LogiCORE IP Object Segmentation v2.0

Product Guide

PG018 October 19, 2011

XILINX®

Table of Contents

Appendix B: Debugging

Appendix C: Application Software Development

Appendix D: C Model Reference

Appendix E: Additional Resources

LogiCORE IP Object Segmentation v2.0

Introduction

The Xilinx® LogiCORE™ Intellectual Property (IP) Object Segmentation core provides a hardware-accelerated method for identifying objects of interest within a video stream. The user provides a set of object criteria that describes the objects of interest and the core processes statistical data generated by the Image Characterization LogiCORE IP to "find" the objects of interest. The objects are output as Metadata for subsequent higher level analysis and processing. The core is programmed either directly through the register set when using the General Purpose Processor configuration or by using the supplied software drivers when using the Embedded Development Kit (EDK) pCore configuration.

Features

- User-defined object criteria:
  - Up to eight Feature Combinations (upper and lower thresholds on mean, variance, edge, motion and color information)
  - Up to four Feature Selections (any Boolean combination of the eight feature combinations)
- Detects up to 31 objects per Feature Selection and up to 124 objects per frame
- Operates at all resolutions and frame rates supported by Image Characterization block (up to 720P60 and 1080P30)
- Selectable processor interface
  - EDK pCore
  - General Purpose Processor
- Advanced eXtensible Interface (AXI4) memory mapped interface; AXI4-Lite processor interface (EDK pCore)
- For use with Xilinx CORE Generator™ tool 13.3

LogiCORE IP Facts Table	
Core Specifics	
Supported Device Family (1)	Virtex®-7, Kintex™-7, Virtex-6, Spartan®-6
Supported User Interfaces	AXI4, AXI4-Lite, General Purpose Processor
Resources	See Table 1-1, Table 1-2, Table 1-3 and Table 1-4.
Provided with Core	
Design Files	Netlist or EDK pCore
Example Design	Not Provided
Test Bench	Provided on the product page (Verilog)
Constraints File	Not Provided
Simulation Model	VHSIC Hardware Description Language (VHDL) or Verilog Structural model C model provided on the product page
Tested Design Tools	
Design Entry Tools	Integrated Software Environment (ISE®) 13.3 Xilinx Platform Studio (XPS) 13.3
Simulation (2)	ModelSim
Synthesis Tools (2)	Xilinx Synthesis Technology (XST)
Support	
Provided by Xilinx @ www.xilinx.com/support	

1. For a complete listing of supported devices, see the release notes for this core.
2. For the supported versions of the tools, see the ISE Design Suite 13: Release Notes Guide.

XILINX®

Chapter 1

Overview

The Object Segmentation core is part of a trio of IP Cores (along with Motion Adaptive Noise Reduction and Image Characterization) that enables video analytics systems. These cores provide a hardware-based solution for the computationally-intensive pixel level processing required in video analytics. They produce object Metadata for processing by a system processor or other processing block, eliminating the burden of pixel processing for these components. This approach enables video analytics solutions that can operate at high-definition resolutions and full-frame rates.

In the video analytics system, objects are defined as a rectangular region that matches a set of defined object characteristics (Figure 1-1). The Object Segmentation core plays a key role in the video analytics system. It is responsible for parsing a data structure that describes the characteristics of an image and then "finding" the objects in the image that meet a set of object characteristics.

At a high level, the video analytics system takes a frame of video and subdivides it into a 2-D grid of NxN subdivisions called image blocks. For each image block, a set of statistics is calculated. The Object Segmentation core then compares the statistics of each image block against a set of thresholds that define upper and lower bounds. An image block whose statistics match the set of thresholds is considered an object block. After all of the image blocks have been tested, the core aggregates the object blocks into full objects. Two blocks are aggregated if they are neighbors horizontally, vertically, or diagonally. After all the object blocks have been aggregated into full objects, each object is analyzed to define a box that completely bounds the object. The bounding box defines the object. The final step for the Object Segmentation core is to generate Metadata that consists of a list of all the objects that were found in the image. The Metadata is written to external memory where it can be read by a software application that performs higher-level analysis and processing.

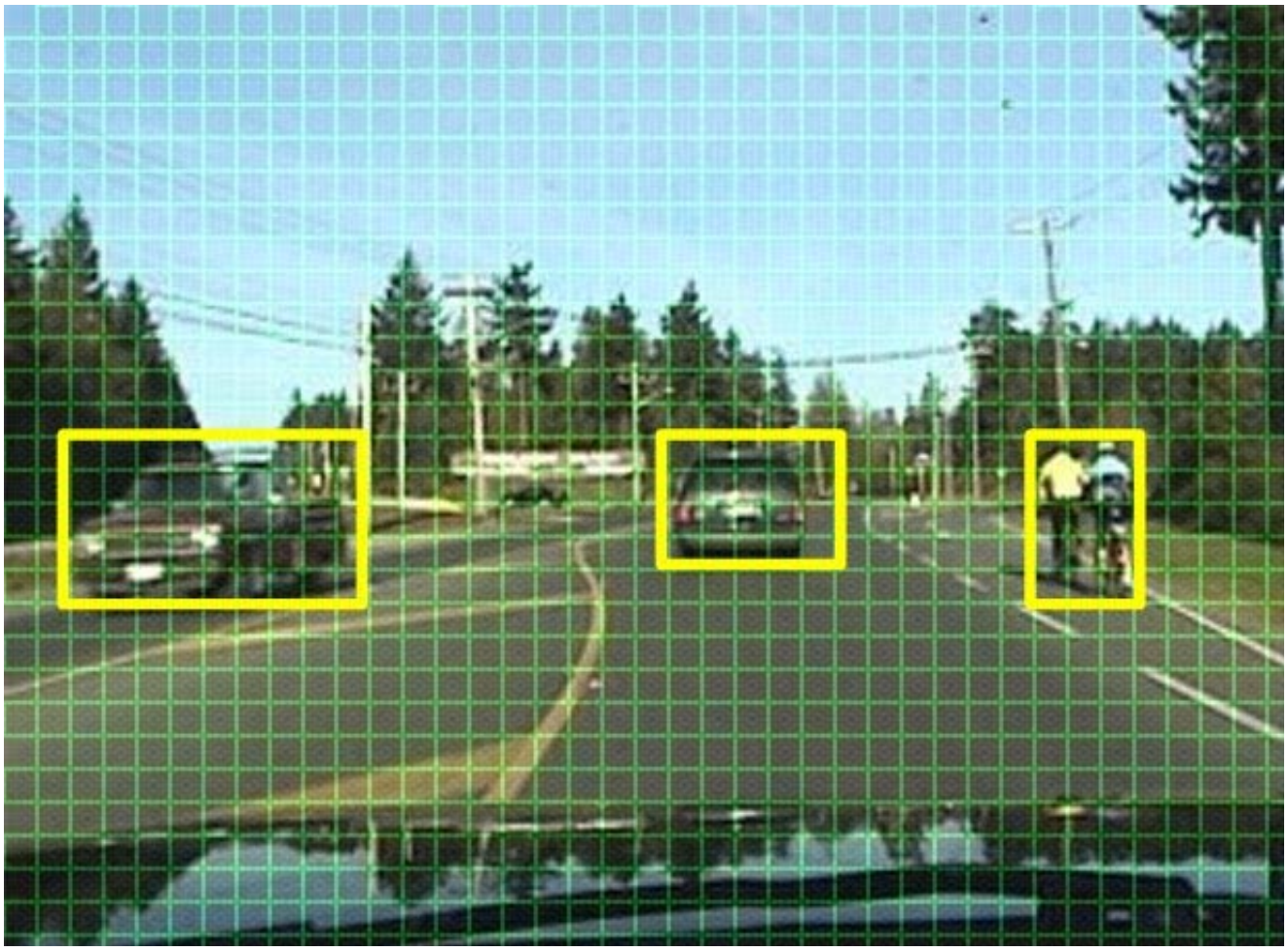

Figure 1-1: **Object Segmentation Image View**

Standards Compliance

The Object Segmentation core is compliant with the AXI4 and AXI4-Lite interconnect standards as defined in the *AXI Reference Guide* (UG761).

Operating System Requirements

For a list of System Requirements, see ISE Design Suite 13: Release Notes Guide.

Feature Summary

The Object Segmentation core supports up to eight Feature Combinations as described in the Feature Combination section. The user has the option of selecting 1– 8 Feature Combinations when generating the core. Selecting fewer Feature Combinations conserves resources. The Object Segmentation core supports up to four Feature Selects as described in the Feature Select section. The user has the option of selecting 1– 4 Feature Selects when generating the core. Selecting fewer Feature Selects conserves resources.

For each Feature Select that is instantiated, the core can detect up to 31 objects per frame. If four Feature Selects are instantiated, up to 124 objects can be detected for each frame.

The Object Segmentation core is capable of operating at all resolutions, frame rates and block sizes that are supported by the Xilinx® Image Characterization v2.0 IP core.

When generating the Object Segmentation core, the user has the option of selecting the type of processor interface that is instantiated on the core. The first option is an EDK pCore interface that can be easily incorporated into an EDK project. The second option is a General Purpose Processor interface. This option exposes the core's registers to the user. The user can wrap the exposed registers in an interface that is compliant with the systems processor.

Applications

- Video Surveillance
- Industrial Control
- Machine Vision
- Automotive
- Other video applications requiring video analytics

Licensing

The Xilinx Image Characterization core provides three licensing options. After installing the required Xilinx ISE® software and IP Service Packs, choose a license option.

Simulation Only

The Simulation Only Evaluation license key is provided with the Xilinx CORE Generator™ tool. This key lets you assess the core functionality with your own design and demonstrates the various interfaces on the core in simulation. (Functional simulation is supported by a dynamically-generated Hardware Description Language (HDL) structural model.)

Full System Hardware Evaluation License

To obtain a Full System Hardware Evaluation license:

1. Navigate to the product page for this core.
2. Click Evaluate.
3. Follow the instructions to install the required Xilinx ISE software and IP Service Packs.

Obtaining a Full License

To obtain a Full license key, you must purchase a license for the core. After doing so, click the "Access Core" link on the Xilinx.com IP core product page for further instructions.

Installing Your License File

The Simulation Only Evaluation license key is provided with the ISE software CORE Generator system and does not require installation of an additional license file. For the Full System Hardware Evaluation license and the Full license, an email will be sent to you containing instructions for installing your license file. Additional details about IP license key installation can be found in the ISE Design Suite Installation, Licensing and Release Notes document.

Performance

The following sections detail the performance characteristics of the Object Segmentation v2.0 core.

Maximum Frequency

The following are typical clock frequencies for the target devices. The maximum achievable clock frequency can vary. The maximum achievable clock frequency and all resource counts can be affected by other tool options, additional logic in the Field Programmable Gate Array (FPGA) device, using a different version of Xilinx tools, and other factors.

- Virtex®-7 FPGA: 225 MHz
- Kintex™-7 FPGA: 150 MHz
- Virtex-6 FPGA: 225 MHz
- Spartan®-6 FPGA: 150 MHz

Latency

The Object Segmentation core outputs a Metadata structure after it has fully processed the input Image Characterization data structure. The Object Segmentation core requires approximately 40 clock cycles to process each set of block statistics in the Image Characterization data structure. Therefore, the latency depends on the number of blocks in the Image Characterization data structure.

Throughput

The Object Segmentation core process the input Image Characterization data structure in two passes. During the first pass, the core outputs one 32-bit word for each set of block statistics in the Image Characterization data structure. During the second pass, the core outputs the Metadata data structure. The size of the Metadata data structure that is output depends on the number of Feature Selects that are instantiated in the core. For each Feature Select, the Object Segmentation core outputs 192 32-bit words. The core also outputs a data structure header that consists of 32 32-bit words.

Resource Utilization

Resources required for the Object Segmentation core have been estimated for these FPGAs: Virtex-7 (Table 1-1), Kintex-7 (Table 1-2), Virtex-6 (Table 1-3) and Spartan-6 (Table 1-4).

Start the resource count with the resources from the "Base Core" which includes the resources for one Feature Combination and one Feature Select. If using more than one Feature Combination, multiply the resources in the Each additional Feature Combination row by the number of extra Feature Combinations and add the results to the resource count. If using more than one Feature Select, multiply the resources in the Each additional Feature Select row by the number of extra Feature Selects and add the results to the resource count. If using the pCore Interface, add the corresponding resources to the results count.

Table 1-1: **Virtex-7 Resource Estimates**

Feature	LUTs	FFs	Block RAMs (36/18)	DSP48E1s
Base Core (Feature Combination = 1, Feature Select = 1)	3883	3315	2/4	4
Each additional Feature Combination	200	149	0/0	0
Each additional Feature Select	642	562	1/0	0
pCore Interface	850	800	0/0	0

Table 1-2: **Kintex-7 Resource Estimates**

Feature	LUTs	FFs	Block RAMs (36/18)	DSP48A1s
Base Core (Feature Combination = 1, Feature Select = 1)	3890	3315	2/4	4
Each additional Feature Combination	210	149	0/0	0
Each additional Feature Select	669	562	1/0	0
pCore Interface	850	800	0/0	0

Table 1-3: **Virtex-6 Resource Estimates**

Feature	LUTs	FFs	Block RAMs (36/18)	DSP48E1s
Base Core (Feature Combination = 1, Feature Select = 1)	3265	3315	2/4	4
Each additional Feature Combination	223	149	0/0	0
Each additional Feature Select	578	559	1/0	0
pCore Interface	850	800	0/0	0

Table 1-4: **Spartan-6 Resource Estimates**

Feature	LUTs	FFs	Block RAMs (16/8)	DSP48E1s
Base Core (Feature Combination = 1, Feature Select = 1)	3086	3322	5/3	4
Each additional Feature Combination	205	150	0/0	0
Each additional Feature Select	650	575	1/0	0
pCore Interface	850	800	0/0	0

XILINX®

Chapter 2

Core Interfaces and Register Space

This chapter provides detailed descriptions for the supported interfaces, along with details about the configuration and control registers for the Object Segmentation core.

Port Descriptions

Core Interfaces

AXI4 Memory Interface

The Object Segmentation core uses an AXI4 interface to connect to the AXI4 Interconnect. The AXI4 Interconnect provides the access to external memory. The core provides registers that allow the user to specify the location in memory of the various data buffers that the Object Segmentation core accesses. See Table 2-4 for more details on these registers.

Processor Interface

There are many video systems developed that use an integrated processor system to dynamically control the parameters within the system. This is especially important when several independent image processing cores are integrated into a single FPGA. The Object Segmentation core can be configured with one of two interfaces: an EDK pCore Interface or a General Purpose Processor Interface.

Common I/O Signals

The EDK pCore interface and the General Purpose Processor interface share a number of the same Input/Output (I/O) signals. The signals that both interfaces share are specified in Table 2-1.

