVC_PrepareCommand function prepares the buffers for the hardware decoding process.

```
void AL_AVC_PrepareCommand (  AL_TDecCtx *          pCtx,
                              AL_TScl *             pSCL,
                              AL_TDecPicParam *     pPP,
                              AL_TDecPicBuffers *   pBufs,
                              AL_TDecSliceParam *   pSP,
                              AL_TAvcSliceHdr *     pSlice,
                              bool                  bIsLastVclNalInAU,
                              bool                  bIsValid
```

Parameters

In	pCtx	Pointer to a decoder context object
In	pSCL	Pointer to a scaling list object
In	pPP	Pointer to the current picture parameters
In	pBufs	Pointer to the current picture buffers
In	pSP	Pointer to the current slice parameters
In	pSlice	Pointer to the current slice header
In	bIsLastVclNalInAU	Specifies if this is the last NAL of the current access unit
In	bIsValid	Specifies if the current NAL has been correctly decoded

Function AL_AVC_spic_timing

```
bool AL_AVC_spic_timing ( AL_TRbspParser *      pRP,
                          AL_TAvcSps *          pSPS,
                          AL_TAvcPicTiming *    pPicTiming )
```

Function AL_AVC_UpdateVideoConfiguration

```
void AL_AVC_UpdateVideoConfiguration ( AL_TVideoConfiguration *  pCfg,
                                       AL_TAvcSps *              pSPS )
```

Function AL_CreateDefaultDecoder

```
AL_TDecoder * AL_CreateDefaultDecoder ( AL_TIDecChannel *   pDecChannel,
                                        AL_TAllocator *     pAllocator,
                                        AL_TDecSettings *   pSettings,
                                        AL_TDecCallBacks *  pCB )
```

Function AL_Decoder_Alloc

The AL_Decoder_Alloc function allocate memory blocks usable by the decoder.

```
bool AL_Decoder_Alloc ( AL_TDecCtx *  pCtx,
                        TMemDesc *    pMD,
                        uint32_t      uSize )
```

Parameters

In	pCtx	decoder context
Out	pMD	Pointer to TMemDesc structure that receives allocated memory information
In	uSize	Number of bytes to allocate

Returns

Returns true for success and false otherwise.

Function AL_Decoder_Create()

Creates a new instance of the Decoder.

```
AL_ERR AL_Decoder_Create (  AL_HDecoder*            hDec,
                            AL_TIDecChannel*        pDecChannel,)
                            AL_TAllocator *         pAllocator,
                            AL_TDecSettings *       pSettings,
                            AL_TDecCallBacks *      pCB
```

Parameters

Out	hDec	Handle to the newly created decoder.
In	pDecChannel	Pointer to a Scheduler structure.
In	pAllocator	Pointer to an allocator handle
In	pSettings	Pointer to the decoder settings
In	pCB	Pointer to the decoder callbacks

Returns

Returns AL_SUCCESS or an error code indicating why the decoder could not be created.

See Also

AL_Decoder_Destroy

Function AL_Decoder_Destroy

The AL_Decoder_Destroy function releases all allocated resources.

```
void AL_Decoder_Destroy  (  AL_HDecoder          hDec      )
```

Parameters

Out	hDec	Handle to the newly created decoder.
In	hDec	Handle to Decoder object previously created with AL_Decoder_Create

See Also

AL_Decoder_Create

Function AL_Decoder_EndDecoding

This function performs decoded picture buffer (DPB) operations after frames decoding.

```
void AL_Decoder_EndDecoding ( void *                pUserParam,
                              AL_TDecPicStatus *    pStatus )
```

Parameters

In	pUserParam	filled with the decoder context
In	pStatus	Current frame decoded status

Function AL_Decoder_Flush

The AL_Decoder_Flush function flushes the decoding request stack when the stream parsing is finished.

```
void AL_Decoder_Flush ( AL_HDecoder hDec )
```

Parameters

In	pUserParam	filled with the decoder context
In	hDec	Handle to a decoder object.

Function AL_Decoder_FlushInput

Flushes the decoder input context (e.g. pending SC).

```
void AL_Decoder_FlushInput ( AL_HDecoder hDec )
```

Parameters

In	pUserParam	filled with the decoder context
In	hDec	Handle to a decoder object.

Function AL_Decoder_ForceStop

Notifies the decoder to stop as soon as possible. This is not a pause.

```
void AL_Decoder_ForceStop ( AL_HDecoder hDec )
```

Parameters

In	pUserParam	filled with the decoder context
In	hDec	Handle to a decoder object.

Function AL_Decoder_GetLastError

Retrieves the last decoder error state.

```
AL_ERR AL_Decoder_GetLastError ( AL_HDecoder hDec )
```

Parameters

In	pUserParam	filled with the decoder context
In	hDec	Handle to a decoder object.

Returns

Returns the current error status.

Function AL_Decoder_GetMaxBD

The AL_Decoder_GetMaxBD function retrieves the maximum bitdepth allowed by the stream profile.

```
int AL_Decoder_GetMaxBD ( AL_HDecoder hDec )
```

Parameters

In	pUserParam	filled with the decoder context
In	hDec	Handle to a decoder object.

Returns

Return the maximum bit depth allowed by the current stream profile

Function AL_Decoder_PushBuffer

Pushes a buffer to the decoder queue.

```
bool AL_Decoder_PushBuffer ( AL_HDecoder     hDec,
                             AL_TBuffer *    pBuf,
                             size_t          uSize,
                             AL_EBufMode     eMode )
```

Parameters

In	hDec	Handle to a decoder object.
In	pBuf	Pointer to the encoded bitstream buffer
In	uSize	Size in bytes of actual data in pBuf
In	eMode	Blocking strategy.

Returns

Returns the current error status.

Function AL_Decoder_PutDecPict

The AL_Decoder_PutDecPict function put the decoded picture buffer inside the decoder internal bufpool It is used to give the decoder buffers where it outputs decoded pictures.

```
void AL_Decoder_PutDecPict ( AL_HDecoder     hDec,
                             AL_TBuffer *    pDecPict
                           )
```

Parameters

In	hDec	Handle to a decoder object.
In	pDecPict	Pointer to the decoded picture buffer

Function AL_Decoder_ReleaseDecPict

Releases the decoded picture buffer.

```
void AL_Decoder_ReleaseDecPict ( AL_HDecoder     hDec,
                                 AL_TBuffer *    pDecPict
                               )
```

Parameters

In	hDec	Handle to a decoder object.
In	pDecPict	Pointer to the decoded picture buffer

Function AL_Decoder_SetParam

```
void AL_Decoder_SetParam ( AL_HDecoder     hDec,
                           bool            bConceal,
                           bool            bUseBoard,
                           int             iNumTrace,
                           int             iNumberTrace )
```

Function AL_Default_Decoder_Destroy

```
void AL_Default_Decoder_Destroy ( AL_TDecoder *     pAbsDec )
```

Function AL_Default_Decoder_Flush

```
void AL_Default_Decoder_Flush ( AL_TDecoder *     pAbsDec )
```

Function AL_Default_Decoder_ForceStop

```
void AL_Default_Decoder_ForceStop ( AL_TDecoder *     pAbsDec )
```

Function AL_Default_Decoder_GetDecPict

```
AL_TBuffer* AL_Default_Decoder_GetDecPict ( AL_TDecoder *     pAbsDec,
                                            AL_TInfoDecode *  pInfo
                                          )
```

Function AL_Default_Decoder_GetLastError

```
AL_ERR AL_Default_Decoder_GetLastError ( AL_TDecoder *     pAbsDec )
```

Function AL_Default_Decoder_GetMaxBD

```
int AL_Default_Decoder_GetMaxBD ( AL_TDecoder *  pAbsDec )
```

Function AL_Default_Decoder_GetStrOffset

```
int AL_Default_Decoder_GetStrOffset ( AL_TDecoder *  pAbsDec )
```

Function AL_Default_Decoder_InternalFlush

```
void AL_Default_Decoder_InternalFlush ( AL_TDecoder *  pAbsDec )
```

Function AL_Default_Decoder_PushBuffer

```
bool AL_Default_Decoder_PushBuffer ( AL_TDecoder *  pAbsDec,
                                     AL_TBuffer *   pBuf,
                                     size_t         uSize,
                                     AL_EBufMode    eMode
                                   )
```

Function AL_Default_Decoder_PutDecPict

```
void AL_Default_Decoder_PutDecPict ( AL_TDecoder *  pAbsDec,
                                     AL_TBuffer *   pDecPict
                                   )
```

Function AL_Default_Decoder_ReleaseDecPict

```
void AL_Default_Decoder_ReleaseDecPict ( AL_TDecoder *  pAbsDec,
                                         AL_TBuffer *   pDecPict
                                       )
```

Function AL_Default_Decoder_SetParam

```
void AL_Default_Decoder_SetParam ( AL_TDecoder *  pAbsDec,
                                   bool           bConceal,
                                   bool           bUseBoard,
                                   int            iFrmID,
                                   int            iNumFrm
                                 )
```

Function AL_Default_Decoder_TryDecodeOneAU

This function performs the decoding of one AU.

```
AL_ERR AL_Default_Decoder_TryDecodeOneAU ( AL_TDecoder *     pAbsDec,
                                           TCircBuffer *     pBufStream
                                         )
```

Parameters

In	pAbsDec	decoder handle
In	pBufStream	circular buffer containing input bitstream to decode

Returns

Returns the error status.