Table 2-1: **Common I/O Signals**

Pin Name	Dir	Width	Description
Core Signals			
clk	I	1	Core clock
fsync_in	I	1	Frame Synchronization
buffer_ptr	I	I	Buffer Select input

Table 2-1: **Common I/O Signals *(Cont'd)***

Pin Name	Dir	Width	Description
AXI4 Memory Map to Stream (MM2S) Read Address Channel			
m_axi_mm2s_araddr	O	32	MM2S Read Address
m_axi_mm2s_arlen	O	8	MM2S Read Length Qualifier
m_axi_mm2s_arsize	O	3	MM2S Read Size Qualifier
m_axi_mm2s_arburst	O	2	MM2S Read Burst Type Qualifier
m_axi_mm2s_arprot	O	3	MM2S Read Protection Qualifier
m_axi_mm2s_arcache	O	4	MM2S Read Cache Qualifier
m_axi_mm2s_arvalid	O	1	MM2S Read Address Valid Qualifier
m_axi_mm2s_arready	I	1	MM2S Read Address Ready Status
AXI4 MM2S Read Data Channel			
m_axi_mm2s_rdata	I	32	MM2S Read Data
m_axi_mm2s_rresp	I	2	MM2S Read Response
m_axi_mm2s_rlast	I	1	MM2S Read Last Indication
m_axi_mm2s_rvalid	I	1	MM2S Read Valid Handshake
m_axi_mm2s_rready	O	1	MM2S Read Ready Handshake
AXI4-Stream to Memory Map (S2MM) Write Address Channel			
m_axi_s2mm_awaddr	O	32	S2MM Write Address
m_axi_s2mm_awlen	O	8	S2MM Write Length Qualifier
m_axi_s2mm_awsize	O	3	S2MM Write Size Qualifier
m_axi_s2mm_awburst	O	2	S2MM Write Burst Type Qualifier
m_axi_s2mm_awprot	O	3	S2MM Write Protection Qualifier
m_axi_s2mm_awcache	O	4	S2MM Write Cache Qualifier
m_axi_s2mm_awvalid	O	1	S2MM Write Address Valid Qualifier
m_axi_s2mm_awready	I	1	S2MM Write Address Ready Qualifier
AXI4 S2MM Write Data Channel			
m_axi_s2mm_wdata	O	32	S2MM Write Data
m_axi_s2mm_wstrb	O	4	S2MM Write Strobes
m_axi_s2mm_wlast	O	1	S2MM Write Last Indication
m_axi_s2mm_wvalid	O	1	S2MM Write Valid Handshake
m_axi_s2mm_wready	I	1	S2MM Write Ready Handshake

Table 2-1: **Common I/O Signals *(Cont'd)***

Pin Name	Dir	Width	Description
AXI4 S2MM Write Response Channel			
m_axi_s2mm_bresp	I	2	S2MM Write Response Data
m_axi_s2mm_bvalid	I	1	S2MM Write Response Valid Handshake
m_axi_s2mm_bready	O	1	S2MM Write Response Ready Handshake

EDK pCore Interface

The pCore interface creates a core that can be easily added to an EDK Project as a hardware peripheral. This section describes the I/O signals associated with the Object Segmentation pCore.

The I/O signals for the Object Segmentation pCore are shown in Figure 2-1. The signals can be broken into two groups: Common I/O signals and AXI4-Lite signals. The Common I/O signals are specified in Table 2-1. The AXI4-Lite signals are specified in Table 2-2.

Core Signals	
clk fsync_in buffer_ptr	
AXI4 MM2S Interface	
m_axi_mm2s_aclk m_axi_mm2s_arready m_axi_mm2s_rdata m_axi_mm2s_rresp m_axi_mm2s_rlast m_axi_mm2s_rvalid m_axi_mm2s_rrready	m_axi_mm2s_araddr m_axi_mm2s_arlen m_axi_mm2s_arsize m_axi_mm2s_arburst m_axi_mm2s_arprot m_axi_mm2s_arcache m_axi_mm2s_arvalid
AXI4 S2MM Interface	
m_axi_s2mm_aclk m_axi_s2mm_awready m_axi_s2mm_wready m_axi_s2mm_bresp m_axi_s2mm_bvalid	m_axi_s2mm_awaddr m_axi_s2mm_awlen m_axi_s2mm_awsize m_axi_s2mm_awburst m_axi_s2mm_awprot m_axi_s2mm_awcache m_axi_s2mm_awvalid m_axi_s2mm_wdata m_axi_s2mm_wstrb m_axi_s2mm_wlast m_axi_s2mm_wvalid m_axi_s2mm_bready
AXI4-Lite Interface	
S_AXI_ACLK S_AXI_ARESETN S_AXI_AWADDR S_AXI_AWVALID S_AXI_WDATA S_AXI_WSTRB S_AXI_WVALID S_AXI_BREADY S_AXI_ARADDR S_AXI_ARVALID S_AXI_RREADY	IP2INTC_Irpt S_AXI_AWREADY S_AXI_BRESP S_AXI_BVALID S_AXI_ARREADY S_AXI_RDATA S_AXI_RRESP S_AXI_RVALID

Figure 2-1: **pCore I/O Diagram**

Table 2-2: **AXI4-Lite pCore I/O Signals**

Pin Name	Dir	Width	Description
AXI4-Lite Global System Signals (1)			
S_AXI_ARESETN	I	1	AXI4-Lite Reset, active low
IP2INTC_Irpt	O	1	Interrupt request output
AXI4-Lite Write Address Channel Signals (1)			
S_AXI_AWADDR	I	[(C_S_AXI_ADDR_WIDTH-1):0]	AXI4-Lite Write Address Bus. The write address bus gives the address of the write transaction.
S_AXI_AWVALID	I	1	AXI4-Lite Write Address Channel Write Address Valid. This signal indicates that valid write address is available. • 1 = Write address is valid. • 0 = Write address is not valid.
S_AXI_AWREADY	O	1	AXI4-Lite Write Address Channel Write Address Ready. Indicates core is ready to accept the write address. • 1 = Ready to accept address. • 0 = Not ready to accept address.
AXI4-Lite Write Data Channel Signals (1)			
S_AXI_WDATA	I	[(C_S_AXI_DATA_WIDTH-1):0]	AXI4-Lite Write Data Bus.
S_AXI_WSTRB	I	[C_S_AXI_DATA_WIDTH/8-1:0]	AXI4-Lite Write Strobes. This signal indicates which byte lanes to update in memory.
S_AXI_WVALID	I	1	AXI4-Lite Write Data Channel Write Data Valid. This signal indicates that valid write data and strobes are available. • 1 = Write data/strobes are valid. • 0 = Write data/strobes are not valid.
S_AXI_WREADY	O	1	AXI4-Lite Write Data Channel Write Data Ready. Indicates core is ready to accept the write data. • 1 = Ready to accept data. • 0 = Not ready to accept data.

Table 2-2: **AXI4-Lite pCore I/O Signals *(Cont'd)***

Pin Name	Dir	Width	Description
AXI4-Lite Write Response Channel Signals [(1)]			
S_AXI_BRESP [(2)]	O	[1:0]	AXI4-Lite Write Response Channel. Indicates results of the write transfer. • 00b = OKAY - Normal access has been successful. • 01b = EXOKAY - Not supported. • 10b = SLVERR - Error. • 11b = DECERR - Not supported.
S_AXI_BVALID	O	1	AXI4-Lite Write Response Channel Response Valid. Indicates response is valid. • 1 = Response is valid. • 0 = Response is not valid.
S_AXI_BREADY	I	1	AXI4-Lite Write Response Channel Ready. Indicates Master is ready to receive response. • 1 = Ready to receive response. • 0 = Not ready to receive response.
AXI4-Lite Read Address Channel Signals [(1)]			
S_AXI_ARADDR	I	[(C_S_AXI_ADDR_WIDTH-1):0]	AXI4-Lite Read Address Bus. The read address bus gives the address of a read transaction
S_AXI_ARVALID	I	1	AXI4-Lite Read Address Channel Read Address Valid. • 1 = Read address is valid. • 0 = Read address is not valid.
S_AXI_ARREADY	O	1	AXI4-Lite Read Address Channel Read Address Ready. Indicates core is ready to accept the read address. • 1 = Ready to accept address. • 0 = Not ready to accept address.

Table 2-2: **AXI4-Lite pCore I/O Signals *(Cont'd)***

Pin Name	Dir	Width	Description
AXI4-Lite Read Data Channel Signals (1)			
S_AXI_RDATA	O	[(C_S_AXI_DATA_WIDTH-1):0]	AXI4-Lite Read Data Bus.
S_AXI_RRESP (2)	O	[1:0]	AXI4-Lite Read Response Channel Response. Indicates results of the read transfer. • 00b = OKAY - Normal access has been successful. • 01b = EXOKAY - Not supported. • 10b = SLVERR - Error. • 11b = DECERR - Not supported.
S_AXI_RVALID	O	1	AXI4-Lite Read Data Channel Read Data Valid. This signal indicates that the required read data is available and the read transfer can complete. 1 = Read data is valid. 0 = Read data is not valid.
S_AXI_RREADY	I	1	AXI4-Lite Read Data Channel Read Data Ready. Indicates master is ready to accept the read data. • 1 = Ready to accept data. • 0 = Not ready to accept data.

1. The function and timing of these signals are defined in the *AMBA® AXI Protocol Version: 2.0 Specification*.
2. For signals S_AXI_RRESP[1:0] and S_AXI_BRESP[1:0], the core does not generate the Decode Error ('11') response. Other responses like '00' (OKAY) and '10' (SLVERR) are generated by the core based upon certain conditions.

General Purpose Processor Interface

The other interface option is the General Purpose Processor (GPP) interface. The GPP Interface is shown in Figure 2-2 and consists of the Common I/O signals listed in Table 2-1 and the Dynamic Configuration Interface signals detailed in Table 2-3. The signals in Table 2-3 correspond to the registers in Table 2-4.

The directly exposed Dynamic Configuration Interface signals allow the user to wrap these signals with a user-defined bus interface targeting any arbitrary processor. It is recommended to disable the `control[1]` (Register Update enable) signal of the control bus before updating the other Dynamic Configuration Interface signals. After the Dynamic Configuration Interface signals are ready to be updated in the core, the control[1] signal should be enabled. Values are written into the core on the falling edge of the `fsync_in` input.

Core Signals	
clk sclr fsync_in buffer_ptr	
AXI4 MM2S Interface	
m_axi_mm2s_aclk m_axi_mm2s_arready m_axi_mm2s_rdata m_axi_mm2s_rresp m_axi_mm2s_rlast m_axi_mm2s_rvalid m_axi_mm2s_rready	m_axi_mm2s_araddr m_axi_mm2s_arlen m_axi_mm2s_arsize m_axi_mm2s_arburst m_axi_mm2s_arprot m_axi_mm2s_arcache m_axi_mm2s_arvalid
AXI4 S2MM Interface	
m_axi_s2mm_aclk m_axi_s2mm_awready m_axi_s2mm_wready m_axi_s2mm_bresp m_axi_s2mm_bvalid	m_axi_s2mm_awaddr m_axi_s2mm_awlen m_axi_s2mm_awsize m_axi_s2mm_awburst m_axi_s2mm_awprot m_axi_s2mm_awcache m_axi_s2mm_awvalid m_axi_s2mm_wdata m_axi_s2mm_wstrb m_axi_s2mm_wlast m_axi_s2mm_wvalid m_axi_s2mm_bready
Dynamic Configuration Interface	
control image_char_start_addr0 image_char_start_addr1 meta_data_start_addr0 meta_data_start_addr1 label_mask_start_addr feature_select_write_bank_addr feature_select_write_bank_addr_we feature_select_data feature_select_we feature_select_active_bank_addr feature_combination_write_bank_addr_we feature_combination_data feature_combination_we feature_combination_active_bank_addr num_h_blocks num_v_blocks num_total_blocks block_size	reg_update_done status_done status_error mm2s_err s2mm_err status version

Figure 2-2: **General Purpose Processor I/O Diagram**

Table 2-3: **Dynamic Configuration Interface Signals**

Pin Name	Dir	Width	Description	
Control				
sclr	I	1	Synchronous Clear	
reg_update_done	O	1	Register Update Done	
status_done	O	1	Frame Done	
status_error	O	1	Frame Error	
mm2s_err	O	1	MM2S Channel Error	
s2mm_err	O	1	S2MM Channel Error	
Registers				
control	I	4	Control Register	
			3	Buffer Select 0 = Use buffer_ptr to specify the Image Characterization buffer to use. 1 = Toggle between Image Characterization buffers.
			2	Metadata Address Selection 0 = Meta_data_start_addr0 1 = Meta_data_start_addr1
			1	Register Update Enable
			0	Core Enable
image_char_start_addr0	I	32	Image Characterization Start Address 0	
image_char_start_addr1	I	32	Image Characterization Start Address 1	
meta_data_start_addr0	I	32	Metadata Start Address 0	
meta_data_start_addr1	I	32	Metadata Start Address 1	
label_mask_start_addr0	I	32	Label Mask Start Address	
feature_select_write_bank_addr	I	3	Feature Select Write Bank Address (0 -7)	
feature_select_write_bank_addr_we	I	1	Feature Select Write Bank Address Write Enable	
feature_select_data	I	4	Feature Select Data	
feature_select_we	I	1	Feature Select Data Write Enable	
feature_select_active_bank_addr	I	3	Feature Select Active Bank Address (0 – 7)	
feature_combination_write_bank_addr	I	4	Feature Combination Write Bank Address	
			3	Corresponds to Feature Combination Active Bank Address
			2:0	Write Feature Combination Bank Address internal to core for feature combination 0 - 7.
feature_combination_write_bank_addr_we	I	1		

Table 2-3: **Dynamic Configuration Interface Signals *(Cont'd)***

feature_combination_data	I	32		
feature_combination_we	I	1		
feature_combination_active_bank_addr	I	1	Feature Combination Active Bank Address	
num_h_blocks	I	10	Number of Horizontal Blocks	
num_v_blocks	I	10	Number of Vertical Blocks	
num_total_blocks	I	20	Total Number of Blocks (H x V)	
block_size	I	8	Block Size	
status	O	1	Status Register	
			Meta Data Address. Specifies which buffer is active. 0 = Meta_data_start_addr0 1 = Meta_data_start_addr1	
version	O	32	Version Register	
			31:28	Version Major
			27:20	Version Minor
			19:16	Version Revision
			15:0	Reserved

Register Space

The pCore interface provides a memory-mapped interface for the programmable registers within the core, which are defined in Table 2-4.

Table 2-4: **Object Segmentation pCore Memory Mapped Register Set**

Address (hex) BASEADDR +	Register Name	Access Type	Description	
0x0000	Control	R/W	Control Register	
			31:4	Reserved
			3	Buffer Selection 0 = Use Buffer Ptr input to specify the Image Characterization Buffer to be read 1 = Toggle between the Image Characterization Buffers. Begin with buffer 0.
			2	Meta Data Address Selection 0 = Meta Data Start Addr 0, 1 = Meta Data Start Addr 1
			1	Register Update Enable. This bit communicates to the IP Core to take new values at the next fsync_in rising edge. Usage: This bit is cleared when the IP Core next fsync_in happens.
			0	Enable the Object Segmentation core on the next fsync_in
0x0004	Status	R	Status Register	
			31:1	Reserved
			0	Meta Data Address. Specifies which buffer is actively being written to: 0 = Meta Data Start Addr 0, 1 = Meta Data Start Addr 1
0x0008	Status Error	R	Status Register for Errors	
			31:3	Reserved
			2	MM2S Error. This active high signal is asserted whenever a Error condition is encountered within the MM2S.
			1	S2MM Error. This active high signal is asserted whenever a Error condition is encountered within the S2MM.
			0	Frame Error The core did not finish before the beginning of the next frame. Usage: This bit is cleared when any value is written to the register.