Function AL_Dpb_AVC_Cleanup

Remove pictures and references that are no longer needed from the DPB.

```
void AL_Dpb_AVC_Cleanup ( AL_TDpb *     pDpb )
```

Parameters

In	pAbsDec	decoder handle
In	pDpb	Pointer to a DPB context object

Function AL_Dpb_BeginNewSeq

This function must be called after each DPB flushing.

```
void AL_Dpb_BeginNewSeq ( AL_TDpb *     pDpb )
```

Parameters

In	pAbsDec	decoder handle
In	pDpb	Pointer to a DPB context object

Function AL_Dpb_ClearOutput

Clear output pictures from the DPB.

```
void AL_Dpb_ClearOutput ( AL_TDpb *  pDpb )
```

Parameters

In	pAbsDec	decoder handle
In, Out	pDpb	Pointer to a DPB context object

Function AL_Dpb_ConvertPicIDToNodeID

Converts a PicID index to a NodeID index.

```
uint8_t AL_Dpb_ConvertPicIDToNodeID ( AL_TDpb const *  pDpb,
                                      uint8_t          uPicID
                                    )
```

Parameters

In	pDpb	Pointer to a DPB context object
In	uPicID	Index to be converted

Returns

Returns the converted NodeID.

Function AL_Dpb_DecrementPicLatency

Decrements the latency of the specified picture.

```
void AL_Dpb_DecrementPicLatency ( AL_TDpb *  pDpb,
                                  uint8_t    uNode
                                )
```

Parameters

In, Out	pDpb	Pointer to a DPB context object
In	uNode	Picture identifier in the DPB Node

Function AL_Dpb_Deinit

Uninitializes the specified DPB context object.

```
void AL_Dpb_Deinit ( AL_TDpb * pDpb )
```

Parameters

In, Out	pDpb	Pointer to a DPB context object
In, Out	pDpb	Pointer to a DPB context object

Function AL_Dpb_Display

Adds the specified picture in the display list.

```
void AL_Dpb_Display ( AL_TDpb * pDpb,
                      uint8_t uNode
```

Parameters

In, Out	pDpb	Pointer to a DPB context object
In	uNode	Index of the node to remove

Returns

Returns the Frame buffer index of the removed node

Function AL_Dpb_EndDecoding

Updates DPB state after a frame decoding.

```
void AL_Dpb_EndDecoding ( AL_TDpb * pDpb,
                          int iFrmID
                        )
```

Parameters

In	pDpb	Pointer to a DPB context object
In	iFrmID	FrmID of the decoded frame

Function AL_Dpb_FillList

Fills the poc list buffer with their respective reference marking status.

```
void AL_Dpb_FillList  (  AL_TDpb *          pDpb,
                         uint8_t            uL0L1,
                         TBufferListRef *   pListRef,
                         int *              pPocList,
                         uint32_t *         pLongTermList
                      )
```

Parameters

In	pDpb	Pointer to a DPB context object
In	uL0L1	Reference direction
In	pListRef	Reference list
Out	pPocList	Poc list output buffer
Out	pLongTermList	Reference picture marking status output buffer

Returns

The node indexes with the given poc_lsb.

Function AL_Dpb_Flush

Flush remaining pictures from the DPB.

```
void AL_Dpb_Flush  (  AL_TDpb *  pDpb  )
```

Parameters

In	pDpb	Pointer to a DPB context object
In, Out	pDpb	Pointer to a DPB context object

Function AL_Dpb_GetDisplayBuffer

Returns the frame buffer index of the next picture to be displayed or UndefID.

```
uint8_t AL_Dpb_GetDisplayBuffer ( AL_TDpb *  pDpb )
```

Parameters

In	pDpb	Pointer to a DPB context object
In, Out	pDpb	Pointer to a DPB context object

Returns

Returns the frame buffer index of the next picture to be displayed or UndefID.

Function AL_Dpb_GetFifoLast

This function gets the last available picture for output index.

```
uint8_t AL_Dpb_GetFifoLast ( AL_TDpb *  pDpb )
```

Parameters

In	pDpb	Pointer to a DPB context object
In	pDpb	Pointer to a DPB context object

Function AL_Dpb_GetFrmID_FromFifo

This function gets the frame ID of a specific picture in the Display FIFO.

```
uint8_t AL_Dpb_GetFrmID_FromFifo ( AL_TDpb *  pDpb,
                                   uint8_t    uID
```

Parameters

In	uID	Picture identifier in the Display FIFO
In	pDpb	Pointer to a DPB context object

Returns

Returns the picture's FrmID.

Function AL_Dpb_GetFrmID_FromNode

This function gets the frame ID of a specific picture in the DPB Nodes.

```
uint8_t AL_Dpb_GetFrmID_FromNode ( AL_TDpb *    pDpb,
                                   uint8_t      uNode
                                 )
```

Parameters

In	pDpb	Pointer to a DPB context object
In	uNode	Picture identifier in the DPB Node

Returns

Returns the picture's FrmID.

Function AL_Dpb_GetHeadPOC

Returns the Node ID of the picture with the smallest POC value.

```
uint8_t AL_Dpb_GetHeadPOC ( AL_TDpb *    pDpb )
```

Parameters

In	pDpb	Pointer to a DPB context object
In	pDpb	Pointer to a DPB context object

Returns

Returns the Node ID of the picture with the smallest POC value.

Function AL_Dpb_GetLastPicID

Returns the Pic ID of the last inserted frame or 0xFF if the DPB is empty.

```
uint8_t AL_Dpb_GetLastPicID ( AL_TDpb *    pDpb )
```

Parameters

In	pDpb	Pointer to a DPB context object
In	pDpb	Pointer to a DPB context object

Returns

Returns the Pic ID of the last inserted frame or 0xFF if the DPB is empty.

Function AL_Dpb_GetMarkingFlag

This function retrieves the reference status of a specific picture.

```
uint8_t AL_Dpb_GetMarkingFlag ( AL_TDpb *  pDpb,
                                uint8_t    uNode
                              )
```

Parameters

In	pDpb	Pointer to a DPB context object
In	uNode	Picture identifier in the DPB Node

Returns

Returns the reference status of a specific picture.

Function AL_Dpb_GetMvID_FromNode

This function gets the frame ID of a specific picture in the DPB Nodes.

```
uint8_t AL_Dpb_GetMvID_FromNode ( AL_TDpb *  pDpb,
                                  uint8_t    uNode
                                )
```

Parameters

In	pDpb	Pointer to a DPB context object
In	uNode	Picture identifier in the DPB Node

Returns

Returns the picture's FrmID.

Function AL_Dpb_GetNextFreeNode

Gets the first free node in the list by arrival order.

```
uint8_t AL_Dpb_GetNextFreeNode ( AL_TDpb *  pDpb )
```

Parameters

In	pDpb	Pointer to a DPB context object
In, Out	pDpb	Pointer to a DPB context object

Returns

Returns the first free node index

Function AL_Dpb_GetNextPOC

This function retrieves the Node ID associated with picture which follows the current picture in POC order.

```
uint8_t AL_Dpb_GetNextPOC ( AL_TDpb *  pDpb,
                            uint8_t    uNode
                          )
```

Parameters

In	pDpb	Pointer to a DPB context object
In	uNode	Current picture identifier in the DPB Node

Returns

Returns the Node ID of the picture which follows the current picture in POC order.

Function AL_Dpb_GetNumOutputPict

This function retrieves the number of picture needed for output.

uint8_t AL_Dpb_GetNumOutputPict (**AL_TDpb** * pDpb)

Parameters

In	pDpb	Pointer to a DPB context object
In	pDpb	Pointer to a DPB context object

Returns

Returns the number of picture needed for output.

Function AL_Dpb_GetNumPic

This function gets the number of managed pictures.

uint8_t AL_Dpb_GetNumPic (AL_TDpb * pDpb)

Parameters

In	pDpb	Pointer to a DPB context object
In, Out	pDpb	Pointer to a DPB context object

Returns

Returns the number of pictures managed by the DPB.

Function AL_Dpb_GetNumRef

This function gets the number of managed references.

uint8_t AL_Dpb_GetNumRef (AL_TDpb * pDpb)

Parameters

In	pDpb	Pointer to a DPB context object
In, Out	pDpb	Pointer to a DPB context object

Returns

Returns the number of references managed by the DPB.

Function AL_Dpb_GetOutputFlag

This function retrieves the pic_output_flag of a specific picture.

```
uint8_t AL_Dpb_GetOutputFlag ( AL_TDpb *    pDpb,
                               uint8_t      uNode
```

Parameters

In	pDpb	Pointer to a DPB context object
In	uNode	Picture identifier in the DPB Node

Returns

Returns the picture's output flag.

Function AL_Dpb_GetPicCount

This function retrieves the number of picture present in the DPB.

```
uint8_t AL_Dpb_GetPicCount ( AL_TDpb *    pDpb )
```

Parameters

In	pDpb	Pointer to a DPB context object
In	pDpb	Pointer to a DPB context object

Returns

Returns the number of picture present in the DPB.

Function AL_Dpb_GetPicID_FromNode

This function gets the Pic ID of a specific picture in the DPB Nodes.

```
uint8_t AL_Dpb_GetPicID_FromNode ( AL_TDpb *    pDpb,
                                   uint8_t      uNode
                                 )
```

Parameters

In	pDpb	Pointer to a DPB context object
In	uNode	Picture identifier in the DPB Node

Returns

Return the picture's PicID.

Function AL_Dpb_GetPicLatency_FromFifo

This function gets the latency of a specific picture in the Display FIFO.

```
uint32_t AL_Dpb_GetPicLatency_FromFifo ( AL_TDpb *    pDpb,
                                         uint8_t      uFrmID
                                       )
```

Parameters

In	pDpb	Pointer to a DPB context object
In	uFrmID	Picture identifier in the Display fifo

Returns

Return the picture's latency.

Function AL_Dpb_GetPicLatency_FromNode

This function gets the latency of a specific picture in the DPB Nodes.

```
uint32_t AL_Dpb_GetPicLatency_FromNode ( AL_TDpb *      pDpb,
                                         uint8_t        uNodeID
                                       )
```

Parameters

In	pDpb	Pointer to a DPB context object
In	uNodeID	Picture identifier in the DPB Node

Returns

Return the picture's latency.

Function AL_Dpb_GetRefCount

This function retrieves the number of references present in the DPB.

```
uint8_t AL_Dpb_GetRefCount ( AL_TDpb *      pDpb )
```

Parameters

In	pDpb	Pointer to a DPB context object
In	pDpb	Pointer to a DPB context object

Returns

Returns the number of references present in the DPB.

Function AL_Dpb_HEVC_Cleanup

Removes from the DPB any pictures that are no longer needed.

```
void AL_Dpb_HEVC_Cleanup ( AL_TDpb *      pDpb,
                           uint32_t       uMaxLatency,
                           uint8_t        uMaxOutput
                         )
```

Parameters

In, Out	pDpb	Pointer to a DPB context object

In	uMaxLatency	Maximum DPB latency for a picture
In	uMaxOutput	Maximum number of picture that need to be held by the DPB

Function AL_Dpb_IncrementPicLatency

Increments the latency of a specific picture when it follows the current picture in output order.

```
void AL_Dpb_IncrementPicLatency ( AL_TDpb *     pDpb,
                                  uint8_t       uNode,
                                  int           iCurFramePOC
                                )
```

Parameters

In, Out	pDpb	Pointer to a DPB context object
In	uNode	Picture identifier in the DPB Node
In	iCurFramePOC	Picture order count of the current picture

Function AL_Dpb_Init

Initializes the specified DPB context object.

```
void AL_Dpb_Init ( AL_TDpb *                pDpb,
                   uint8_t                  uNumRef,
                   uint8_t                  uNumInterBuf,
                   void *                   pPfnCtx,
                   PfnIncrementFrmBuf       pfnIncrementFrmBuf,
                   PfnReleaseFrmBuf         pfnReleaseFrmBuf,
                   PfnOutputFrmBuf          pfnOutputFrmBuf,
                   PfnIncrementMvBuf        pfnIncrementMvBuf,
                   PfnReleaseMvBuf          pfnReleaseMvBuf
                 )
```

Parameters

In, Out	pDpb	Pointer to a DPB context object
In	uNumRef	Number of reference to manage
In	uNumInterBuf	Number of intermediate buffers to manage
In	pPfnCtx	User callback parameter
In	pfnIncrementFrmBuf	User Callback increasing access count on frame buffer

In	pfnReleaseFrmBuf	User Callback decreasing access count on a frame buffer
In	pfnOutputFrmBuf	User Callback managing frame buffer output
In	pfnIncrementMvBuf	User Callback increasing access count on motion-vector buffer
In	pfnReleaseMvBuf	User Callback releasing a motion vector reference buffer

Function AL_Dpb_InitBSlice_RefList

Initializes the reference list for a B slice.

```
void AL_Dpb_InitBSlice_RefList ( AL_TDpb *          pDpb,
                                 int                iCurFramePOC,
                                 TBufferListRef *   pRefList
                               )
```

Parameters

In	pDpb	Pointer to a DPB context object
In	iCurFramePOC	POC of the current picture
Out	pRefList	Pointer on the reference picture list object

Function AL_Dpb_InitPSlice_RefList

Initializes the reference list for a P slice.

```
void AL_Dpb_InitPSlice_RefList ( AL_TDpb *          pDpb,
                                 TBufferRef *       pRefList
                               )
```

Parameters

In	pDpb	Pointer to a DPB context object
Out	pRefList	Pointer on the reference picture list object

Function AL_Dpb_Insert

Insert a new frame buffer in a reference buffer pool.

```
void AL_Dpb_Insert  (  AL_TDpb *          pDpb,
                       int                iFramePOC,
                       uint32_t           uPocLsb,
                       uint8_t            uNode,
                       uint8_t            uFrmID,
                       uint8_t            uMvID,
                       uint8_t            pic_output_flag,
                       AL_EMarkingRef     eMarkingFlag,
                       uint8_t            uNonExisting,
                       AL_ENut            eNUT
                    )
```

Parameters

In, Out	pDpb	Pointer to a DPB context object
In	iFramePOC	Picture order count of the added frame buffer
In	uPocLsb	Value used to identify long term reference picture
In	uNode	Node index of the added reference
In	uFrmID	Frame Buffer index of the added reference
In	uMvID	Associated motion-vector buffer index of the added reference
In	pic_output_flag	Specifies whether the current picture is displayed or not
In	eMarkingFlag	Added reference status
In	uNonExisting	Added non existing status
In	eNUT	Added NAL Unit Type

Function AL_Dpb_LastHasMMCO5

Checks if the last picture has a MMCO 5 opcode.

```
bool AL_Dpb_LastHasMMCO5 ( AL_TDpb *  pDpb )
```

Parameters

In, Out	pDpb	Pointer to a DPB context object
In, Out	pDpb	Pointer to a DPB context object

Returns

Returns true if the last picture has an MMCO 5 opcode false otherwise

Function AL_Dpb_MarkingProcess

Updates the reference status of the pictures present in the DPB.

```
void AL_Dpb_MarkingProcess ( AL_TDpb *  pDpb,
                             AL_TAvcSliceHdr *  pSlice
                           )
```

Parameters

In	pDpb	Pointer to a DPB context object
In	pSlice	Current slice header

Function AL_Dpb_ModifLongTerm

Modifies the reference picture list on long-term reference pictures.

```
void AL_Dpb_ModifLongTerm ( AL_TDpb *  pDpb,
                            AL_TAvcSliceHdr *  pSlice,
                            uint8_t  uOffset,
                            int  iL0L1,
                            uint8_t *  pRefIdx,
                            TBufferListRef *  pListRef
                          )
```

Parameters

In	pDpb	Pointer to a DPB context object
In	pSlice	Current slice header

In	uOffset	Number of modification processed on short-term reference picture
In	iL0L1	Reference list ID
In, Out	pRefIdx	Reference index of the current modified picture
In, Out	pListRef	Pointer to the reference picture list object

Function AL_Dpb_ModifShortTerm

Modifies the reference picture list on short-term reference pictures.

```
void AL_Dpb_ModifShortTerm ( AL_TDpb *              pDpb,
                             AL_TAvcSliceHdr *      pSlice,
                             int                    iPicNumIdc,
                             uint8_t                uOffset,
                             int                    iL0L1,
                             uint8_t *              pRefIdx,
                             int *                  pPicNumPred,
                             TBufferListRef *       pListRef
                           )
```

Parameters

In	pDpb	Pointer to a DPB context object
In	pSlice	Current slice header
In	iPicNumIdc	Picture reordering opcode
In	uOffset	Number of modification processed on short-term reference picture
In	iL0L1	Reference list ID
In, Out	pRefIdx	reference index of the current modified picture
In	pPicNumPred	Pic Num of the processed picture without wrapping
In, Out	pListRef	Pointer on the reference picture list object

Function AL_Dpb_NodeIsReset

Checks if the Node identified by uNode has been reset.

```
bool AL_Dpb_NodeIsReset ( AL_TDpb *      pDpb,
                          uint8_t        uNode
                        )
```

Parameters

In	pDpb	Pointer to a DPB context object
In	uNode	Node identifier

Returns

Returns true if the node has been reseted false otherwise.

Function AL_Dpb_PictNumberProcess

Calculate the pic_num of each reference picture.

```
void AL_Dpb_PictNumberProcess ( AL_TDpb *          pDpb,
                                AL_TAvcSliceHdr *  pSlice
                              )
```

Parameters

In	pDpb	Pointer to a DPB context object
In	pSlice	Current slice header

Function AL_Dpb_ReleaseDisplayBuffer

This function releases a picture index previously obtained through DPB_GetDisplayBuffer.

```
uint8_t AL_Dpb_ReleaseDisplayBuffer ( AL_TDpb *  pDpb )
```

Parameters

In	pDpb	Pointer to a DPB context object
In, Out	pDpb	Pointer to a DPB context object

Returns

Returns the frame index of the released picture.

Function AL_Dpb_Remove

Remove the specified node from the reference buffer pool.

```
uint8_t AL_Dpb_Remove ( AL_TDpb *  pDpb,
                        uint8_t    uNode
                      )
```

Parameters

In, Out	pDpb	Pointer to a DPB context object
In	uNode	Index of the node to remove

Returns

Returns the frame buffer identifier of the deleted picture.

Function AL_Dpb_RemoveHead

Remove the First Node (Decoding order) from the pool.

```
uint8_t AL_Dpb_RemoveHead ( AL_TDpb *  pDpb )
```

Parameters

In, Out	pDpb	Pointer to a DPB context object
In, Out	pDpb	Pointer to a DPB context object

Returns

Returns the frame buffer identifier of the deleted picture.