Table 2-4: **Object Segmentation pCore Memory Mapped Register Set *(Cont'd)***

0x000C	Status Done	R	General read register for status done	
			31:1	Reserved
			0	Frame Done Done bit can be polled by software for end of object segmentation operation. Usage: This bit is cleared when any value is written to the register
0x0010	Image Characterization Start Address 0	R/W	Starting address for input Image Characterization Buffer 0	
			31:0	
0x0014	Image Characterization Start Address 1	R/W	Starting address for input Image Characterization Buffer 1	
			31:0	
0x0018	Metadata Start Address 0	R/W	Starting address for output Metadata Buffer 0	
			31:0	
0x001C	Metadata Start Address 1	R/W	Offset address for output Metadata Buffer 1	
			31:0	
0x0020	Label Mask Start Address 0	R/W	Starting address for label mask output	
			31:0	
0x0024	Reserved			
0x0028	Reserved			
0x002C	Reserved			
0x0030	Feature Select Write Bank Address	R/W	Bank address of feature select being written	
			31:3	Reserved
			2:0	Bank address to which the Feature Select Data are written
0x0034	Feature Select Data	R/W	Feature Select input data truth table	
			31:4	Reserved
			3:0	Data for Feature Select truth table, 256 values per bank
0x0038	Feature Select Active Bank Address	R/W	Active Feature Select Bank	
			31:3	Reserved
			2:0	Feature Select Bank for next frame
0x003C	Feature Combination Write Bank Addr	R/W	Bank address of feature combination data being written	
			31:4	Reserved
			3	Corresponds to Feature Combination Active Bank Address
			2:0	Write Feature Combination Bank Address internal to core for feature combination 0 - 7.

Table 2-4: **Object Segmentation pCore Memory Mapped Register Set *(Cont'd)***

0x0040	Feature Combination Data	R/W	Feature Combination input data for thresholds	
			31:0	Data for Feature Combination Thresholds, 40 values per bank.
0x0044	Feature Combination Active Bank Addr	R/W	Active feature combination bank for next frame	
			31:1	Reserved
			0	Active Feature Combination Bank to be use for the next frame.
0x0048	Number of Horizontal Blocks	R/W	Number of horizontal blocks in the input data set	
			31:10	Reserved
			9:0	Number of horizontal blocks in the system, that is, horizontal resolution divided by block size
0x004C	Number of Vertical Blocks	R/W	Number of vertical blocks in the input data set	
			31:10	Reserved
			9:0	Number of vertical blocks in the system, that is, vertical resolution divided by block size
0x0050	Number of Total Blocks	R/W	Number of total blocks in the input data set	
			31:20	Reserved
			19:0	Number of total blocks in the system, that is, number of horizontal blocks * number of vertical blocks
0x0054	Block Size	R/W	Block size of VA system	
			31:8	Reserved
			7:0	Block Size
0x00F0	Version Register	R	Version Register	
			31:28	Version Major
			27:20	Version Minor
			19:16	Version Revision
			15:0	Reserved
0x0100	Software Reset	R/W	Software Reset	
			31:1	Reserved
			0	1 = reset core
0x021C	GIER	R/W	Global Interrupt Enable	
			31	Mask to enable global interrupts
			30:0	Reserved

Table 2-4: **Object Segmentation pCore Memory Mapped Register Set *(Cont'd)***

0x0220	ISR	R/W	Interrupt Status Register Read to determine the source of the interrupt Write to clear the interrupt	
			31:4	Reserved
			3	Edge sensitive interrupt for MM2S Error
			2	Edge sensitive interrupt for S2MM Error
			1	Edge sensitive interrupt for Frame Done
			0	Edge sensitive interrupt for Frame Error
0x0228	IER	R/W	Interrupt Enable Register 0 = mask out an interrupt 1 = enable an interrupt	
			31:4	Reserved
			3	Mask or Enable for MM2S Error
			2	Mask or Enable for S2MM Error
			1	Mask or Enable for Frame Done
			0	Mask or Enable for Frame Error

Chapter 3

Customizing and Generating the Core

This chapter includes information on using Xilinx tools to customize and generate the core.

Graphical User Interface (GUI)

CORE Generator Software GUI

The Xilinx® Image Characterization core is easily configured to meet the developer's specific needs through the CORE Generator™ software GUI. This section provides a quick reference to the parameters that can be configured at generation time. The GUI is shown in Figure 3-1.

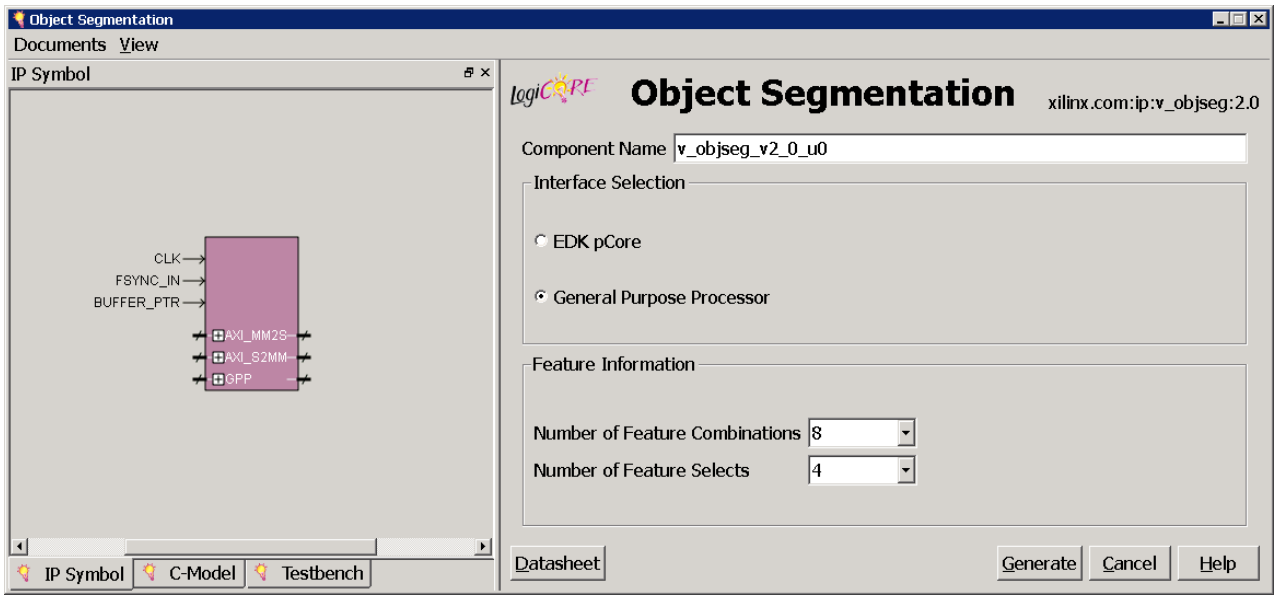

Figure 3-1: **Object Segmentation CORE Generator GUI**

The screen displays a representation of the IP symbol on the left side, and the parameter assignments on the right side, described as follows:

- **Component Name**: The component name is used as the base name of output files generated for the module. Names must begin with a letter and must be composed from characters: a to z, 0 to 9, and "_".

  Note: The name "v_objseg_v2_0" is not allowed.

- **Interface Selection**: The Image Characterization core is generated with one of two processor interfaces.
  - **EDK pCore Interface**: CORE Generator software generates the core as a pCore that can be easily imported into an EDK project as a hardware peripheral. The core registers can then be programmed in real-time via an embedded microprocessor. See the EDK pCore Interface section for details. When the EDK pCore is selected, the rest of the options are disabled and set to the default value. All modifications to the Object Segmentation pCore are made with the EDK GUI.
  - **General Purpose Processor Interface**: CORE Generator software generates a set of ports that can be used to program the core. See the General Purpose Processor Interface section for details. When the General Purpose Processor interface is selected, the rest of the configuration options become active and can be used to generate a customized Object Segmentation core.
- **Feature Information**
  - **Number of Feature Combinations**: Sets the number of Feature Combination units that are instantiated in the core. The range of valid choices is 1 - 8. The higher the selected value the more resources that are used.
  - **Number of Feature Selects**: Sets the number of Feature Select units that are instantiated in the core. The range of valid choices is 1 - 4. The higher the selected value the more resources that are used.

pCore Generation in the CORE Generator Software

When generated by the CORE Generator software, the new pCore is located in the CORE Generator software project directory at <Component_Name>/pcores/axi_objseg_v2_00_a. The pCore should be copied to the user's <EDK_Project>/pcores directory or to a user pCores repository. The Object Segmentation pCore driver software is located in the CORE Generator project directory at <Component_Name>/drivers/os_v2_00_a. The driver software should be copied to the user's <EDK_Project>/drivers directory or to a user pCores repository.

EDK pCore Graphical User Interface (GUI)

When the Xilinx Object Segmentation core is generated from the CORE Generator software as an EDK pCore, it is generated with each option set to the default value. All customizations of an Object Segmentation pCore are done with the EDK pCore graphical user interface (GUI). Figure 3-2 illustrates the EDK pCore GUI for the Object Segmentation pCore. All of the options in the EDK pCore GUI for the Object Segmentation core correspond to the same options in the CORE Generator software GUI. See CORE Generator Software GUI for details about each option.

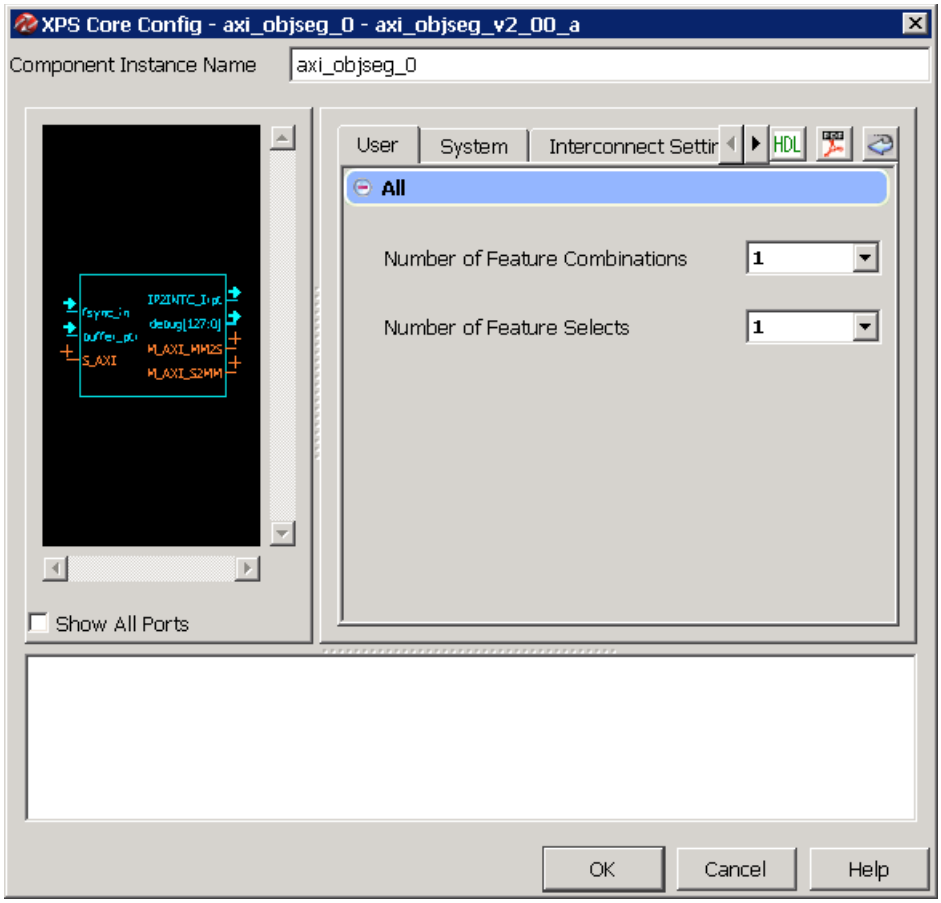

Figure 3-2: **Object Segmentation pCore GUI**

Parameter Values in the XCO File

Table 3-1 defines valid entries for the Xilinx CORE Generator (XCO) software parameters. Xilinx strongly suggests that XCO parameters are not manually edited in the XCO file; instead, use the CORE Generator software GUI to configure the core and perform range and parameter value checking. The XCO parameters are helpful in defining the interface to other Xilinx tools.

Table 3-1: **XCO Parameters**

XCO Parameter	Default	Valid Values
component_name	v_objseg_v2_0_u0	ASCII text using characters: a..z, 0..9 and "_" starting with a letter. ***Note:*** "v_objseg_v2_0" is not allowed.
interface_selection	EDK_pCore	EDK_pCore, General_Purpose_Processor
num_feature_combinations	8	1-8
num_feature_selects	4	1-4

Output Generation

The output files generated from the Xilinx CORE Generator software for the Object Segmentation core depend upon whether the interface selection is set to EDK pCore or General Purpose Processor. The output files are placed in the project directory.

EDK pCore Files

When the interface selection is set to EDK pCore, the CORE Generator tool then outputs the core as a pCore that can be easily incorporated into an EDK project. The pCore output consists of a hardware pCore and a software driver. The pCore has the following directory structure:

- <Component_Name>
  - drivers
    - os_v2_00_a
      - data
      - doc
        - html
          - api
      - example
      - src
  - pcores
    - axi_objseg_v2_00_a
      - data
      - hdl
        - vhdl

File Details

- <project directory>

  This is the top-level directory. It contains xco and other assorted files.

Name	Description
<component_name>.xco	Log file from CORE Generator software describing which options were used to generate the core. An XCO file can also be used as an input to the CORE Generator software.
<component_name>_flist.txt	A text file listing all of the output files produced when the customized core was generated in the CORE Generator software.

- <project directory>/<component_name>/pcores/axi_objseg_v2_00_a/data

  This directory contains files that EDK uses to define the interface to the pCore.

- < project directory>/<component_name>/pcores/axi_objseg_v2_00_a/hdl/vhdl

  This directory contains the Hardware Description Language (HDL) files that implement the pCore.

- < project directory>/<component_name>/drivers/os_v2_00_a/data

  This directory contains files that Software Development Kit (SDK) uses to define the operation of the pCore's software driver.
- < project directory>/<component_name>/drivers/os_v2_00_a/doc/html/api

  This directory contains HTML documentation files for the pCore's software driver.
- < project directory>/<component_name>/drivers/os_v2_00_a/src

  This directory contains the source code of the pCore's software driver.

Name	Description
xos.c	Provides the Application Program Interface (API) access to all features of the Object Segmentation device driver.
xos.h	Provides the API access to all features of the Object Segmentation device driver.
xos_g.c	Contains a template for a configuration table of Object Segmentation core.
xos_hw.h	Contains identifiers and register-level driver functions (or macros) that can be used to access the Object Segmentation core.
xos_intr.c	Contains interrupt-related functions of the Object Segmentation device driver.
xos_sinit.c	Contains static initialization methods for the Object Segmentation device driver.