Function AL_Dpb_ResetMMCO5

Takes into account the non-presence of the MMCO 5 opcode.

```
void AL_Dpb_ResetMMCO5 ( AL_TDpb *  pDpb )
```

Parameters

In, Out	pDpb	Pointer to a DPB context object
In, Out	pDpb	Pointer to a DPB context object

Function AL_Dpb_ResetOutputFlag

This function removes a picture from the picture list needed for output.

```
void AL_Dpb_ResetOutputFlag ( AL_TDpb *     pDpb,
                              uint8_t       uNode
                            )
```

Parameters

In, Out	pDpb	Pointer to a DPB context object
In	uNode	Picture identifer in the DPB Node

Function AL_Dpb_SearchPOC

Searches the picture with the given iPOC in the DPB excluding unreferenced nodes.

```
uint8_t AL_Dpb_SearchPOC ( AL_TDpb *     pDpb,
                           int           iPOC )
```

Parameters

In	pDpb	Pointer to a DPB context object
In	iPOC	Picture order count value to search in the DPB

Returns

The node indexes with the given iPOC or the end of the node list, if iPOC is not found.

Function AL_Dpb_SearchPocLsb

Searches the picture with the given poc_lsb in the DPB with the corresponding marking flag.

```
uint8_t AL_Dpb_SearchPocLsb ( AL_TDpb *     pDpb,
                              uint32_t      poc_lsb
                            )
```

Parameters

In	pDpb	Pointer to a DPB context object
In	poc_lsb	poc_lsb value to search in the DPB

Returns

Returns the node index with the given poc_lsb.

Function AL_Dpb_SetMarkingFlag

This function sets the reference status of a specific picture.

```
void AL_Dpb_SetMarkingFlag ( AL_TDpb *          pDpb,
                             uint8_t            uNode,
                             AL_EMarkingRef     eMarkingFlag
                           )
```

Parameters

In, Out	pDpb	Pointer to a DPB context object
In	uNode	Picture identifer in the DPB Node
In	eMarkingFlag	Reference status to apply to the picture

Function AL_Dpb_SetMMCO5

Takes into account the presence of the MMCO 5 opcode.

```
void AL_Dpb_SetMMCO5 ( AL_TDpb * pDpb )
```

Parameters

In, Out	pDpb	Pointer to a DPB context object
In, Out	pDpb	Pointer to a DPB context object

Function AL_Dpb_SetNumRef

This function must be called after each DPB flushing.

```
void AL_Dpb_SetNumRef ( AL_TDpb *          pDpb,
                        uint8_t            uMaxRef
                      )
```

Parameters

In, Out	pDpb	Pointer to a DPB context object
In	uMaxRef	Number of reference to be managed

Function AL_Dpb_Terminate

Flush last DPB removal orders.

```
void AL_Dpb_Terminate ( AL_TDpb * pDpb )
```

Parameters

In, Out	pDpb	Pointer to a DPB context object
In, Out	pDpb	Pointer to a DPB context object

Function AL_HEVC_DecodeOneNAL

The HEVC_DecodeOneNAL function prepares the buffers for the hardware decoding process.

```
bool AL_HEVC_DecodeOneNAL ( AL_THevcAup *  pAUP,
                            AL_TDecCtx *   pCtx,
                            AL_ENut        eNUT,
                            bool           bIsLastAUNal,
                            bool *         bFirstIsValid,
                            bool *         bValidFirstSliceInFrame,
                            int *          iNumSlice,
                            bool *         bBeginFrameIsValid
                          )
```

Parameters

In	pAUP	pAUP Pointer to the current Access Unit
In	pCtx	Pointer to a decoder context object
In	eNUT	NAL Unit Type of the current NAL
In	bIsLastAUNal	Specifies if this is the last NAL of the current access unit
In, Out	bFirstIsValid	Specifies if a previous consistent slice has already been decoded
In, Out	bValidFirstSliceInFrame	Specifies if a previous consistent slice has already been decoded in the current frame
In, Out	iNumSlice	Add the number of slice in the NAL to iNumSlice
In, Out	bBeginFrameIsValid	The begin frame is valid.

Function AL_HEVC_DeinitSEI

The AL_HEVC_DeinitSEI function releases memory allocation operated by the SEI structure.

```
void AL_HEVC_DeinitSEI ( AL_THevcSei * pSEI )
```

Parameters

In	pAUP	pAUP Pointer to the current Access Unit
Out	pSEI	Pointer to the SEI structure that is to be freed

Function AL_HEVC_FillPictParameters

```
void AL_HEVC_FillPictParameters ( const AL_THevcSliceHdr * pSlice,
                                  const AL_TDecCtx * pCtx,
                                  AL_TDecPicParam * pPP
                                )
```

Function AL_HEVC_FillSliceParameters

```
void AL_HEVC_FillSliceParameters ( const AL_THevcSliceHdr * pSlice,
                                   const AL_TDecCtx * pCtx,
                                   AL_TDecSliceParam * pSP,
                                   bool bConceal
                                 )
```

Function AL_HEVC_FillSlicePicIdRegister

```
void AL_HEVC_FillSlicePicIdRegister ( const AL_THevcSliceHdr * pSlice,
                                      AL_TDecCtx * pCtx,
                                      AL_TDecPicParam * pPP,
                                      AL_TDecSliceParam * pSP
                                    )
```

Function AL_HEVC_InitAUP

This function initializes an HEVC Access Unit instance.

```
void AL_HEVC_InitAUP ( AL_THevcAup * pAUP )
```

Parameters

Out	pAUP	Pointer to the Access Unit object to be initialized

Function AL_HEVC_InitSEI

The InitSEI function initializes a SEI structure. The HEVC_InitSEI function initializes a SEI structure.

```
void AL_HEVC_InitSEI ( AL_THevcSei * pSEI )
```

Parameters

Out	pSEI	Pointer to the SEI structure that to be initialized

Function AL_HEVC_IsVideoConfigurationCompatible

```
bool AL_HEVC_IsVideoConfigurationCompatible ( AL_TVideoConfiguration * pCfg,
                                              AL_THevcSps * pSPS )
```

Function AL_HEVC_ParsePPS

The ParsePPS function parses a PPS NAL.

```
void AL_HEVC_ParsePPS ( AL_THevcPps pPPSTable[],
                        AL_TRbspParser * pRP,
                        AL_THevcSps pSPSTable[],
                        uint8_t * pPpsId )
```

Parameters

Out	pPPSTable	Pointer to the table where the parsed pps are stored
In	pRP	Pointer to NAL parser
In	pSPSTable	Pointer to the table where the parsed sequence parameter sets (SPS) are stored
Out	pPpsId	Pointer to a variable that receive the PPS ID

Function AL_HEVC_ParseSEI

The HEVC_ParseSEI function parses a SEI NAL.

```
bool AL_HEVC_ParseSEI (  AL_THevcSei *          pSEI,
                         AL_TRbspParser *       pRP,
                         AL_THevcSps *          pSpsTable,
                         AL_THevcSps **         pActiveSps )
```

Parameters

Out	pSEI	Pointer to the SEI message structure that is to be filled
In	pRP	Pointer to NAL parser
In	pSpsTable	Pointer to the SPS table
Out	pActiveSps	Pointer receiving the active SPS

Returns

Returns true when all SEI messages had been parsed false otherwise.

Function AL_HEVC_ParseSPS

The ParseSPS function parses a SPS NAL.

```
AL_PARSE_RESULT AL_HEVC_ParseSPS (‘  AL_THevcSps          pSPSTable[],
                                     AL_TRbspParser *    pRP,
                                     uint8_t *           SpsId
```

Parameters

Out	pSPSTable	Pointer to the table holding the parsed SPS
In	pRP	Pointer to NAL parser
Out	SpsId	ID of the SPS

Function AL_HEVC_PictMngr_BuildPictureList

Builds the reference picture list of the current slice.

```
bool AL_HEVC_PictMngr_BuildPictureList ( AL_TPictMngrCtx *      pCtx,
                                         AL_THevcSliceHdr *     pSlice,
                                         TBufferListRef *       pListRef
```

Parameters

In	pCtx	Pointer to a Picture Manager context object
In	pSlice	Pointer to the slice header of the current slice
Out	pListRef	Pointer to the current reference list

Function AL_HEVC_PictMngr_ClearDPB

Removes from the DPB all pictures that are unreferenced and not needed for output.

```
void AL_HEVC_PictMngr_ClearDPB ( AL_TPictMngrCtx *      pCtx,
                                 AL_THevcSps *          pSPS,
                                 bool                   bClearRef,
                                 bool                   bNoOutputPrior )
```

Parameters

In	pCtx	Pointer to a Picture Manager context object
In	pSPS	Pointer to the Sequence Parameter Set structure holding info on picture DPB latency
In	bClearRef	Specifies if the reference pool picture is cleared
In	bNoOutputPrior	Specifies if the pictures stored in the DPB is to is be output or discarded when bClearRef is true

Function AL_HEVC_PictMngr_EndFrame

This function updates the Picture Manager context each time a picture have been decoded.

```
void AL_HEVC_PictMngr_EndFrame (  AL_TPictMngrCtx *      pCtx,
                                  uint32_t               uPocLsb,
                                  AL_ENut                eNUT,
                                  AL_THevcSliceHdr *     pSlice,
                                  uint8_t                pic_output_flag )
```

Parameters

In	pCtx	Pointer to a Picture Manager context object
In	uPocLsb	Value used to identify long term reference picture
In	eNUT	NalUnitType of the current decoded picture
In	pSlice	Pointer to the last slice header current's frame
In	pic_output_flag	Specifies whether the current picture is displayed or not

Function AL_HEVC_PictMngr_GetBuffers

Retrieves all buffers (input and output) required to decode the current slice.

```
bool AL_HEVC_PictMngr_GetBuffers (  AL_TPictMngrCtx *          pCtx,
                                    AL_TDecPicParam *          pPP,
                                    AL_TDecSliceParam *        pSP,
                                    AL_THevcSliceHdr *         pSlice,
                                    TBufferListRef *           pListRef,
                                    TBuffer *                  pListAddr,
                                    TBufferPOC **              ppPOC,
                                    TBufferMV **               ppMV,
                                    TBuffer *                  pWP,
                                    AL_TBuffer **              ppRec,
                                    AL_EFbStorageMode          eFBStorageMode )
```

Parameters

In	pCtx	Pointer to a Picture Manager context object
In	pPP	Pointer to the current picture parameters
In	pSP	Pointer to the current slice parameters
In	pSlice	Pointer to the slice header of the current slice
In	pListRef	Pointer to the current picture reference lists
Out	pListAddr	Pointer to the buffer that receives the references, colocated POC and colocated motion vectors address list
Out	ppPOC	Receives pointer to the POC buffer where reference Pictures order count are stored.
Out	ppMV	Receives pointer to the MV buffer where Motion Vectors should be stored.
Out	pWP	Receives slices Weighted Pred tables
Out	ppRec	Receives pointer to the frame buffer where reconstructed picture should be stored.
In	eFBStorageMode	the way frame buffer is stored

Returns

Returns true on success and false otherwise.

Function AL_HEVC_PictMngr_HasPictInDPB

This function returns true if the DPB has reference.

```
bool AL_HEVC_PictMngr_HasPictInDPB ( AL_TPictMngrCtx *  pCtx )
```

Parameters

In	pCtx	Pointer to a Picture Manager context object
In	pCtx	Pointer to a Picture Manager context object

Function AL_HEVC_PictMngr_InitRefPictSet

Prepares the reference picture set for the current slice reference picture list construction.

```
void AL_HEVC_PictMngr_InitRefPictSet ( AL_TPictMngrCtx *  pCtx,
                                       AL_THevcSliceHdr *  pSlice )
```

Parameters

In	pCtx	Pointer to a Picture Manager context object
In	pSlice	Pointer to the slice header of the current slice

Function AL_HEVC_PictMngr_RemoveHeadFrame

This function removes from the DPB the oldest picture if it is full.

```
void AL_HEVC_PictMngr_RemoveHeadFrame ( AL_TPictMngrCtx *  pCtx )
```

Parameters

In	pCtx	Pointer to a Picture Manager context object

Function AL_HEVC_PictMngr_UpdateRecInfo

This function updates the reconstructed resolution information.

```
void AL_HEVC_PictMngr_UpdateRecInfo ( AL_TPictMngrCtx *  pCtx,
                                      AL_THevcSps *  pSPS,
                                      AL_TDecPicParam *  pPP )
```

Parameters

In	pCtx	Pointer to a Picture Manager context object

In	pSPS	Pointer to a HECV SPS structure
In	pPP	Pointer to the current picture parameters

Function AL_HEVC_PrepareCommand

The AL_HEVC_PrepareCommand function prepares the buffers for the hardware decoding process.

```
void AL_HEVC_PrepareCommand (  AL_TDecCtx *          pCtx,
                               AL_TScl *             pSCL,
                               AL_TDecPicParam *     pPP,
                               AL_TDecPicBuffers *   pBufs,
                               AL_TDecSliceParam *   pSP,
                               AL_THevcSliceHdr *    pSlice,
                               bool                  bIsLastVclNalInAU,
                               bool                  bIsValid )
```

Parameters

In	pCtx	Pointer to a decoder context object
In	pSCL	Pointer to a scaling list object
In	pPP	Pointer to the current picture parameters
In	pBufs	Pointer to the current picture buffers
In	pSP	Pointer to the current slice parameters
In	pSlice	Pointer to the current slice header
In	bIsLastVclNalInAU	Specifies if this is the last NAL of the current access unit
In	bIsValid	Specifies if the current NAL has been correctly decoded

Function AL_HEVC_short_term_ref_pic_set

Computes the short-term reference picture set.

```
void AL_HEVC_short_term_ref_pic_set (  AL_THevcSps *      pSPS,
                                       uint8_t            RefIdx,
                                       AL_TRbspParser *   pRP )
```

Parameters

Out	pSPS	Pointer to the SPS structure containing the ref_pic_set structure and variables
In	RefIdx	Idx of the current ref_pic_set
In	pRP	Pointer to NAL parser

Function AL_HEVC_UpdateVideoConfiguration

```
void AL_HEVC_UpdateVideoConfiguration ( AL_TVideoConfiguration *  pCfg,
                                        AL_THevcSps *             pSPS )
```

Function AL_InitFrameBuffers

The AL_InitFrameBuffers function initializes the frame buffers needed to process the current frame decoding.

```
void AL_InitFrameBuffers ( AL_TDecCtx *      pCtx,
                           int               iWidth,
                           int               iHeight,
                           AL_TDecPicParam * pPP )
```

Parameters

In	pCtx	Pointer to a decoder context object
In	iWidth	Picture width in pixels unit
In	iHeight	Picture height in pixels unit
In	pPP	Pointer to the current picture parameters

Function AL_InitIntermediateBuffers

The AL_InitIntermediateBuffers function initializes the intermediate buffers needed to decode the current frame.

```
void AL_InitIntermediateBuffers ( AL_TDecCtx *        pCtx,
                                  AL_TDecPicBuffers * pBufs )
```

Parameters

In	pCtx	Pointer to a decoder context object
In	pBufs	Pointer to the current picture buffers

Function AL_LaunchFrameDecoding

The AL_LaunchDecoding function launches a frame decoding request to the hardware IP.

```
void AL_LaunchFrameDecoding (   AL_TDecCtx *                pCtx
```

Parameters

In	pCtx	Pointer to a decoder context object
In	pCtx	Pointer to a decoder context object

Function AL_LaunchSliceDecoding

The AL_LaunchSliceDecoding function launches a slice decoding request to the Hardware IP.

```
void AL_LaunchSliceDecoding (   AL_TDecCtx *                pCtx,
                                bool                        bIsLastAUNal
```

Parameters

In	pCtx	Pointer to a decoder context object
In	bIsLastAUNal	Specify if it's the last AU's slice data

Function AL_PictMngr_BeginFrame

This function prepares the Picture Manager context to new frame encoding; it shall be called before of each frame encoding.

```
bool AL_PictMngr_BeginFrame (   AL_TPictMngrCtx *           pCtx,
                                int                         iWidth,
                                int                         iHeight )
```

Parameters

In	pCtx	Pointer to a Picture Manager context object
In	iWidth	Width of the decoded frame
In	iHeight	Height of the decoded frame

Returns

Returns true if free buffer identifiers have been found and false otherwise.

Function AL_PictMngr_CancelFrame

This function prepares the Picture Manager context to new frame encoding; it shall be called before of each frame encoding.

```
void AL_PictMngr_CancelFrame (  AL_TPictMngrCtx *  pCtx
```

Parameters

In	pCtx	Pointer to a Picture Manager context object
In	pCtx	Pointer to a Picture Manager context object

Function AL_PictMngr_Deinit

Uninitializes the Picture Manager.

```
void AL_PictMngr_Deinit (  AL_TPictMngrCtx *  pCtx
```

Parameters

In	pCtx	Pointer to a Picture Manager context object
In	pCtx	Pointer to a Picture Manager context object

Function AL_PictMngr_EndDecoding

This function updates the Picture Manager context each time a picture has been decoded.

```
void AL_PictMngr_EndDecoding (  AL_TPictMngrCtx *  pCtx,
                                uint8_t            uFrmID,
                                uint8_t            uMvID,
                                uint32_t           uCRC )
```

Parameters

In	pCtx	Pointer to a Picture Manager context object
In	uFrmID	Buffer identifier of the decoded frame buffer
In	uMvID	Buffer identifier of the decoded frame's motion vector buffer
In	uCRC	CRC of the current decoded picture

Function AL_PictMngr_Flush

This function updates the Picture Manager context each time a picture have been decoded.

```
void AL_PictMngr_Flush (  AL_TPictMngrCtx *  pCtx )
```

Parameters

In	pCtx	Pointer to a Picture Manager context object
In	pCtx	Pointer to a Picture Manager context object

Function AL_PictMngr_GetBuffers

```
bool AL_PictMngr_GetBuffers (  AL_TPictMngrCtx *  pCtx,
                               AL_TDecPicParam *  pPP,
                               AL_TDecSliceParam *  pSP,
                               TBufferListRef *  pListRef,
                               TBuffer *  pListAddr,
                               TBufferPOC **  ppPOC,
                               TBufferMV **  ppMV,
                               AL_TBuffer **  ppRec,
                               AL_EFbStorageMode  eFBStorageMode )
```

Function AL_PictMngr_GetCurrentFrmID

Retrieves the current decoded frame identifier.

```
uint8_t AL_PictMngr_GetCurrentFrmID (  AL_TPictMngrCtx *  pCtx )
```

Parameters

In	pCtx	Pointer to a Picture Manager context object

Returns

Return the current decoded frame identifier.

Function AL_PictMngr_GetCurrentMvID

Retrieves the current decoded frame's motion-vectors buffer identifier.

```
uint8_t AL_PictMngr_GetCurrentMvID (   AL_TPictMngrCtx *                pCtx )
```

Parameters

In	pCtx	Pointer to a Picture Manager context object

Returns

Returns the current decoded frame's motion-vectors buffer identifier.

Function AL_PictMngr_GetCurrentPOC

Retrieves the POC of the current decoded frame.

```
int32_t AL_PictMngr_GetCurrentPOC (   AL_TPictMngrCtx *                pCtx )
```

Parameters

In	pCtx	Pointer to a Picture Manager context object

Returns

Returns the POC value of the current decoded frame.