General Purpose Processor Files

When the interface selection is set to General Purpose Processor, the CORE Generator tool outputs the core as a netlist that can be inserted into a processor interface wrapper or instantiated directly in an HDL design. The output is placed in the <project directory>.

File Details

The CORE Generator software output consists of some or all the following files.

Name	Description
<component_name>_readme.txt	Readme file for the core.
<component_name>.ngc	The netlist for the core.
<component_name>.veo <component_name>.vho	The HDL template for instantiating the core.
<component_name>.v <component_name>.vhd	The structural simulation model for the core. It is used for functionally simulating the core.
<component_name>.xco	Log file from CORE Generator software describing which options were used to generate the core. An XCO file can also be used as an input to the CORE Generator software.
<component_name>_flist.txt	A text file listing all of the output files produced when the customized core was generated in the CORE Generator software.

<component_name>.asy	IP symbol file.
<component_name>.gise <component_name>.xise	ISE® software subproject files for use when including the core in ISE software designs.

XILINX®

Chapter 4

Designing with the Core

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

Architecture

A high-level view of the Object Segmentation core is shown in Figure 4-1. The core uses an AXI4 interface to transfer data between the core and buffers in external memory. The Object Segmentation core uses a two-phase architecture to find and segment objects in a video frame.

In the first phase, the core inputs the Image Characterization data structure and compares it against the Feature Combination threshold values. The results of that processing are combined to form the Feature Select results. The Feature Select results are processed to create labeled regions within the data structure. This label data is written to external memory and the list of labeled regions is processed to aggregate neighboring labels into an object. After this is done, the second phase of processing begins by reading the label data from external memory and remapping the labels into objects. After an object is defined, the statistics of the object are calculated and written out as Metadata.

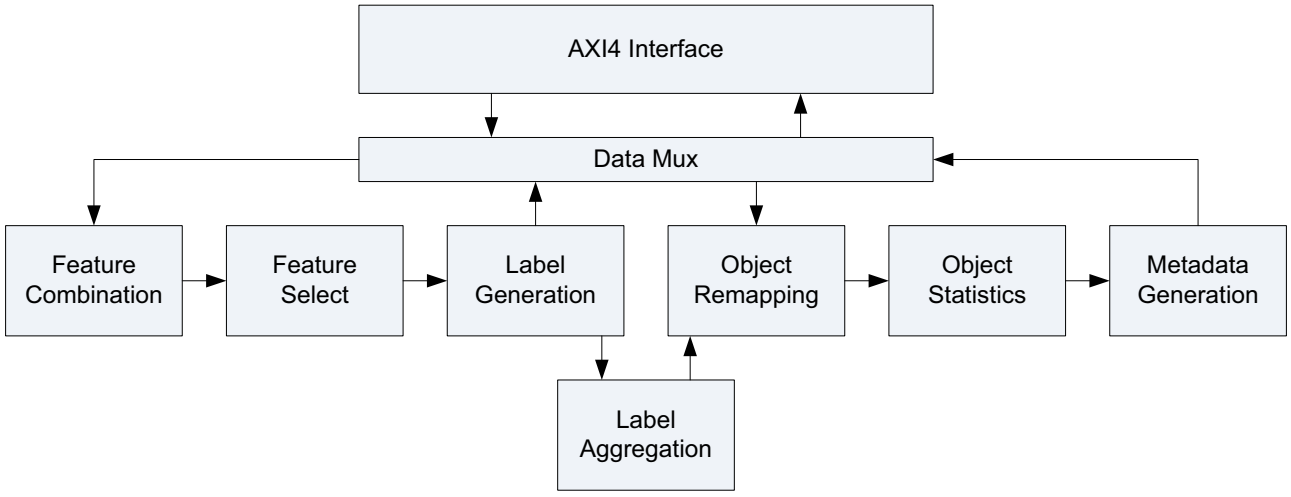

Figure 4-1: **Object Segmentation Block Diagram**

Feature Combination

A Feature Combination (FC) is defined in the Feature Combination Threshold Data Structure section. The Feature Combination Data Threshold Structure consists of a set of threshold values with a lower and an upper bound. The Feature Combination Lower Global Thresholds and Upper Global Thresholds in Table 4-2 match the Image Characterization IP core Global Statistics output as shown in Table 4-6. Additionally, the Feature Combination Lower Block Thresholds and Upper Block Thresholds in Table 4-2 match the Image Characterization IP core output Block Statistics shown in Table 4-7. A Feature Combination unit must be properly initialized with a Feature Combination data structure before it can begin processing.

A Feature Combination unit is implemented as a set of comparators. It compares the Image Characterization data input against the Feature Combination data structure that is loaded by the user. Each value in the Image Characterization data structure is compared against its corresponding threshold values in the Feature Combination data structure. A value passes the comparison if it meets the following criteria:

FC Lower Threshold < Image Characterization value < FC Upper Threshold

The first portion of the Image Characterization data that is read is the Global Statistics. As the Global Statistics are read in they are compared against the Feature Combination Global Thresholds. If any of the Global Statistics fail to match the threshold ranges, the entire frame is considered invalid and no objects will be found in the image. If all of the Global Statistics match the Global Statistics Thresholds, then processing advances to the Block data processing.

Next the Image Characterization Block Data is tested against the Feature Combination Block Thresholds. For each Image Characterization block a 1-bit result is calculated. If all of the statistics for an Image Characterization Block match the Feature Combination Block Thresholds, the block is given a value of '1'. If one or more of the statistics for an Image Characterization Block fails to match the Feature Combination Block Thresholds, the block is given a value of '0'.

Up to eight separate Feature Combination units are supported by the Object Segmentation core. Each Feature Combination unit is a separate entity and must be properly initialized with a unique Feature Combination data structure. Each Feature Combination unit generates a 1-bit result for each block in the Image Characterization data structure. These eight 1-bit values are passed to the Feature Select Generation block for further processing. If less than eight Feature Combination units are instantiated, the results are padded with '0's in the MSB to make 8-bits.

Feature Select

The Feature Select block takes the eight 1-bit values from the Feature Combination block and transforms them into a Feature Select. A Feature Select is defined as any logical equation using the Feature Combination results as terms in the equation.

A Feature Select is implemented as a 1-bit x 256-entry look-up table. The 8 bits of Feature Combination data are used as an address to this look-up table. The look-up table is used as a truth table. The initialization of the look-up table determines the logical equation that defines the Feature Select. See Feature Select Data Structure for more details.

The Feature Select block supports up to four Feature Selects. Each Feature Select generates a 1-bit result for each block in the original Image Characterization data structure. These four 1-bit results are passed on to the Label Generation block for further processing.

Label Generation

The Label Generation block is used to aggregate neighboring blocks with Feature Select values of '1'. As each block is examined, if it is a value of '0' then it is ignored because it did not test positive for Feature Select. If a block has a value of '1', it is assigned a label value. If the block has a neighbor to its left, left upper diagonal, upper, or right upper diagonal that already has a label value then the new block is assigned the same label value. If none of the blocks neighbors has a label value then the block is assigned a new label value. Figure 4-2 shows an example Feature Select result for an 8x8 block image. The blocks with a value of '1' passed the Feature Select processing. Empty blocks did not pass the Feature Select processing. Figure 4-3 shows the results of the Label Generation processing. The Feature Select data is aggregated into four sets of labels. Labels 1, 3 and 4 are neighbors.

1							1
1	1	1	1			1	1
1				1			1
1		1		1			1
		1		1			
		1			1		
		1	1	1			
1	1	1					

Figure 4-2: **Feature Select Results for an 8x8 Frame**

1							2
1	1	1	1			2	2
1				1			2
1		3		1			2
		3		1			
		3			1		
		3	3	3			
4	4	4					

Figure 4-3: **Label Data for an 8x8 Frame**

There is a separate Label Generation circuit for each Feature Select. Label Generation circuits run in parallel with each other. The Label data is written to an external memory buffer to complete the first pass of processing.

Label Aggregation

The Label Aggregation block is active between the end of the first pass and the start of the second pass of processing. The Label Aggregation is responsible for finding labels that are neighbors and aggregating them into an object. The Label Generation block reports out a list of labels that are neighbors. The Label Aggregation block recursively searches the list for all labels that are connected and adds them to a list of new object values. This list of object values is used during the second pass processing to remap labels into objects. There is a separate Label Aggregation circuit for each Label Generation circuit. The Object Segmentation core supports up to four Label Aggregation circuits.

Continuing with the example used in the previous section, the Label Generation block reports that there are four labels (1, 2, 3 and 4) and that 3 connects to 1 and 4 connects to 3. The Label Aggregation block processes this list and remaps the labels 1, 3 and 4 to object A and label 2 to object B. This new object mapping is passed to the Object Remapping block.

Object Remapping

After the Label Aggregation block is finished consolidating labels into objects, the Object Remapping block uses the object mapping list to assign new object values to the label data. As the label data is read from memory, the Object Remapping block looks at the label assigned to a block and remaps the block to a new object value. Figure 4-4 and Figure 4-5 illustrate the remapping process for the example use in the previous sections.

1							2
1	1	1	1			2	2
1				1			2
1		3		1			2
		3		1			
		3			1		
		3	3	3			
4	4	4					

Figure 4-4: **Label Data for an 8x8 Frame**

A							B
A	A	A	A			B	B
A				A			B
A		A		A			B
		A		A			
		A			A		
		A	A	A			
A	A	A					

Figure 4-5: **Object Data for an 8x8 Frame**

There is a separate Object Remapping circuit for each instantiated Feature Select for up to four independent circuits.

Object Statistics

The new object data is passed along to the Object Statistics block. The Object Statistics block keeps track of the size of each object and calculates the coordinates of a rectangle that would completely bound the object. The Object Statistics block also counts the number of blocks that make up an object. For Figure 4-5, the Object Statistics block would count 20 blocks for object A and 5 blocks for object B.

There is a separate Object Statistics circuit for each instantiated Feature Select.

Metadata Generation

The calculated object statistics are passed from the Object Statistics block to the Metadata Generation block. This block is responsible for taking the object statistics for each of the Object Statistics circuits and formatting the data into the Object Segmentation Metadata that is written to external memory. Metadata is written for each instantiated Feature Select.

Data Structures

The Object Segmentation core uses several different data structures. The Feature Combination Threshold data structure and the Feature Select data structure define the data that is used to initialize the Object Segmentation core prior to processing data. The Image Characterization data structure defines the format of the input data that the core processes. The Object Segmentation Metadata data structure defines the format of the output data that the core produces for each frame of Image Characterization data.

Feature Combination Threshold Data Structure

The Object Segmentation core supports up to eight Feature Combination units. Each Feature Combination unit requires the input of a set of threshold values. The set of threshold values is defined by the Feature Combination Threshold data structure. The data structure consists of a lower threshold and an upper threshold for each of the global and block statistics in the Image Characterization Data Structure that is defined in the Image Characterization Data Structure section.

The global and block mean ("_mean") statistics are 8-bit values and the thresholds can be any value from 0 to 255. The global and block variance ("_var") statistics are 16-bit values and the thresholds can have any value from 0 to 65535. Each block color_select value represents the number of pixels in the block that matched the specified color range for that color_select in the Image Characterization core. The color_select's value range is determined by the block size used to produce the Image Characterization data, and by the video format that is processed. Table 4-1 shows the possible value ranges for color_selects.

Table 4-1: **Value Ranges for Image Characterization Color_Selects**

Block Size	Video Format 4:2:2	Video Format 4:2:0
64x64	0 to 2048	0 to 1024
32x32	0 to 512	0 to 256
16x16	0 to 128	0 to 64
8x8	0 to 32	0 to 16
4x4	0 to 8	0 to 4

Each Feature Combination unit must be properly initialized before it can be used to process input data. Table 4-2 shows the format of the Feature Combination Threshold data structure and the order in which each element should be loaded. Each Feature Combination unit is loaded independently. There are two memory banks associated with each Feature Combination unit, with the active bank specified by a register value. This configuration allows the user to load two different Feature Combination sets and then switch between the two in real-time by changing the active bank register selection. Changes to the active bank register are acted upon at the beginning of each frame. This configuration also allows the user to calculate a new Feature Combination set, load the new set into the inactive Feature Combination bank and then make the new set the active bank in real-time. See Feature Combination Bank Programming for more details.

Table 4-2: **Feature Combination Threshold Data Structure**

Line #	Byte 3	Byte 2	Byte 1	Byte 0
	Lower Global Threshold			
0x0	Low_Freq_Mean	V_mean	U_Mean	Y_Mean
0x1	Saturation_Mean	Motion_mean	Edge_Mean	High_Freq_Mean
0x2	U_Var		Y_Var	
0x3	Low_Freq_Var		V_Var	
0x4	Edge_Var		High_Freq_Var	
0x5	Saturation_Var		Motion_Var	
	Lower Block Threshold			
0x6	Low_Freq_Mean	V_mean	U_Mean	Y_Mean
0x7	Saturation_Mean	Motion_mean	Edge_Mean	High_Freq_Mean
0x8	U_Var		Y_Var	
0x9	Low_Freq_Var		V_Var	
0xA	Edge_Var		High_Freq_Var	
0xB	Saturation_Var		Motion_Var	
0xC	Color_Sel_2		Color_Sel_1	
0xD	Color_Sel_4		Color_Sel_3	
0xE	Color_Sel_6		Color_Sel_5	
0xF	Color_Sel_8		Color_Sel_7	
0x10	Reserved			
0x11	Reserved			

Table 4-2: **Feature Combination Threshold Data Structure *(Cont'd)***

0x12	Reserved			
0x13	Reserved			
	Upper Global Threshold			
0x14	Low_Freq_Mean	V_mean	U_Mean	Y_Mean
0x15	Saturation_Mean	Motion_mean	Edge_Mean	High_Freq_Mean
0x16	U_Var		Y_Var	
0x17	Low_Freq_Var		V_Var	
0x18	Edge_Var		High_Freq_Var	
0x19	Saturation_Var		Motion_Var	
	Upper Block Threshold			
0x1A	Low_Freq_Mean	V_mean	U_Mean	Y_Mean
0x1B	Saturation_Mean	Motion_mean	Edge_Mean	High_Freq_Mean
0x1C	U_Var		Y_Var	
0x1D	Low_Freq_Var		V_Var	
0x1E	Edge_Var		High_Freq_Var	
0x1F	Saturation_Var		Motion_Var	
0x20	Color_Sel_2		Color_Sel_1	
0x21	Color_Sel_4		Color_Sel_3	
0x22	Color_Sel_6		Color_Sel_5	
0x23	Color_Sel_8		Color_Sel_7	
0x24	Reserved			
0x25	Reserved			
0x26	Reserved			
0x27	Reserved			

Feature Select Data Structure

The Object Segmentation core supports up to four Feature Select units. A Feature Select unit takes the 1-bit results from each of the Feature Combination units and applies a logical expression to them. Each Feature Combination unit is represented as a separate entity in the logical expression. All logical expressions are supported.