Function AL_PictMngr_GetDisplayBuffer

Returns the next picture buffer to be displayed if possible. The function HEVC_ReleaseDisplayBuffer must be called immediately after buffer is no longer needed.

```
AL_TBuffer* AL_PictMngr_GetDisplayBuffer (  AL_TPictMngrCtx *    pCtx,
                                            AL_TInfoDecode *     pInfo )
```

Parameters

In	pCtx	Pointer to a Picture Manager context object
Out	pInfo	Pointer to retrieve information about the decoded frame

Returns

Returns a pointer to the picture buffer to be displayed if it exists and NULL otherwise.

See Also

HEVC_ReleaseDisplayBuffer()

Function AL_PictMngr_GetLastPicID

This function returns the Pic ID of the last inserted frame.

```
uint8_t AL_PictMngr_GetLastPicID (  AL_TPictMngrCtx *    pCtx )
```

Parameters

In	pCtx	Pointer to a Picture Manager context object
In	pCtx	Pointer to a Picture Manager context object

Returns

Returns the Pic ID of the last inserted frame 0xFF if the DPB is empty.

Function AL_PictMngr_Init

Initializes the Picture Manager.

```
bool AL_PictMngr_Init (  AL_TPictMngrCtx *   pCtx,
                         bool                bAvc,
                         uint16_t            uWidth,
                         uint16_t            uHeight,
                         uint8_t             uNumMvBuf,
                         uint8_t             uNumRef,
                         uint8_t             uNumInterBuf )
```

Parameters

In	pCtx	Pointer to a Picture Manager context object
In	bAvc	Specifies if we use the AVC or HEVC codec
In	uWidth	Picture width in pixels unit
In	uHeight	Picture height in pixels unit
In	uNumMvBuf	Number of motion-vector buffer to manage
In	uNumRef	Number of reference to manage
In	uNumInterBuf	Number of intermediate buffer to manage

Returns

Returns true on success and false otherwise.

Function AL_PictMngr_Insert

Inserts a decoded frame into the DPB.

```
void AL_PictMngr_Insert (  AL_TPictMngrCtx *   pCtx,
                           int                 iFramePOC,
                           uint32_t            uPocLsb,
                           uint8_t             uFrmID,
                           uint8_t             uMvID,
                           uint8_t             pic_output_flag,
                           AL_EMarkingRef      eMarkingFlag,
                           uint8_t             uNonExisting,
                           AL_ENut             eNUT )
```

Parameters

In, Out	pCtx	Pointer to a Picture Manager context object
In	iFramePOC	Picture order count of the decoded picture
In	uPocLsb	poc_lsb value of the decoded picture
In	uFrmID	Frame idx of he associated frame buffer
In	uMvID	Motion-vector Idx of the associated frame buffer
In	pic_output_flag	Flag which specifies if the decoded picture is needed for output
In	eMarkingFlag	Reference status of the decoded picture
In	uNonExisting	Non existing status of the decoded picture
In	eNUT	Added NAL unit type

Function AL_PictMngr_LockRefMvId

Locks reference motion vector buffers.

```
void AL_PictMngr_LockRefMvId (  AL_TPictMngrCtx *   pCtx,
                                uint8_t             uNumRef,
                                uint8_t *           pRefMvId )
```

Parameters

In	pCtx	Pointer to a Picture Manager context object
In	uNumRef	Number of reference pictures
In	pRefMvId	List of motion vectors buffer IDs associated to the reference pictures

Function AL_PictMngr_PutDisplayBuffer

```
void AL_PictMngr_PutDisplayBuffer ( AL_TPictMngrCtx *  pCtx,
                                    AL_TBuffer *       pBuf
```

Function AL_PictMngr_ReleaseDisplayBuffer

This function releases a picture buffer previoulsy obtained through HEVC_GetDispalyBuffer.

```
void AL_PictMngr_ReleaseDisplayBuffer ( AL_TPictMngrCtx *  pCtx,
                                        AL_TBuffer *       pBuf )
```

Parameters

In	pCtx	Pointer to a Picture Manager context object
In	pBuf	Pointer to the picture buffer to release

Function AL_PictMngr_Terminate

Flushes all pictures so all buffers are fully released.

```
void AL_PictMngr_Terminate ( AL_TPictMngrCtx *  pCtx )
```

Parameters

In	pCtx	Pointer to a Picture Manager context object
In	pCtx	Pointer to a Picture Manager context object

Function AL_PictMngr_UnlockRefMvId

Unlocks reference motion vector buffers.

```
void AL_PictMngr_UnlockRefMvId ( AL_TPictMngrCtx *  pCtx,
                                 uint8_t            uNumRef,
                                 uint8_t *          pRefMvId )
```

Parameters

In	pCtx	Pointer to a Picture Manager context object
In	uNumRef	Number of reference pictures
In	pRefMvId	List of motion vectors buffer IDs associated to the reference pictures

Function AL_PictMngr_UpdateDPBInfo

This function updates the number of reference managed by the picture manager.

```
void AL_PictMngr_UpdateDPBInfo (  AL_TPictMngrCtx *   pCtx,
                                  uint8_t             uMaxRef )
```

Parameters

In	pCtx	Pointer to a picture manager context object
In	uMaxRef	Maximum number of references managed by the picture manager

Function AL_SetConcealParameters

The AL_SetConcealParameters sets the conceal ID buffer and availability flag.

```
void AL_SetConcealParameters (  AL_TDecCtx *        pCtx,
                                AL_TDecSliceParam * pSP )
```

Parameters

In	pCtx	Pointer to a decoder context object
Out	pSP	Pointer to the current slice parameters

Function AL_TerminatePreviousCommand

The AL_TerminatePreviousCommand flushes one decoding command by computing parameters relative to next slice.

```
void AL_TerminatePreviousCommand (  AL_TDecCtx *        pCtx,
                                    AL_TDecPicParam *   pPP,
                                    AL_TDecSliceParam * pSP,
                                    bool                bIsLastVclNalInAU,
                                    bool                bNextIsDependent
```

Parameters

In	pCtx	Pointer to a decoder context object
In	pPP	Pointer to the current picture parameters
In	pSP	Pointer to the current slice parameters
In	bIsLastVclNalInAU	Specifies if this is the last NAL of the current access unit
In	bNextIsDependent	Specifies if the next slice segment is a dependent or non-dependent slice

Function avc_scaling_list_data

Computes the custom scaling_list from the SPS NAL.

```
void avc_scaling_list_data (  uint8_t *            pScalingList,
                              AL_TRbspParser *     pRP,
                              int                  iSize,
                              uint8_t *            pUseDefaultScalingMatrixFlag
```

Parameters

In	pRP	Pointer to NAL parser
In	iSize	Size of the scaling list
Out	pUseDefaultScalingMatrixFlag	Array specifying when default scaling maxtrix is used
Out	pScalingList	Pointer to the custom scaling matrix values

Function avc_vui_parameters

Parses the vui_parameters.

```
bool avc_vui_parameters (  AL_TVuiParam *       pVuiParam,
                           AL_TRbspParser *     pRP )
```

Parameters

In	pRP	Pointer to NAL parser
Out	pVuiParam	Pointer to the vui_parameters structure that is to be filled in

Function GetBlk2Buffers

Retrieves the buffer needed for the second block decoding.

```
void GetBlk2Buffers (  AL_TDecCtx *            pCtx,
                       AL_TDecSliceParam *     pSP )
```

Parameters

In	pCtx	Pointer to a decoder context object
In	pSP	Pointer to the current slice parameters

Function getParserOnNonVclNal

```
AL_TRbspParser getParserOnNonVclNal (  AL_TDecCtx *     pCtx
```

Function hevc_hrd_parameters

Parses the hrd_parameters.

```
void hevc_hrd_parameters ( AL_THrdParam *       pHrdParam,
                           bool                 bInfoFlag,
                           uint8_t              uMaxSubLayers,
                           AL_TRbspParser *     pRP )
```

Parameters

Out	pHrdParam	Pointer to the hrd_parameters structure that is to be filled in
In	bInfoFlag	Common info present flag
In	uMaxSubLayers	Max number of sub layers
In	pRP	Pointer to NAL parser

Function hevc_scaling_list_data

Builds the scaling list.

```
void hevc_scaling_list_data ( AL_TSCLParam *       pSCLParam,
                              AL_TRbspParser *     pRP )
```

Parameters

Out	pSCLParam	Receives the scaling list data
In	pRP	Pointer to NAL parser

Function hevc_vui_parameters

Parses the vui_parameter.

```
void hevc_vui_parameters ( AL_TVuiParam *       pVuiParam,
                           uint8_t              uMaxSubLayers,
                           AL_TRbspParser *     pRP )
```

Parameters

Out	pVuiParam	Pointer to the vui_parameters structure that is to be filled in
In	uMaxSubLayers	Max number of sub layer
In	pRP	Pointer to NAL parser

Function ParseVPS

Parses a VPS NAL.

```
void ParseVPS ( AL_THevcVps          pVPSTable[],
                AL_TRbspParser *     pRP )
```

Parameters

Out	pVPSTable	Pointer to the table where the parsed VPSs are stored
In	pRP	Pointer to NAL parser

Function profile_tier_level

Retrieves the profile and level syntax elements.

```
void profile_tier_level ( AL_TProfilevel *     pPrfLvl,
                          uint8_t              uMaxSubLayers,
                          AL_TRbspParser *     pRP )
```

Parameters

Out	pPrfLvl	Pointer to the profile and level structure that is to be filled in
In	uMaxSubLayers	Max number of sub layer
In	pRP	Pointer to NAL buffer

Function SkipNal

```
bool SkipNal (  )
```

Function UpdateCircBuffer

```
void UpdateCircBuffer ( AL_TRbspParser *     pRP,
                        TCircBuffer *        pBufStream,
                        int *                pSliceHdrLength )
```

Function UpdateContextAtEndOfFrame

Resets the context at the end of a frame.

```
void UpdateContextAtEndOfFrame (   AL_TDecCtx *                pCtx)
```

Parameters

In	pCtx	Pointer to a decoder context object

Debugging

This appendix includes details about resources available on the Xilinx Support website and debugging tools.

TIP: *If the IP generation halts with an error, there might be a license issue.*

Finding Help on Xilinx.com

To help in the design and debug process when using the Video Codec Unit, the Xilinx Support web page contains key resources such as product documentation, release notes, answer records, information about known issues, and links for obtaining further product support.

Documentation

This product guide is the main document associated with the Video Codec Unit. This guide, along with documentation related to all products that aid in the design process, can be found on the Xilinx Support web page or by using the Xilinx Documentation Navigator.

Download the Xilinx Documentation Navigator from the Downloads page. For more information about this tool and the features available, open the online help after installation.

Solution Centers

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

Answer Records

Answer Records include information about commonly encountered problems, helpful information on how to resolve these problems, and any known issues with a Xilinx product. Answer Records are created and maintained daily ensuring that users have access to the most accurate information available.

Answer Records for this core can be located by using the Search Support box on the main Xilinx support web page. To maximize your search results, use proper keywords such as

- Product name
- Tool message(s)
- Summary of the issue encountered

A filter search is available after results are returned to further target the results.

Master Answer Record for the Video Codec Unit

AR: 66763

Technical Support

Xilinx provides technical support at the Xilinx Support web page for this LogiCORE™ IP product when used as described in the product documentation. Xilinx cannot guarantee timing, functionality, or support if you do any of the following:

- Implement the solution in devices that are not defined in the documentation.
- Customize the solution beyond that allowed in the product documentation.
- Change any section of the design labeled DO NOT MODIFY.

To contact Xilinx Technical Support, navigate to the Xilinx Support web page.

Debug Tools

There are many tools available to address Video Codec Unit design issues. It is important to know which tools are useful for debugging various situations.

Vivado Design Suite Debug Feature

The Vivado® Design Suite debug feature inserts logic analyzer and virtual I/O cores directly into your design. The debug feature also allows you to set trigger conditions to capture application and integrated block port signals in hardware. Captured signals can then be analyzed. This feature in the Vivado IDE is used for logic debugging and validation of a design running in Xilinx devices.

The Vivado logic analyzer is used with the logic debug IP cores, including:

- ILA 2.0 (and later versions)
- VIO 2.0 (and later versions)

See the *Vivado Design Suite User Guide: Programming and Debugging* (UG908) [Ref 5].

Reference Boards

Various Xilinx development boards support the Video Codec Unit. These boards can be used to prototype designs and establish that the core can communicate with the system.

- ZCU106

Hardware Debug

Hardware issues can range from link bring-up to problems seen after hours of testing. This section provides debug steps for common issues. The Vivado debug feature is a valuable resource to use in hardware debug. The signal names mentioned in the following individual sections can be probed using the debug feature for debugging the specific problems.

General Checks

Ensure that all the timing constraints for the core were properly incorporated from the example design and that all constraints were met during implementation.

- Does it work in post-place and route timing simulation? If problems are seen in hardware but not in timing simulation, this could indicate a PCB issue. Ensure that all clock sources are active and clean.
- If using MMCMs in the design, ensure that all MMCMs have obtained lock by monitoring the `locked` port.
- If your outputs go to 0, check your licensing.

Debugging a VCU-based System

Troubleshooting a VCU based system can be complicated. This appendix offers a decision tree to guide you efficiently to the most productive areas to investigate. The troubleshoot steps apply to a VCU based system capable of performing live capture, encode, decode, transport and display.

Debug Flow

The following chart shows the system level debug flow.

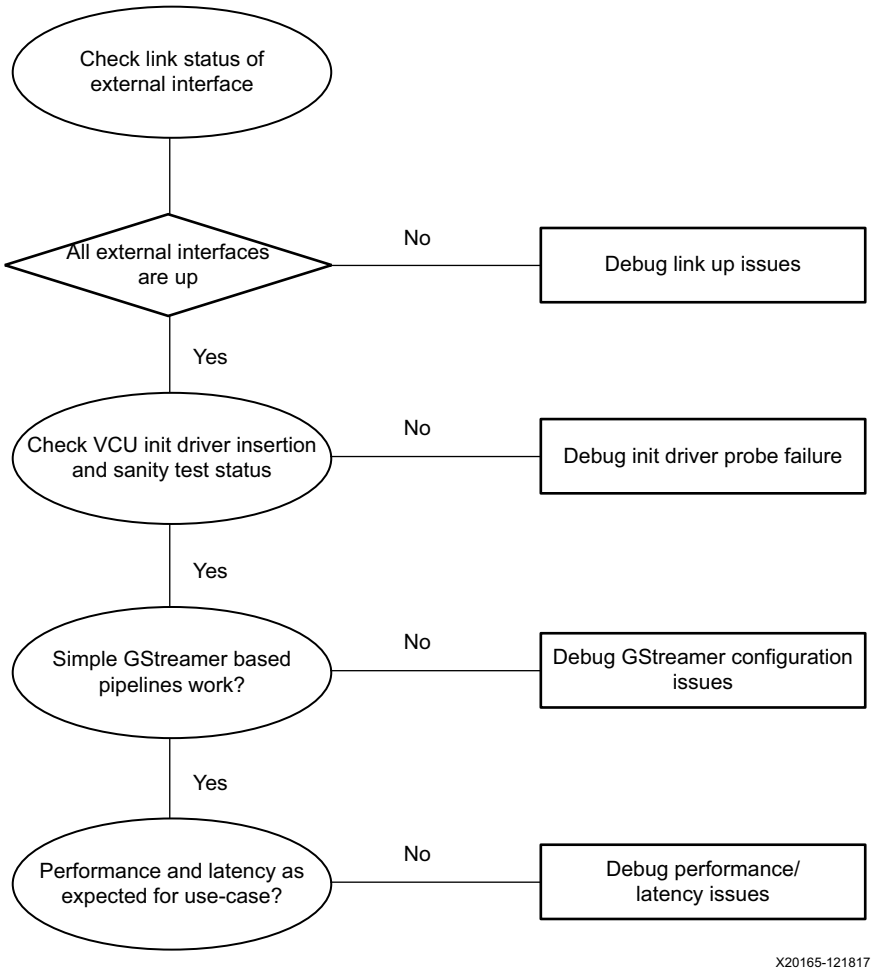


Figure A-1: **Debug Flow**

Troubleshooting

Debugging External Interfaces

If you are using capture and display interfaces based on high-speed serial IO (eg. HDMI, MIPI, SDI etc), ensure physical links are up before debugging the upper layers. Please follow these steps:

1. Monitor for LEDs that indicate heart beat clocks that are derived based on reference clock input to transceiver. If the heart beat clocks are absent, check for GT PLL lock status.
2. If PLL is not locked check for Video PHY IP configuration and reference clock constraint. Most of the time, the reference clock is sourced from a programmable clock chip. Ensure the frequency is programmed properly in the device tree source file for the programmable clock chip and the frequency is not modified by other Linux devices (happens when the `phandle` of the clock source node is shared between multiple components).
3. If the capture interface is compatible with the Video for Linux (v4l2) framework, use the media-ctl API to verify the link topology and link status.

   ```
   xmedia-ctl -p -d /dev/mediaX
   ```

   The mediaX represents the pipeline device in the v4l2 pipeline. If you have multiple TPG/HDMI pipelines, they appear as `/dev/media0`, `/dev/media1`, etc. The link status is indicated in the corresponding sub-device node. For example, a HDMI RXSS node indicates link status for the HDMI link in its sub-device properties while using the above command. If link up fails, a "no-link" message appears. Otherwise, valid resolution with proper color format is detected.
4. If the display interface is compatible with DRM/KMS framework, please ensure DRM is linking up by running the one of following commands:

   If the PL mixer block is used,

   ```
   modetest -M xilinx_drm_mixer
   ```

   If the PL mixer block is not used,

   ```
   modetest -M xilinx_drm
   ```

   If you want to display a pattern to ensure the display path is working, use one of the following commands:

   To display using BG24 format,

   ```
   modetest -M xilinx_drm_mixer -s <connector_id>@<crtc_id>:3840x2160-60@BG24
   ```

To display using NV12 format

```
modetest -M xilinx_drm_mixer -P <crtc_id>:3840x2160-60@NV12
```

Run the `modetest` command multiple times to ensure a stable link at different resolution before proceeding with different VCU use-cases.

5. If the capture pipeline (v4l2 based) or display pipeline (DRM/KMS based) are broken, then `media-ctl` or `modetest` applications show a broken pipeline and sub-devices are not being linked properly. If so, browse through the boot log to see if there is a probe failure for any of the v4l2 sub-device drivers. If the probe fails, check your dts file for any property name mismatch or missing/incorrect property.

Debugging the VCU Interface

To debug the VCU interface, perform the following steps:

1. Ensure VCU init driver probe is successful during boot process. In the boot log, you can see the following messages

   ```
   xilinx-vcu <axilite address>.vcu: xvcu_probe: Probed successfully
   ```

2. If PLL fails to lock, VCU initialization driver reports a lock status failure message. Ensure that the PLL input reference clock frequency of VCU LogiCORE IP is set properly in the IPI block design. Ensure that the PLL clock source (parent clock) is modelled correctly in device tree node as a fixed clock source. Ensure VCU init driver node properties are proper and `phandle` for PLL reference clock is passed correctly.
3. After successful probe of VCU init driver, check VCU drivers with the following command:

   ```
   lsmod
   ```

   which should show al5d, al5e and Allegro modules as being inserted.