A Feature Select unit is implemented as a 1-bit x 256 entry RAM. The eight Feature Combination units are used as address bits to the RAM. Feature Combination 1 is the Least Significant Bit (LSB) and Feature Combination 8 is the Most Significant Bit (MSB) of the address. The RAM creates a look-up table that can be used as a truth-table and therefore allows the implementation of any logical expression of the eight Feature Combinations.

Because up to four Feature Selects are supported, each is implemented as one bit in a 4-bit x 256 entry RAM. Feature Select 1 is the LSB and Feature Select 4 is the msb of the Random Access Memory (RAM) output data as illustrated in Table 4-3. The Object Segmentation core supports eight banks of Feature Select data. The active bank is selected through an active bank register. This configuration allows the user to load multiple banks of Feature Select data and then switch between the banks in real-time. Changes to the active bank selection are processed at the beginning of a frame. Each Feature Select bank is loaded independently which allows the loading of new Feature Select data at any time. See Feature Select Bank Programming for more details.

Table 4-3: **Feature Select Data Structure**

Feature Combinations(8:1)	Bit 3	Bit 2	Bit 1	Bit 0
0x00 (00000000)	FS4_0	FS3_0	FS2_0	FS1_0
0x01 (00000001)	FS4_1	FS3_1	FS2_1	FS1_1
0x02 (00000010)	FS4_2	FS3_2	FS2_2	FS1_2
…	…	…	…	…
0xFD (11111101)	FS4_253	FS3_253	FS2_253	FS1_253
0xFE (11111110)	FS4_254	FS3_254	FS2_254	FS1_254
0xFF (11111111)	FS4_255	FS3_255	FS2_255	FS1_255

Image Characterization Data Structure

The Image Characterization Data Structure defines how the image characterization statistics are organized in external memory. The data structure is made up of three pieces which are located contiguously in memory:

- Frame Header (Table 4-5)
- Global Statistics and Histograms (Table 4-6)
- Block Statistics (Table 4-7)

The Frame Header, Global Statistics and Histograms are all static in size. The size of the Block Statistics structure is dependent on the number of blocks in the processed image. There will be one instance of the Block Statistics data structure for each block in the image. The Block Statistics data structures are arranged contiguously in memory. The order of the blocks corresponds to traversing through the blocks from left to right and from top to bottom.

The values in the data structure use these bit widths:

- Mean ("_mean"): 8-bits
- Variance ("_var"): 16-bits
- Histogram: 32-bits (21-bits actual)
- Color_Select: 16-bits (12-bits actual)
- PAD: 32-bits (0x0000)

Table 4-4: **Image Characterization Data Structure**

Byte 3	Byte 2	Byte 1	Byte 0
Frame Header (32 words)			
Global Statistics (32 words)			
Histograms (1024 words)			
Block Statistics – Block #1 (14 words)			
…			
Block Statistics – Block # HxV (14 words)			

Table 4-5: **Image Characterization Data Structure Frame Header**

Byte 3	Byte 2	Byte 1	Byte 0
Struct_Valid			
Frame_Index			
PAD (x30)			

Table 4-6: **Image Characterization Data Structure Global Statistics**

Byte 3	Byte 2	Byte 1	Byte 0
Low_Freq_Mean	V_mean	U_Mean	Y_Mean
Saturation_Mean	Motion_mean	Edge_Mean	High_Freq_Mean
U_Var		Y_Var	
Low_Freq_Var		V_Var	
Edge_Var		High_Freq_Var	
Saturation_Var		Motion_Var	
PAD (x26)			
Y_Histogram (x256)			
U_Histogram (x256)			
V_Histogram (x256)			
Hue_Histogram (x256)			

Table 4-7: **Image Characterization Data Structure Block Statistics**

Byte 3	Byte 2	Byte 1	Byte 0
Low_Freq_Mean	V_mean	U_Mean	Y_Mean
Saturation_Mean	Motion_mean	Edge_Mean	High_Freq_Mean
U_Var		Y_Var	
Low_Freq_Var		V_Var	
Edge_Var		High_Freq_Var	
Saturation_Var		Motion_Var	
Color_Sel_2		Color_Sel_1	
Color_Sel_4		Color_Sel_3	
Color_Sel_6		Color_Sel_5	
Color_Sel_8		Color_Sel_7	
Reserved			
Reserved			
Reserved			
Reserved			

Note: The Block Statistics repeats once for each block in the image. For example, a 1280x720 image with block size 16 would result in 3600 contiguous instances of Block Statistics data.

Metadata Data Structure

The Metadata data structure contains all of the data calculated by the Object Segmentation core to describe the objects found in the current Image Characterization data structure. The Metadata data structure is a list of up to 31 objects described for up to four Feature Selects resulting to a total of 124 objects that can be found for each frame.

The Metadata data structure begins with a header that is specified in Table 4-9. The Structure_Valid entry specifies whether the entire frame has been written and is valid. It holds a value of 0x00000001 if the data structure is incomplete. It holds a value of 0xFFFFFFFF if the data structure is complete. The Frame Index is a unique identifier for each frame. The values for the number of objects correspond to the number of objects in the Metadata data structure for the specified value. The data structure header also includes the Global Image Characterization statistics that were copied directly from the Image Characterization data structure.

After the Metadata header, the objects for FS1 are provided followed by the objects for FS2, FS3 and FS4. The Object Segmentation core only writes out metadata for the Feature Selects that are instantiated in the core. If only two Feature Selects are instantiated, then only the metadata for FS1 and FS2 is written.

The metadata associated with an object is defined in Table 4-10, and consists of the following information: FS#, object number, X/Y start/stop values, X/Y centroid values, object density and object identifier. The FS# corresponds to the Feature Select that the object belongs to. The object number is the number of the object in the list of objects for that FS. The X/Y start/stop values define the coordinates of the bounding box that surrounds the object. The X/Y centroid values are the coordinates of the center of the bounding box.

The object density is the number of blocks inside the bounding box that belong to the object. The object identifier is a unique value specified for each object.

Table 4-8: **Object Segmentation Metadata Data Structure**

Byte 3	Byte 2	Byte 1	Byte 0
Metadata Header (32 words)			
FS1 Object Data (32 instances)			
FS2 Object Data (32 instances)			
FS3 Object Data (32 instances)			
FS4 Object Data (32 instances)			

Table 4-9: **Object Segmentation Metadata Header**

Byte 3	Byte 2	Byte 1	Byte 0
Struct_Valid			
Frame_Index			
Total_Num_Objects			
FS4_num_obj	FS3_num_obj	FS2_num_obj	FS1_num_obj
Low_Freq_Mean	V_mean	U_Mean	Y_Mean
Saturation_Mean	Motion_mean	Edge_Mean	High_Freq_Mean
U_Var		Y_Var	
Low_Freq_Var		V_Var	
Edge_Var		High_Freq_Var	
Saturation_Var		Motion_Var	
PAD (x22)			

Note: The Mean and Variance values are the global values from the corresponding Image Characterization data structure.

Table 4-10: **Object Segmentation Metadata Object Data**

Byte 3	Byte 2	Byte 1	Byte 0
FS#		Object_Number	
X_stop		X_start	
Y_stop		Y_start	
Y_centroid		X_centroid	
Object_density			
Object_identifier			

Note: Repeat 6 object words for total of 32 objects. Object_number = 0 signals end of objects for this Feature Select. The remaining objects to 32 will = 0.

General Design Guidelines

Object Segmentation Control and Timing

The Object Segmentation core provides a great deal of operational flexibility through a simple register set. The Feature Combinations and Feature Selects define the operation of the core. These features are fully configurable and can be updated with configurations in real-time. As a result, the Object Segmentation core must be properly initialized before it can be used to process Image Characterization data. Each of the instantiated Feature Combinations and Feature Selects must be initialized valid data structures as described in the Feature Combination Threshold Data Structure and Feature Select Data Structure sections.

The process of programming the Feature Combinations and Feature Selects is slightly different depending upon whether the core is generated with an EDK pCore interface or a General Purpose Processor interface. Both methods are discussed in the following sections. The process of the programming the rest of the register set is essentially the same for both interfaces.

pCore Bank Programming

Feature Select Bank Programming

The Object Segmentation core supports eight banks of Feature Select data regardless of the number of Feature Selects that are instantiated. All of the Feature Select banks can be loaded at any time. It is the user's responsibility to verify that a bank is not loaded while it is in active use.

To load the Feature Selects:

1. Set the Feature Select Write Bank Address to the address (0 – 7) of the bank to be loaded.
2. Write 256 Feature Select data values to the Feature Select Data register. See the Feature Select Data Structure section for more details.
3. Repeat steps 1 and 2 for any additional Feature Select banks to be loaded.
4. Specify the active Feature Select bank by writing the bank's address to the Feature Select Active Bank Address register.

Feature Combination Bank Programming

The Object Segmentation core supports up to eight Feature Combinations. Each Feature combination is implemented with two banks of Feature Combination data. This makes for a total of up to 16 Feature Combination banks that can be independently loaded. All of the Feature Combination banks can be loaded at any time. It is the user's responsibility to verify that banks are not loaded while they are active.

To load the Feature Combinations:

1. Set the Feature Combination Write Bank Address to the address (0 –15) of the Feature Combination bank to be loaded. See Table 2-4 or Table 2-3 for more detail on the Feature Combination Write Bank Address register.
2. Write 40 Feature Combination data values to the Feature Combination Data register. See the Feature Combination Threshold Data Structure section for more detail.
3. Repeat steps 1 and 2 for any additional Feature Combination banks to be loaded.

4. Specify the active Feature Combination bank by writing the banks address to the Feature Combination Active Bank Address register.

GPP Bank Programming

Feature Select Bank Programming

The Object Segmentation core supports eight banks of Feature Select data regardless of the number of Feature Selects that are instantiated. All of the Feature Select banks can be loaded at any time. It is the user's responsibility to verify that a bank is not loaded while it is in active use. See Figure 4-6.

To load the Feature Selects:

1. Set the Feature Select Write Bank Address to the address (0 – 7) of the bank to be loaded.
2. Write 256 Feature Select data values to the feature_select_data. The `feature_select_we` signal must be toggled for the data to be written. See the Feature Select Data Structure section for more details about the Feature Select data set.
3. Repeat steps 1 and 2 for any additional Feature Select banks to be loaded.
4. Specify the active Feature Select bank by writing the bank's address to the Feature Select Active Bank Address register.

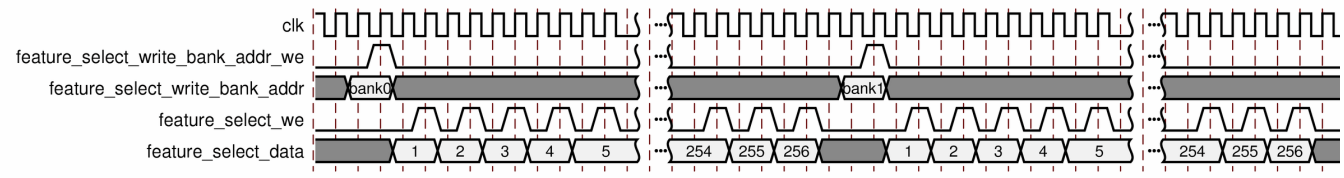

Figure 4-6: **Feature Select Bank Programming**

Feature Combination Bank Programming

The Object Segmentation core supports up to eight Feature Combinations. Each Feature Combination is implemented with two banks of Feature Combination data. This makes for a total of up to 16 Feature Combination banks that can be independently loaded. All of the Feature Combination banks can be loaded at any time. It is the user's responsibility to verify that banks are not loaded while they are active. See Figure 4-7.

To load the Feature Combinations:

1. Set the Feature Combination Write Bank Address to the address (0 –15) of the Feature Combination bank to be loaded. See Table 2-4 or Table 2-3 for information about the Feature Combination Write Bank Address register.

1. Write 40 Feature Combination data values to the feature_combination_data. The `feature_combination_we` signal must be toggled for the data to be written. See the Feature Combination Threshold Data Structure section for more details on the Feature Combination Threshold data set.
2. Repeat steps 1 and 2 for any additional Feature Combination banks to be loaded.

Specify the active Feature Combination bank by writing the banks address to the Feature Combination Active Bank Address register

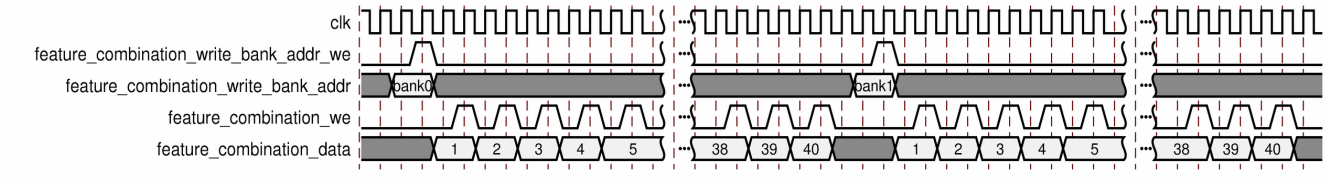

Figure 4-7: **Feature Combination Bank Programming**

Register Updates

The Object Segmentation core is controlled by a register set that must be initialized before the core begins processing. See Table 2-4 for more detail about the Object Segmentation register set. The registers should be initialized as follows:

1. Set the register update enable bit of the control register (bit 1) to '0' to disable register updates.
2. Load the Image Characterization Start Address 0 and 1 registers to the start addresses of the Image Characterization data buffers. See the Buffer Management section for more details.
3. Load the Metadata Start Address 0 and 1 registers to the start addresses of the Metadata buffers.
4. Load the Label Mask Start Address 0 register to the start address of the Label data buffer.
5. Load the Number of Horizontal Blocks register to the number of blocks in the horizontal direction. Typically this value is the horizontal resolution of the processed frame divided by the block size. Truncate any decimal portion.
6. Load the Number of Vertical Blocks register with the number of blocks in the vertical direction. Typically this value is the vertical resolution of the processed frame divided by the block size. Truncate any decimal portion.
7. Load the Number of Total Blocks with the number blocks that are going to be processed per frame. This value should be the Number of Horizontal blocks x the Number of Vertical blocks.
8. Load the Block Size register with the block size of the blocks that are being processed. A block is defined as a NxN 2-D grid of pixels where N is the block size. The Image Characterization core supports block sizes of 4, 8, 16, 32 and 64.
9. Set the register update enable bit of the control register (bit 1) to '1' and the core enable bit of the control register (bit 0) to '1' to fully enable the core. All of the preceding register values are written into the core on the next falling edge of the register values are written into the core on the next falling edge of the `fsync_in` signal.

Any of the core's registers can be updated while the core is running. It is recommended that the register update enable bit of the control register (bit 1) be set to '0' before any register are updated.

After all register changes have been written, the register update enable bit should be set to '1'. The new register values are written into the core on the next falling edge of the `fsync_in` signal.

Buffer Management

The Object Segmentation core makes use of five external memory buffers to handle the transfer of data. The buffer locations are defined by registers that specify their starting address location. Two of the buffers are used to input Image Characterization data. Two buffers are used for outputting the Object Segmentation Metadata. The last buffer is used to hold intermediate results from the first pass of processing until it is read in during the second pass of processing.