4. Check if sufficient CMA is allocated. VCU operation requires at least 1000 MB of CMA. You can check for available CMA by using

   ```
   cat /proc/meminfo
   ```

5. If CMA size is not sufficient, increase the size either in u-boot using a command:

   ```
   cma=xxxxM
   ```

   or while building the Linux kernel using kernel configuration property.

6. Run a VCU sanity test using on of the VCU Control Software sample applications. You can use `AL_Encoder.exe` and `AL_Decoder.exe` applications to run a sanity test.

Debugging Gstreamer Based Application

To debug a GStreamer based application, perform the following steps.

1. Run capture to display pipeline to ensure live capture to display is working as expected.

```
gst-launch-1.0 -v v4l2src io-mode=4 device=/dev/video1 ! video/
x-raw,format=NV12,width=3840,height=2160, framerate=30/1 ! kmssink
driver-name=xilinx_drm_mixer
```

2. If you see a streamon error with the capture->display pipeline, ensure that the format is set to NV12 using v4l2-ctl and media-ctl API inside hdmi configuration script. Here is an example

```
v4l2-ctl -d /dev/video1 --set-fmt-video=width=3840,height=2160,pixelformat='NV12'
xmedia-ctl -d /dev/media1 -V "\"a0080000.scaler\":0  [fmt:RBG888_1X24/3840x2160
field:none]"
xmedia-ctl -d /dev/media1 -V "\"a0080000.scaler\":1  [fmt:VYYUYY8_1X24/3840x2160
field:none]"
```

3. Run a pipeline with VCU components in the pipeline. Use omxh264/omxh265enc/dec components in the pipeline.
4. For a list of supported properties of omxh264/omxh265enc/dec, use the following command:

```
gst-inspect-1.0 <omxh264/omxh265enc/dec>
```

   which lists all the supported properties of VCU encoder and decoder blocks. Use the properties as appropriate in the pipeline.

5. Start with a known pipeline example first. Here is an example:

```
gst-launch-1.0 -v  \
v4l2src num-buffers=2100 device=/dev/video8 io-mode=4 \
! video/x-raw,format=NV12,width=3840,height=2160, framerate=30/1 \
! omxh265enc ip-mode=2 target-bitrate=70000 prefetch-buffer-size=630 \
 control-rate=2 gop-length=30 b-frames=0 \
! video/x-h265, profile=main,level=\(string\)5.1,tier=main \
! omxh265dec op-mode=1 ip-mode=1 latency-mode=0 internal-entropy-buffers=2 \
! queue \
! fpsdisplaysink name=fpssink text-overlay=false \
video-sink="kmssink max-lateness=100000000 async=false \
sync=true driver-name=xilinx_drm_mixer" -v
```

Debugging Performance Issues

For additional debugging information, see the Debugging Tools page of GStreamer website at https://gstreamer.freedesktop.org/documentation/tutorials/basic/debugging-tools.html.

If the problem is low frame rate or frame dropping, follow these steps to debug the system.

1. Use the `fpsdisplaysink` element to report frame rate and dropped frame count (refer to the example above)

2. Check the QoS settings of HP ports that interface VCU with PS DDR. Check for outstanding transaction count configuration. Please note that for VCU traffic, the QoS should be set as Best Effort (BE) and outstanding transaction count should be set to maximum (0xF for AFI ports).
3. Check if SMMU is enabled in the device tree. If SMMU is enabled, disable it since it imparts longer latency in the transaction completion.
4. Check if encoder buffer is used in the user design. If not, check if encoder buffer impacts the performance. If performance improves with encoder buffer, it might indicate a system bandwidth issue.
5. Try with different encoder/decoder properties to see if the performance drop is related to any of the properties. Avoid B-frames in the pipeline to see if there is any performance improvement. If there is improvement, it might indicate a system bandwidth issue. Reduce the bitrate to see if there is improvement in framer ate. If reducing target-bitrate gives better throughput, it might indicate a system bandwidth issue.
6. Check for CPU utilization while the pipeline is running. A higher CPU utilization indicates there could be impact in interrupt processing time which explains lower framerate.
7. Try using a `queue` element between two GStreamer plugins that are in the datapath to check for any performance improvement
8. Check for DDR bandwidth utilization using DDR APM and VCU APM
9. Use gst-shark (a GStreamer-based tool) to verify performance and create scheduletime and interlatency plots to understand which element is causing performance drops.

Debugging Latency Issues

If the problem is high end-to-end latency, follow these steps to debug the system:

1. Understand how much the jitter buffer is used in the client side pipeline (udpsrc). Please note that the latency for the `rtpjitterbuffer` should correspond to the CPB size in the server pipeline. Often it is useful to use LowLatency rate control (or hardware rate control) algorithm to maintain a lower CPB buffer that reduces the `rtpjitterbuffer` latency.
2. Many times, latency is related to frame drop. Ensure there is no frame drop in the pipeline before measuring latency.
3. Check if use of the filler-data setting in the encoder pipeline is causing longer latency. If so, set filler-data=false in the pipeline and check for latency
4. Use a fine-tuned internal-entropy-buffers count. Note that internal-entropy-buffers setting impacts the latency and an optimal value needs to be used. You can update this decoder property to ensure no frame drop first and later to optimize the pipeline latency.

Debugging VCU Quality Issues

Perform the following steps to debug HEVC/AVC encoder quality issues.

1. Evaluate the valid bitrate range for the given video stream by using CONST_QP rate control algorithm.
2. Use a QP value of 22, 27, 32, 37 with the CONST_QP rate control algorithm
3. Using the bitrate range from step 2, evaluate PSNR for CBR/VBR/Low latency rate control algorithms
4. Run similar experiments for libx264 or libx265 encoders.

   Here is an example pipeline for CBR:

   ```
   ffmpeg -s 1280x720 -pix_fmt yuv420p -framerate 50 -i /srv/data1/kvikrama/
   vcu_video_quality_runs/source/old_town_cross_420_720p50.yuv -c:v libx265 -preset
   medium -x265-params bframes=0:ref=1:keyint=30:rc
   lookahead=0:ipratio=1:pbratio=1:trellis=0 -b:v 1M -minrate 1M -maxrate 1M -bufsize
   2M   -frames 500   old_town_cross_420_720p50_x265.mp4
   ```

5. Calculate the BD-rate using JCT-VC common test conditions evaluation metric.
6. If there is difference in PSNR between VCU and libx264/libx265, tune the following parameters to see impact:

MinQP, MaxQP, ScnChgResilience (should be Enabled), NumSlices (set to 1).

Interface Debug

AXI4-Lite Interfaces

To verify that the interface is functional, try reading from a register that does not have all 0s as its default value. Output `s_axi_arready` asserts when the read address is valid, and output `s_axi_rvalid` asserts when the read data/response is valid. If the interface is unresponsive, ensure that the following conditions are met:

- The `s_axi_aclk` and `aclk` inputs are connected and toggling.
- The interface is not being held in reset, and `s_axi_areset` is an active-Low reset.
- The interface is enabled, and `s_axi_aclken` is active-High (if used).
- The main core clocks are toggling and that the enables are also asserted.
- If the simulation has been run, verify in simulation and/or a debug feature capture that the waveform is correct for accessing the AXI4-Lite interface.

AXI4-Stream Interfaces

If data is not being transmitted or received, check the following conditions:

- If transmit `<interface_name>_tready` is stuck Low following the `<interface_name>_tvalid` input being asserted, the core cannot send data.
- If the receive `<interface_name>_tvalid` is stuck Low, the core is not receiving data.
- Check that the `aclk` inputs are connected and toggling.
- Check that the AXI4-Stream waveforms are being followed.
- Check core configuration.

Additional Resources and Legal Notices

Xilinx Resources

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

References

These documents provide supplemental material useful with this product guide:

1. *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994)
2. *Vivado Design Suite User Guide: Designing with IP* (UG896)
3. *Vivado Design Suite User Guide: Getting Started* (UG910)
4. *Vivado Design Suite User Guide: Logic Simulation* (UG900)
5. *Vivado Design Suite User Guide: Programming and Debugging* (UG908)
6. *Vivado Design Suite User Guide: Implementation* (UG904)
7. *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906)
8. *LogiCORE IP AXI Interconnect Product Guide* (PG059)
9. *Zynq UltraScale+ MPSoC Production Errata* (EN285)
10. *Zynq UltraScale+ Device Technical Reference Manual* (UG1085)
11. *Zynq UltraScale+ MPSoC Register Reference* (UG1087)
12. *Zynq UltraScale+ MPSoC Data Sheet: Overview* (DS891)
13. *Zynq UltraScale+ MPSoC Data Sheet: DC and AC Switching Characteristics* (DS925)
14. *LogiCORE IP Zynq UltraScale+ MPSoC Processing System Product Guide* (PG201)
15. *PetaLinux Tools Documentation* (UG1144)
16. OpenMax Integration Layer (https://www.khronos.org/registry/OpenMAX-IL/specs/OpenMAX_IL_1_1_2_Specification.pdf)

17. GStreamer (https://gstreamer.freedesktop.org/features/)

Training Resources

1. Vivado Design Suite Hands-on Introductory Workshop
2. Vivado Design Suite Tool Flow

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
12/20/2017	1.0	Reorganized content and updated API.
10/04/2017	1.0	Initial Release.

Please Read: Important Legal Notices