The Image Characterization Start Addr 0/1 registers hold the start addresses of the buffers that the Object Segmentation core uses to input Image Characterization data. Two buffers are used so that the Image Characterization core can write to one buffer while the Object Segmentation core can read from the other buffer. This arrangement ensures that the Object Segmentation core is always processing a valid Image Characterization data structure.

The Buffer Selection bit (3) of the Control register (See Table 2-3) determines how the Image Characterization Buffers are used. When the Buffer Selection bit = "0", the "buffer_ptr" signal is sampled on the falling edge of "fsync_in" signal. If `buffer_ptr` = 0, Image Characterization Buffer 0 is used. If `buffer_ptr` = 1, Image Characterization Buffer 1 is used. When the Buffer Selection bit = "1", Image Characterization Buffer 0 is used for the first frame. For the next frame, Buffer 1 is use. The core continues to switch between buffers on each successive frame.

The Metadata Start Addr 0/1 registers hold the start addresses of the buffers that the Object Segmentation core uses when writing the Object Segmentation Metadata. The Object Segmentation core uses the "Metadata Address Selection" register (control[2]) to specify which buffer is being actively written. The Object Segmentation core only writes to the active buffer. When the processor is ready to read the latest frame of Metadata, it first modifies the Metadata Address Selection register to swap the inactive buffer to be the active buffer and the active buffer to be the inactive buffer. After the next frame cycle begins, the newly inactive buffer contains the Metadata from the previous frame and is safe for the processor to access without danger of being overwritten. This mechanism is used because a processor might need to use multiple frame cycles to process the Metadata.

The Label Mask Start Addr 0 register holds the start address of the buffer that the Object Segmentation core uses to hold the intermediate Label data. This Label data is the output of the first pass of processing and the input of the second pass of processing. Only the Object Segmentation core uses this buffer, so there are no synchronization issues with which to be concerned.

Interrupts

The Object Segmentation core can flag four interrupts. The Status Error and Status Done interrupts report the processing operation of the core. The MM2S Error and S2MM Error interrupts report errors that occurred while transferring data across the AXI4 interface.

Status Error

On the falling edge of `fsync_in` (which signifies the start of the next frame), the Object Segmentation core checks to make sure that all of the Metadata from the previous frame has been written to memory. If any data has not been written to memory, then the core flags a status error. When using the General Purpose Processor interface, the error is indicated by the logic '1' state of the `status_error` signal. The `status_error` signal is reset to '0' on the next rising edge of `fsync_in`. When using the pCore interface, the `status_error` signal is used to drive bit 0 of the interrupt controller. It also sets bit 0 of the Status Error Register. The value in the Status Error Register can be reset by writing any value to the register.

Status Done

When the Object Segmentation core finishes writing all of the Metadata to memory, it flags that it has completed processing the current frame. When using the General Purpose Processor interface, the `status_done` signal is set to '1'. The `status_done` signal is reset to '0' on the falling edge of `fsync_in`, which denotes the start of the next frame. When using the pCore interface, the `status_done` signal is used to drive bit 1 of the interrupt controller. It also sets bit 0 of the Status Done Register. The value in the Status Done Register can be reset by writing any value to the register.

MM2S Error

The MM2S Error is asserted whenever an error condition is encountered within the MM2S portion of the AXI4 interface. When using the General Purpose Processor interface, the `mm2s_err` signal is set to '1' when an error is reported. When using the pCore interface, the `mm2s_err` signal is used to drive bit 3 of the interrupt controller. It also sets bit 2 of the Status Error Register.

S2MM Error

The S2MM Error is asserted whenever an error condition is encountered within the S2MM portion of the AXI4 interface. When using the General Purpose Processor interface, the `s2mm_err` signal is set to '1' when an error is reported. When using the pCore interface, the `s2mm_err` signal is used to drive bit 2 of the interrupt controller. It also sets bit 1 of the Status Error Register.

Evaluation Core Timeout

When generated with a Full System Hardware license, the core includes a timeout circuit that disables the core after a specific period of time. The timeout circuit can only be reset by reloading the FPGA bitstream. The timeout period for this core is set to approximately 8 hours for a 75 MHz clock. Using a faster or slower clock changes the timeout period proportionally. For example, using a 150 MHz clock results in a timeout period of approximately four hours.

Example Case

The key to understanding how to use the Object Segmentation core lies in learning how to properly configure the Feature Combinations and the Feature Selects, as described in the previous sections.

For this example, we will use the Object Segmentation core to detect objects that match either of the following feature descriptions:

1. Road signs that are Green with edges.
2. Road signs that are Yellow with edges.

For the purposes of this example we do not care to differentiate between the two feature descriptions, we just want to find objects of either type. To accomplish this, two Feature Combination units are needed; one to detect the "green signs" and one to detect the "yellow signs". Only one Feature Select is needed for this example because we are looking for either "green signs" or "yellow signs".

To properly setup the Object Segmentation core for this example, use the following configuration options:

- Color Select 1 is configured to detect the color "Green"

- Color Select 2 is configured to detect the color "Yellow"
- The video format is 4:2:0
- The frame resolution is 1280x720
- The Block Size is 8x8

Then to set these register values:

- Number of Horizontal Blocks = 1280/8 = 160
- Number of Vertical Blocks = 720/8 = 90
- Number of Total Blocks = 160x90 = 14400
- Block Size = 8

The next step is to configure the two Feature Combinations (FC1 and FC2). FC1 is looking for blocks that match the "green sign" feature description. To do this, it sets lower and upper block thresholds for "Color_Sel_1" (green) and for "Edge_Mean" (edges). FC2 is looking for blocks that match the "yellow sign" feature description. To do this it sets lower and upper block thresholds for "Color_Sel_2" (yellow) and for "Edge_Mean" (edges). For FC1 and FC2, block values that are not part of the feature description should be set to their widest threshold settings so that they do not limit the data comparisons.

For this example, the global statistics are not of interest so the corresponding global thresholds in FC1 and FC2 should also be set to their widest threshold settings. Two separate Feature Combination Threshold data structures are illustrated in Table 4-11, one for FC1 and another for FC2. The values in red highlight the threshold values that correspond to the feature descriptions for FC1 and FC2.

The FC1 Feature Combination Threshold data structure has these values:

1. Line 0x7 - The Edge_Mean lower block threshold was set to a value of 0x21
2. Line 0xC - the Color_Sel_1 lower block threshold was set to a value 0x0004
3. Line 0x1B - The Edge_Mean upper block threshold was set to a value of 0x53
4. Line 0x20 - The Color_Sel_1 upper block threshold was set to a value of 0x0010

The FC2 Feature Combination Threshold data structure has these values:

1. Line 0x7 - The Edge_Mean lower block threshold was set to a value of 0x32
2. Line 0xC - the Color_Sel_1 lower block threshold was set to a value 0x0005
3. Line 0x1B - The Edge_Mean upper block threshold was set to a value of 0x61
4. Line 0x20 - The Color_Sel_1 upper block threshold was set to a value of 0x0012

Table 4-11: **Feature Combination Threshold Data Structures for FC1 and FC2**

Line #	FC1	FC2
Lower Global Statistics		
0x0 – 0x5	0x00000000	0x00000000
Lower Block Statistics		
0x6	0x00000000	0x00000000
0x7	0x00002100	0x00003200
0x8-0xB	0x00000000	0x00000000
0xC	0x00000004	0x00050000
0xD – 0x13	0x00000000	0x00000000
Upper Global Statistics		
0x14 – 0x19	0xFFFFFFFF	0xFFFFFFFF
Upper Block Statistics		
0x1A	0xFFFFFFFF	0xFFFFFFFF
0x1B	0x00005300	0x00006100
0x1C – 0x1F	0xFFFFFFFF	0xFFFFFFFF
0x20	0xFFFF0010	0x0012FFFF
0x21 – 0x23	0xFFFFFFFF	0xFFFFFFFF
0x24 -0x27	0x00000000	0x00000000

The last step is to configure the Feature Select (FS1). FS1 should combine FC1 and FC2 such that any block that matches FC1 or FC2 is considered a member of FS1. FS1 can be described by the following Boolean logic equation:

FS1=FC2 or FC1

This equation is implemented as a 1-bit x 256-entry look-up table. Table 4-12 has an 8-bit address range. FC1 is mapped to the lsb of the address range, FC2 is mapped to the "lsb + 1" and FC8 is mapped to the "lsb + 7". In this example, Only FC1 and FC2 are instantiated so FC3 - FC8 are each set to a value of "0". The Feature Select data structure that implements the logic equation for FS1 in this example is shown in Table 4-12. The Feature Select data structure is 4-bits wide and 256 entry deep regardless of the number of Feature Selects that are instantiated. The entire data structure must be created and loaded to properly configure the Feature Selects.

.

Table 4-12: **Feature Select Data Structure for FS1 = FC2 or FC1**

Look-up Table Address FC8=msb, FC1=lsb	FS4	FS3	FS2	FS1
0x00 (00000000)	0	0	0	0
0x01 (00000001)	0	0	0	1
0x02 (00000010)	0	0	0	1
0x03 (00000011)	0	0	0	1
0x04 (00000100)	0	0	0	0
0x05 (00000101)	0	0	0	0
0x06 – 0xFF	0	0	0	0

Because FS4 - FS2 are not instantiated, the column under FS1 is the only portion of Table 4-12 that is of concern. FC1 and FC2 are the only bits of the address range that can change because FC8 - FC3 are not instantiated and therefore each is set to a value of "0". As a result, the effective address range is 0x0 (00000000) - 0x3 (00000011). Because FS1 is equal to the "FC1 or FC2", FS1 has a value of '1' any time FC1 is a '1' as well as anytime FC2 is a '1'.

Use Model

Figure 4-8 illustrates using the Object Segmentation core in a larger system. In this system, the Image Characterization core writes its calculated image statistics to an external memory buffer. This external memory buffer is then read by the Object Segmentation core. The core then analyzes the data with the user-defined object characteristics to find the specified objects. A list of objects found is written back to an external Metadata buffer for use in higher level analysis and processing. Such a system can be easily built using the building blocks provided by Xilinx (for example, AXI_VDMA, Timing Controller, On Screen Display (OSD).

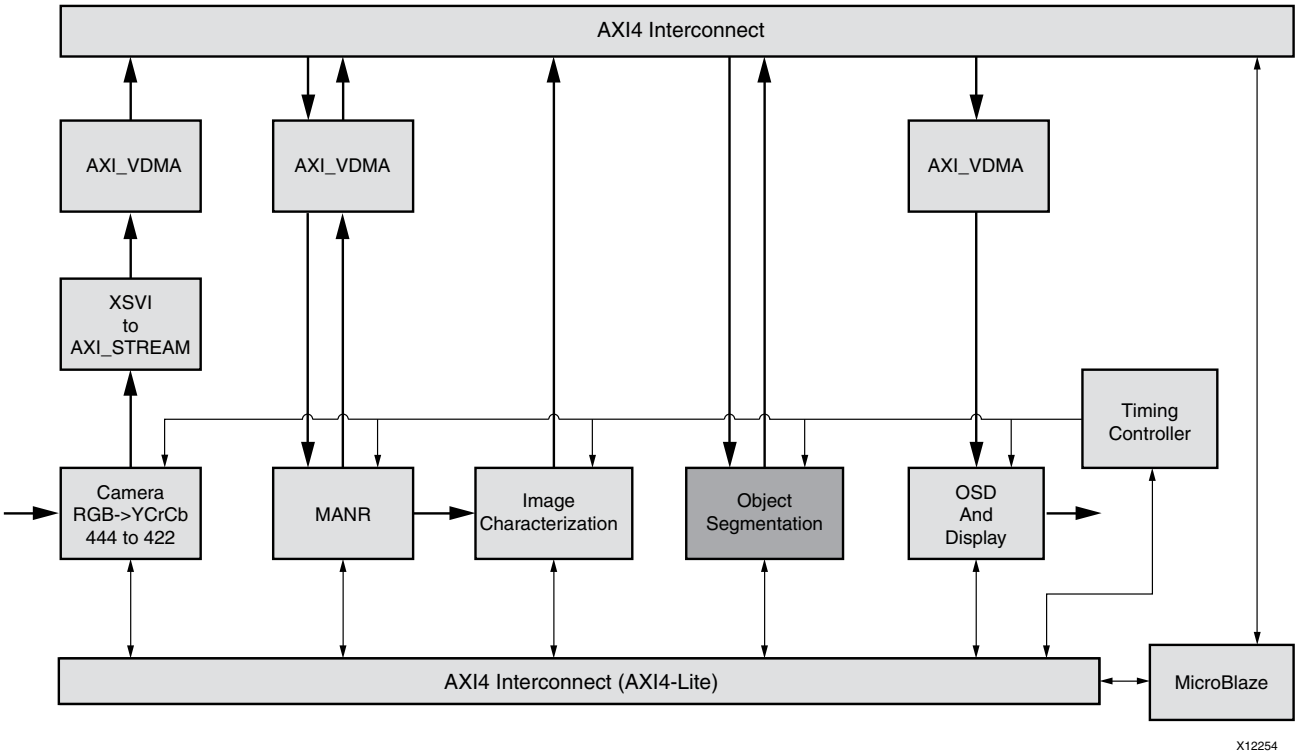

Figure 4-8: **Object Segmentation Example Use Model**

Clocking

The Object Segmentation core has one clock ("clk") that is used to clock the entire core. This includes the AXI interfaces and the core logic.

Resets

The Object Segmentation core has one reset ("sclr") that is used for the entire core. The reset is active high.

Protocol Description

For the pCore version of the Object Segmentation core, the register interface is compliant with the AXI4-Lite interface. The S2MM and MM2S interfaces are compliant with the AXI4 Memory Mapped interface.

XILINX®

Chapter 5

Constraining the Core

Required Constraints

There are no required constraints for the Object Segmentation core.

XILINX®

Chapter 6

Detailed Example Design

Directory and File Contents

Expected

The Expected directory contains the pre-generated expected/golden data used by the test bench to compare to the actual output data.

- Label_out.txt
- Meta_out.txt

Stimuli

The Stimuli directory contains the pre-generated input data used by the test bench to simulate the core (including register programming values).

- c_in.txt
- fs_in.txt
- ic_stats.txt
- label_in.txt
- reg_in.txt

Results

The Results directory is where the actual simulation output data file are written.

src

The src directory contains the .vhd and .xco files of the core. The .vhd file is a netlist generated using CORE Generator™ software. The .xco file can be used with the CORE Generator software to regenerate the netlist.

- v_objseg_v2_0_u0.vhd
- v_objseg_v2_0_u0.xco

tb_src

The tb_src directory contains the top-level test bench design. This directory also contains other packages used by the test bench.

- SimPack.sv
- tb_v_objseg_v2_0VHT.sv
- xi_config.sv
- isim_wave.wcfg - Waveform configuration file for iSim
- mti_wave.do - Waveform configuration for ModelSim
- run_isim.bat - Runscript for iSim in Windows OS
- run_isim.sh - Runscript for iSim in Linux OS
- run_mti.bat - Runscript for ModelSim in Windows OS
- run_mti.sh - Runscript for ModelSim in Linux OS

Demonstration Test Bench

The demonstration test bench is provided as a simple introductory package that enables core users to observe the core generated by the CORE Generator tool operating in a waveform simulator. The user is encouraged to observe core-specific aspects in the waveform, make simple modifications to the test conditions, and observe the changes in the waveform.

Simulation

Simulation using ModelSim for Linux:

- From the console, Type "source run_mti.sh".

Simulation using ModelSim for Windows:

- Double-click on "run_mti.bat" file.

Simulation using iSim for Linux:

- Double-click on "run_isim.bat" file.

Messages and Warnings

"Memory Collision Errors" have been observed when running this demonstration test bench. The issue has been investigated and it has been determined that these errors can be safely ignored. This error message can be suppressed in ModelSim when the global "SIM_COLLISION_CHECK" option is set to "NONE".

XILINX®

Appendix A

Verification, Compliance, and Interoperability

Simulation

A highly parameterizable test bench was used to test the Object Segmentation core. Testing included the following:

- Register accesses
- Processing of multiple frames of data
- Testing of various frame sizes and block sizes
- Varying instantiations of the core (Feature Selects = 1 – 4 and Feature Combinations = 1 – 8)
- Varying Feature Select Bank usage
- Varying Feature Combination Bank usage

Hardware Testing

The Object Segmentation core has been tested in a variety of hardware platforms at Xilinx to represent a variety of parameterizations, including the following:

- A test design was developed for the core that incorporated a MicroBlaze™ processor, AXI4 Interface and various other peripherals. The software for the test system included pre-generated input and output for the Object Segmentation core. Various tests could be supported by varying the configuration of the Object Segmentation core or by loading a different software executable. The MicroBlaze processor was responsible for:
  - Initializing the appropriate input and output buffers in external memory.
  - Initializing the Object Segmentation core.
  - Launching the test.
  - Comparing the output of the Object Segmentation core against the expected results.
  - Reporting the Pass/Fail status of the test and any errors that were found.

XILINX®

Appendix B

Debugging

Following are some debugging tips:

- Can the Version register be read properly? See Table 2-4 for register definitions.
- Verify that the `fsync_in` input is being properly driven.
- Verify that bits 0 and 1 of the core's Control register are both set to "1". Bit 0 is the Core Enable bit. Bit 1 is the Register Update Enable bit.
- Verify that the Feature Combination bank and Feature Select bank been initialized.
- Verify that the Feature Combination Active bank and Feature Select Active bank registers are specifying banks that have been properly initialized.
- Check the "Status Error" and "Status Done" registers. Ideally the Status Error register will read 0x00 and the Status Done register will read 0x01. Check Table 2-4 for definitions of each bit.
- Verify that the start addresses for the Image Characterization buffers, Metadata buffers and Label buffer have been properly initialized.
- Verify that the "Number of Horizontal Blocks", "Number of Vertical Blocks" and "Total Number of Blocks" registers have been properly initialized. SeeTable 2-4 for register definitions.
- Verify that the "Block Size" register has been properly initialized. See Table 2-4 for register definitions.
- See Solution Centers in Appendix E for information helpful to the debugging progress.

XILINX®

Appendix C

Application Software Development

pCore Driver Files

The Object Segmentation pCore includes a software driver written in the C programming language that the user can use to control the core. A high-level API provides application developers easy access to the features of the Xilinx® Object Segmentation core. A low-level API is also provided for developers to access the core directly through the system registers described in Register Space in Chapter 2.

Table C-1 lists the files included with the Object Segmentation pCore driver.

Table C-1: **Object Segmentation pCore Drivers**

File name	Description
xos.c	Provides the API access to all features of the Object Segmentation device driver.
xos.h	Provides the API access to all features of the Object Segmentation device driver.
xos_g.c	Contains a template for a configuration table of Object Segmentation core.
xos_hw.h	Contains identifiers and register-level driver functions (or macros) that can be used to access the Object Segmentation core.
xos_intr.c	Contains interrupt-related functions of the Object Segmentation device driver.
xos_sinit.c	Contains static initialization methods for the Object Segmentation device driver.

pCore API Functions

This section describes the functions included in the pcore Driver files generated for the Object Segmentation pCore. The software API is provide to allow easy access to the registers of the pCore as defined in Table 2-2 in the Register Space section. To utilize the API functions provided, the following header files must be included in the user's C code:

```
#include "xparameters.h"
#include "xos.h"
```

The hardware settings of your system, including the base address of your Object Segmentation core are defined in the xparameters.h file. The xos.h file provides the API access to all of the features of the Object Segmentation device driver.

More detailed documentation of the API functions can be found by opening the file index.html in the pCore directory os_v2_00_a/doc/html/api.

Functions in xos.c

- int XOS_CfgInitialize (XOS *InstancePtr, XOS_Config *CfgPtr, u32 EffectiveAddr)

  This function initializes an OS device.
- void XOS_SetImageStatAddr (XOS *InstancePtr, u32 Addr1, u32 Addr2)

  This function sets up input image statistics frame buffer addresses for an OS device.
- void XOS_GetImageStatAddr (XOS *InstancePtr, u32 *Addr1Ptr, u32 *Addr2Ptr)

  This function fetches the input image statistics frame buffer addresses for an OS device.
- void XOS_SetMetaDataAddr (XOS *InstancePtr, u32 Addr1, u32 Addr2)

  This function sets up output meta data frame buffer addresses for an OS device.
- void XOS_GetMetaDataAddr (XOS *InstancePtr, u32 *Addr1Ptr, u32 *Addr2Ptr)

  This function fetches output meta data frame buffer addresses for an OS device.
- void XOS_SetLabelMaskAddr (XOS *InstancePtr, u32 Addr1)

  This function sets up the output label mask data frame buffer addresses for an OS device.
- void XOS_GetLabelMaskAddr (XOS *InstancePtr, u32 *Addr1Ptr)

  This function fetches the output label mask data frame buffer addresses for an OS device.
- void XOS_FlipMetaDataAddr (XOS *InstancePtr)

  This function flips the meta data output buffer for an OS device.
- int XOS_FlipMetaDataAddrDone (XOS *InstancePtr)

  This function checks if the meta data output buffer flip operation is done for an OS device.
- u32 * XOS_GetReadyMetaDataAddr (XOS *InstancePtr)

  This function returns the active meta data output buffer address for an OS device.
- void XOS_SetBlock (XOS *InstancePtr, XOS_DimensionCfg *DimensionCfgPtr)

  This function sets up dimension related configuration information used by an OS device.
- void XOS_SetFeatureSelectWriteBankAddr (XOS *InstancePtr, u8 BankIndex)

  This function sets the feature select write bank address to be used by an OS device.
- void XOS_GetFeatureSelectWriteBankAddr (XOS *InstancePtr, u8 *BankIndex)

  This function fetches the feature select write bank address being used by an OS device.
- void XOS_SetFeatureSelectBank (XOS *InstancePtr, u8 *BankData)

  This function loads a feature select bank to be used by an OS device.
- void XOS_SetFeatureSelectActiveBankAddr (XOS *InstancePtr, u8 BankIndex)

  This function sets the feature select active bank address to be used by an OS device.
- void XOS_GetFeatureSelectActiveBankAddr (XOS *InstancePtr, u8 *BankIndex)

  This function fetches the feature select active bank address being used by an OS device.

- void XOS_LoadFeatureSelectBank (XOS *InstancePtr, u8 BankIndex, u8 *BankData)

  This function loads a feature select bank to be used by an OS device.

- void XOS_SetFeatureCombinationWriteBankAddr (XOS *InstancePtr, u8 BankIndex)

  This function sets the feature combination write bank address to be used by an OS device.

- void XOS_GetFeatureCombinationWriteBankAddr (XOS *InstancePtr, u8 *BankIndex)

  This function fetches the feature combination write bank address being used by an OS device.

- void XOS_SetFeatureCombinationBank (XOS *InstancePtr, u32 *BankData)

  This function loads a feature combination bank to be used by an OS device.

- void XOS_SetFeatureCombinationActiveBankAddr (XOS *InstancePtr, u8 BankIndex)

  This function sets the feature combination active bank address to be used by an OS device.

- void XOS_GetFeatureCombinationActiveBankAddr (XOS *InstancePtr, u8 *BankIndex)

  This function fetches the feature combination active bank address being used by an OS device.

- void XOS_LoadFeatureCombinationBank (XOS *InstancePtr, u8 BankIndex, u32 *BankData)

  This function loads a feature combination bank to be used by an OS device.

- void XOS_GetVersion (XOS *InstancePtr, u16 *Major, u16 *Minor, u16 *Revision)

  This function returns the version of an OS device.

Functions in xos_sinit.c

- XOS_Config * XOS_LookupConfig (u16 DeviceId)

  XOS_LookupConfig returns a reference to an XOS_Config structure based on the unique device id, DeviceId.

Functions in xos_intr.c

- void XOS_IntrHandler (void *InstancePtr)

  This function is the interrupt handler for the Object Segmentation driver.

- int XOS_SetCallBack (XOS *InstancePtr, u32 HandlerType, void *CallBackFunc, void *CallBackRef)

  This routine installs an asynchronous callback function for the given HandlerType:.

XILINX®

Appendix D

C Model Reference

The Xilinx® LogiCORE™ IP Object Segmentation v2.0 core has a bit accurate C model designed for system modeling.

Features

- Bit accurate with Object Segmentation core (v_objseg_v2_0)
- Statically linked library (.lib, .o, .obj - Windows)
- Dynamically linked library (.so – Linux)
- Available for 32-bit Windows, 64-bit Windows, 32-bit Linux, and 64-bit Linux platforms
- Supports all features of the Object Segmentation core that affect numerical results
- Designed for rapid integration into a larger system model
- Example C code is provided to show how to use the function
- Example application C code wrapper files support 8-bit image characterization input only

Overview

The LogiCORE IP Object Segmentation core has a bit accurate C model for 32-bit Windows, 64-bit Windows, 32-bit Linux, and 64-bit Linux platforms. The model has an interface consisting of a set of C functions, which reside in a statically link library (shared library). Full details of the interface are provided in Interface. An example piece of C code is provided to show how to call the model.

The model is bit accurate because it produces exactly the same output data as the core on a frame-by-frame basis. However, the model is not cycle accurate because it does not model the core's latency or its interface signals.

The latest version of the model is available for download on the LogiCORE IP Object Segmentation Web page at:
http://www.xilinx.com/products/ipcenter/EF-DI-VID-OBJ-SEG.htm

Additional Core Resources

For detailed information and updates about the Object Segmentation v2.0 core, see the documents listed on the core product page at:

http://www.xilinx.com/products/ipcenter/EF-DI-VID-OBJ-SEG.htm

Technical Support

For technical support, go to www.xilinx.com/support. Questions are routed to a team with expertise using the Object Segmentation core.

Xilinx provides technical support for use of this product as described in this product guide.

Xilinx cannot guarantee functionality or support of this product for designs that do not follow these guidelines.

Feedback

Xilinx welcomes comments and suggestions about the Object Segmentation v2.0 core and the accompanying documentation.

Object Segmentation Bit Accurate C Model and IP Core

For comments or suggestions about the Object Segmentation core and bit accurate C model, submit a WebCase from:
http://www.xilinx.com/support/clearexpress/websupport.htm

Be sure to include the following information:

- Product name
- Core version number
- Explanation of your comments

Document

For comments or suggestions about the documentation for the Object Segmentation core and bit accurate C model, submit a WebCase from:
http://www.xilinx.com/support/clearexpress/websupport.htm

Be sure to include the following information:

- Document title
- Document number
- Page number(s) to which your comments refer
- Explanation of your comments

User Instructions

Unpacking and Model Contents

Unzip the `v_objseg_v2_0_bitacc_model.zip` file, containing the bit accurate models for the Object Segmentation IP Core. This creates the directory structure and files in Table D-1.

Table D-1: **Directory Structure and Files of the Object Segmentation Bit Accurate C Model**

File Name	Contents
README.txt	Release notes
pg018_v_obj_seg.pdf	LogiCORE IP Object Segmentation Product Guide
v_objseg_v2_0_bitacc_cmodel.h	Model header file
rgb_utils.h	Header file declaring the RGB image/video container type and support functions
video_utils.h	Header file declaring the generalized image/video container type, I/O and support functions
yuv_utils.h	Header file declaring the YUV image/video container type and support functions
image_char_stats_utils.h	Header file declaring the Image Characterization Statistics container type and support functions
run_bitacc_cmodel.c	Example code calling the C model
ic_stats_512.txt	Example Image Characterization statistics files
objseg_config_512.cfg	Example configuration file
fc1.cfg	Example Feature Combination configuration file
fs1.cfg	Example Feature Select configuration file
/lin	Precompiled bit accurate American National Standards Institute (ANSI) C reference model for simulation on 32-bit Linux platforms
libIp_v_objseg_v2_0_bitacc_cmodel.so	Model shared object library
libstlport.so.5.1	STL library, referenced by libIp_v_objseg_v2_0_bitacc_cmodel.so
/lin64	Precompiled bit accurate ANSI C reference model for simulation on 64-bit Linux platforms
libIp_v_objseg_v2_0_bitacc_cmodel.so	Model shared object library
libstlport.so.5.1	STL library, referenced by libIp_v_ic_v2_0_bitacc_cmodel.so
/win32	Precompiled bit accurate ANSI C reference model for simulation on 32-bit Windows platforms
libIp_v_objseg_v2_0_bitacc_cmodel.lib	Precompiled library file for Win32 compilation
/win64	Precompiled bit accurate ANSI C reference model for simulation on 64-bit Windows platforms
libIp_v_objseg_v2_0_bitacc_cmodel.lib	Precompiled library file for Win64 compilation

Installation

For Linux, make sure these files are in a directory that is in your $LD_LIBRARY_PATH environment variable:

- libIp_v_objseg_v2_0_bitacc_cmodel.so
- libstlport.so.5.1

Software Requirements

The Object Segmentation C models were compiled and tested with the software listed in Table D-2.

Table D-2: **Compilation Tools for the Bit Accurate C Models**

Platform	C Compiler
32-bit Linux	GCC 4.1.1
64-bit Linux	GCC 4.1.1
32-bit Windows	Microsoft Visual Studio 2008
64-bit Windows	Microsoft Visual Studio 2008

Interface

The bit accurate C model is accessed through a set of functions and data structures, declared in the header file `v_objseg_v2_0_bitacc_cmodel.h`

Before using the model, the structures holding the inputs, generics and output of the Image Characterization instance must be defined:

```
struct  xilinx_ip_v_objseg_v2_0_generics objseg_generics;
struct  xilinx_ip_v_objseg_v2_0_inputs   objseg_inputs;
struct  xilinx_ip_v_objseg_v2_0_outputs  objseg_outputs
```

The declaration of these structures are in the `v_objseg_v2_0_bitacc_cmodel.h` file.

Calling `xilinx_ip_v_objseg_v2_0_get_default_generics` (and `objseg_generics`) initializes the generics structure with the default values for each element of the structure.

The generics defaults are:

```
frames = 1                    // Number of frames
num_feature_combinations = 8; // Number of Feature Combination units
num_feature_selects = 4;      // Number of Feature Selection units
num_h_blocks = 160;           // Number of Horizontal blocks in IC Stats
num_v_blocks = 90;            // Number of Vertical blocks in IC Stats
num_total_blocks = 14400;     // Number of Total blocks in IC Stats
block_size = 8;               // Block Size (4, 8, 16, 32 or 64)

For fc[1] - fc[8]
   // Global Stats Lower Thresholds
   fc[i].lower.global_y_mean = 0;
   fc[i].lower.global_u_mean = 0;
   fc[i].lower.global_v_mean = 0;
   fc[i].lower.global_lf_mean = 0;
   fc[i].lower.global_hf_mean = 0;
   fc[i].lower.global_edge_mean = 0;
   fc[i].lower.global_mot_mean = 0;
   fc[i].lower.global_sat_mean = 0;
   fc[i].lower.global_y_var = 0;
   fc[i].lower.global_u_var = 0;
   fc[i].lower.global_v_var = 0;
   fc[i].lower.global_lf_var = 0;
   fc[i].lower.global_hf_var = 0;
   fc[i].lower.global_edge_var = 0;
   fc[i].lower.global_mot_var = 0;
   fc[i].lower.global_sat_var = 0;
   // Global Stats Upper Thresholds
   fc[i].upper.global_y_mean = 255;
   fc[i].upper.global_u_mean = 255;
   fc[i].upper.global_v_mean = 255;
   fc[i].upper.global_lf_mean = 255;
   fc[i].upper.global_hf_mean = 255;
   fc[i].upper.global_edge_mean = 255;
   fc[i].upper.global_mot_mean = 255;
   fc[i].upper.global_sat_mean = 255;
   fc[i].upper.global_y_var = 65535;
   fc[i].upper.global_u_var = 65535;
   fc[i].upper.global_v_var = 65535;
   fc[i].upper.global_lf_var = 65535;
   fc[i].upper.global_hf_var = 65535;
   fc[i].upper.global_edge_var = 65535;
   fc[i].upper.global_mot_var = 65535;
   fc[i].upper.global_sat_var = 65535;
   fc[i].upper.global_sat_var = 65535;
   // Block Stats Lower Thresholds
   fc[i].lower.block_y_mean = 0;
   fc[i].lower.block_u_mean = 0;
   fc[i].lower.block_v_mean = 0;
   fc[i].lower.block_lf_mean = 0;
   fc[i].lower.block_hf_mean = 0;
   fc[i].lower.block_edge_mean = 0;
   fc[i].lower.block_mot_mean = 0;
   fc[i].lower.block_sat_mean = 0;
   fc[i].lower.block_y_var = 0;
   fc[i].lower.block_u_var = 0;
   fc[i].lower.block_v_var = 0;
   fc[i].lower.block_lf_var = 0;
```

```
    fc[i].lower.block_hf_var = 0;
    fc[i].lower.block_edge_var = 0;
    fc[i].lower.block_mot_var = 0;
    fc[i].lower.block_sat_var = 0;
    fc[i].lower.block_col_sel1 = 0;
    fc[i].lower.block_col_sel2 = 0;
    fc[i].lower.block_col_sel3 = 0;
    fc[i].lower.block_col_sel4 = 0;
    fc[i].lower.block_col_sel5 = 0;
    fc[i].lower.block_col_sel6 = 0;
    fc[i].lower.block_col_sel7 = 0;
    fc[i].lower.block_col_sel8 = 0;
    // Block Stats Upper Thresholds
    fc[i].upper.block_y_mean = 255;
    fc[i].upper.block_u_mean = 255;
    fc[i].upper.block_v_mean = 255;
    fc[i].upper.block_lf_mean = 255;
    fc[i].upper.block_hf_mean = 255;
    fc[i].upper.block_edge_mean = 255;
    fc[i].upper.block_mot_mean = 255;
    fc[i].upper.block_sat_mean = 255;
    fc[i].upper.block_y_var = 65535;
    fc[i].upper.block_u_var = 65535;
    fc[i].upper.block_v_var = 65535;
    fc[i].upper.block_lf_var = 65535;
    fc[i].upper.block_hf_var = 65535;
    fc[i].upper.block_edge_var = 65535;
    fc[i].upper.block_mot_var = 65535;
    fc[i].upper.block_sat_var = 65535;
    fc[i].upper.block_sat_var = 65535;
    fc[i].upper.block_col_sel1 = 4095;
    fc[i].upper.block_col_sel2 = 4095;
    fc[i].upper.block_col_sel3 = 4095;
    fc[i].upper.block_col_sel4 = 4095;
    fc[i].upper.block_col_sel5 = 4095;
    fc[i].upper.block_col_sel6 = 4095;
    fc[i].upper.block_col_sel7 = 4095;
    fc[i].upper.block_col_sel8 = 4095;

fs[1] = fc[1];
fs[2] = fc[2];
fs[3] = fc[3];
fs[4] = fc[4];
```

The structure `objseg_inputs` defines the values of the input image characterization statistics. For a description of the input structure, see Image Characterization Statistics Input Structure.

The structure `objseg_outputs` defines the values of the output object segmentation metadata. For a description of the output structure, see Object Segmentation Metadata Output Structure.

Note: The `objseg_input` and `objseg_output` variables are not initialized, as the initialization depends on the actual test to be simulated. The next chapters describe the initialization of the `objseg_input` and `objseg_output` structures.

After the inputs are defined, the model can be simulated by calling the function:

```
int xilinx_ip_v_objseg_v2_0_bitacc_simulate(
      struct xilinx_ip_v_objseg_v2_0_generics* generics,
      struct xilinx_ip_v_objseg_v2_0_inputs*   inputs,
      struct xilinx_ip_v_objseg_v2_0_outputs*  outputs).
```

Results are provided in the outputs structure. After the outputs are evaluated and saved, dynamically allocated memory for input and output video structures must be released by calling the function:

```
void xilinx_ip_v_objseg_v2_0_destroy(
       struct xilinx_ip_v_objseg_v2_0_inputs *input,
       struct xilinx_ip_v_objseg_v2_0_outputs *output)
```

Successful execution of all provided functions, except for the destroy function, return a value of 0. Otherwise, a non-zero error code indicates that problems occurred during function calls.

Image Characterization Statistics Input Structure

The Object Segmentation reference model inputs a set of image characterization statistics for each frame that is processed. The input statistics are provided by `image_char_stats_struct`, which is defined in `image_char_stats_utils.h`.

```
struct image_char_stats_struct
{
  int   frames;                              // Number of frames
  int   num_blocks_wide;                     // Number of blocks wide
  int   num_blocks_high;                     // Number of blocks high
  int*  frame_index;                        // Frame Index for each frame
  struct global_stats_struct** global;       // Global stats
  struct block_stats_struct*** block;        // Block stats
  int** y_histogram;                         // Y Histogram
  int** u_histogram;                         // U Histogram
  int** v_histogram;                         // V Histogram
  int** hue_histogram;                       // Hue Histogram
};

struct global_stats_struct
{
  uint8   y_mean;       // Y mean
  uint8   u_mean;       // U mean
  uint8   v_mean;       // V mean
  uint8   m_mean;       // Motion mean
  uint8   e_mean;       // Edge mean
  uint8   lp_mean;      // Low Frequency mean
  uint8   hp_mean;      // High Frequency mean
  uint8   sat_mean;     // Saturation mean
  uint16  y_var;        // Y variance
  uint16  u_var;        // U variance
  uint16  v_var;        // V variance
  uint16  m_var;        // Motion variance
  uint16  e_var;        // Edge variance
  uint16  lp_var;       // Low Frequency variance
  uint16  hp_var;       // High Frequency variance
  uint16  sat_var;      // Saturation variance
};

struct block_stats_struct
```

```
{
  uint8   y_mean;       // Y mean
  uint8   u_mean;       // U mean
  uint8   v_mean;       // V mean
  uint8   m_mean;       // Motion mean
  uint8   e_mean;       // Edge mean
  uint8   lp_mean;      // Low Frequency mean
  uint8   hp_mean;      // High Frequency mean
  uint8   sat_mean;     // Saturation mean
  uint16  y_var;        // Y variance
  uint16  u_var;        // U variance
  uint16  v_var;        // V variance
  uint16  m_var;        // Motion variance
  uint16  e_var;        // Edge variance
  uint16  lp_var;       // Low Frequency variance
  uint16  hp_var;       // High Frequency variance
  uint16  sat_var;      // Saturation variance
  uint16  color_sel[8]; // Color Select (x8)
};
```

The `image_char_stats_struct` holds the results of multiple processed frames. The number of frames in the structure is specified in the frames element of the structure. The `num_blocks_wide` and `num_blocks_high` elements denote the width and height of the 2-D grid of block statistics that are stored for each frame of statistics. The `frame_index` is an array with one value per frame; it holds the index values of each frame. The global element is an array of `global_stats_structs` with one structure per frame. It holds the global statistics as defined in `global_stats_struct`. The block element is a 3-D grid of `block_stats_structs`. The first dimension is based on the number of frames, the second dimension is based on `num_blocks_high`, and the third dimension is based on `num_blocks_wide`. Each point of the grid is an instance of `block_stats_struct`, which holds the block statistics for each block of each frame. The `y_histogram`, `u_histogram`, `v_histogram` and `hue_histogram` are 2-D arrays. The first dimension is based on the frames and the second dimension is an array of 256 elements. They each hold a corresponding 256 bin histogram for each frame.

Working With Image_char_stats_struct Containers

The `image_char_stats_utils.h` file defines functions to simplify the use of image characterization statistics structures.

```
int alloc_ic_stats_buff(struct image_char_stats_struct* ic_stats);
void free_ic_stats_buff(struct image_char_stats_struct* ic_stats);
int write_ic_stats(FILE *output_fid, struct image_char_stats_struct
*stats);
int read_ic_stats(FILE *input_fid, struct image_char_stats_struct*
stats);
```

The `alloc_ic_stats_buff` function can be used to dynamically create an `image_char_stats_struct`. The frame, `num_blocks_wide` and `num_blocks_high`, elements of the structure must be specified before calling this routine. The `free_ic_stats_buff` function can be used to destroy `image_char_stats_struct`.

The `write_ic_stats` function writes `image_char_stats_struct` to a text file. The `read_ic_stats` function reads `image_char_stats_struct` from a text file. Each frame of statistics in the text file is stored in this order:

1. Structure header
2. Global statistics
3. Histograms (Y, U, V and Hue)
4. Block statistics (each column of each row)

The data structure matches the format of the output of the Image Characterization core. See *PG015 - LogiCORE IP Image Characterization v2.0 Product Guide* for more information.

Object Segmentation Metadata Output Structure

The Object Segmentation reference model outputs a set of object metadata for each frame that is processed. The object metadata are provided by `obj_seg_metadata_struct`, which is defined in `obj_seg_metadata_utils.h`.

```
struct obj_seg_metadata_struct
{
  int   frames;               // Number of frames
  int*  frame_index;          // Frame Index for each frame
  int*  total_num_objects;    // Total Number of Objects in the Frame
  int**  fs_num_objects;      // Number of Objects for FS1-4 in the Frame
  int*  y_mean;               // Global Y Mean for the Frame
  int*  u_mean;               // Global U Mean for the Frame
  int*  v_mean;               // Global V Mean for the Frame
  int*  lf_mean;              // Global Low Frequency Mean for the Frame
  int*  hf_mean;              // Global High Frequency Mean for the Frame
  int*  edge_mean;            // Global Edge Content Mean for the Frame
  int*  mot_mean;             // Global Motion Mean for the Frame
  int*  sat_mean;             // Global Saturation Mean for the Frame
  int*  y_var;                // Global Y Variance for the Frame
  int*  u_var;                // Global U Variance for the Frame
  int*  v_var;                // Global V Variance for the Frame
  int*  lf_var;               // Global Low Frequency Variance for the Frame
  int*  hf_var;               // Global High Frequency Variance for the Frame
  int*  edge_var;             // Global Edge Content Variance for the Frame
  int*  mot_var;              // Global Motion Variance for the Frame
  int*  sat_var;              // Global Saturation Variance for the Frame

  struct object_metadata_struct*** fs; // FS1-4 Object Data (32 objects each FS)
};

struct object_metadata_struct
{
  uint16   object_number;      // Object Number
  uint16   FS_number;          // Feature Select Number
  uint16   xstart;             // X Start coordinate of the bounding box
  uint16   xstop;              // X Stop coordinate of the bounding box
  uint16   ystart;             // Y Start coordinate of the bounding box
  uint16   ystop;              // Y Stop coordinate of the bounding box
  uint16   xcentroid;          // X Centroid of the object
  uint16   ycentroid;          // Y Centroid of the bounding box
  int      object_density;     // Number of object blocks inside the bounding box
  int      object_identifier;  // Unique object identifier
};
```

`Obj_seg_meta_data_struct` can hold the results of multiple processed frames. The number of frames in the structure is specified in the frames element of the structure. The `frame_index` is an array with one value per frame; it holds the index values of each frame. The `total_num_objects` element is an array with one value per frame. Each value holds the total number of objects found in all of the feature selects (fs1-4) for the corresponding frame. The element `fs_num_objects` is a 2-D array in which the first dimension is based on the number of frames and the second dimension is based on the number of Feature Selects. They hold the number of objects found for each Feature Select for each frame. The "*_mean" and "*_var" elements are arrays of global statistics; each array contains one value per frame. The fs element is a 3-D array of object data. The first dimension is based on the number of frames, the second dimension is based on the number of Feature Selects (fs1-fs4), and the third dimension is based on the maximum number of objects (32) for a feature select. The leaf element is a pointer to `object_metadata_struct`.

`Object_metadata_struct` holds the metadata associated with an object that was found by the Object Segmentation core, and consists of:

- The object number (1 - 32)
- The number of the Feature Select associated with it
- The coordinates of the objects bounding box
- The number of blocks inside the box that belongs to the object
- A unique object identifier

Working With Obj_seg_metadata_struct Containers

The `obj_seg_metadata_utils.h` file defines functions to simplify the use of Object Segmentation Metadata structures.

```
int alloc_obj_seg_metadata_buff(struct obj_seg_metadata_struct* obj_seg_meta);
void free_obj_seg_metadata_buff(struct obj_seg_metadata_struct* obj_seg_meta);

int write_obj_seg_metadata(FILE *output_fid, struct obj_seg_metadata_struct meta);
int read_obj_seg_metadata(FILE *output_fid, struct obj_seg_metadata_struct* meta);
```

The `alloc_obj_seg_metadata_buff` function can be used to dynamically create `obj_seg_metadata_struct`. The frame element of the structure must be specified before calling this routine. The `free_ic_stats_buff` function can be used to destroy `obj_seg_metadata_struct`.

The `write_obj_seg_metadata` function writes `obj_seg_metadata_struct` to a text file. The `read_obj_seg_metadata` function reads `obj_seg_metadata_struct` from a text file. Each frame of metadata in the text file is stored in this order:

1. Structure header
2. FS1 object 1 - FS1 0bject 32
3. FS2 object 1 - FS2 0bject 32
4. FS3 object 1 - FS3 0bject 32
5. FS4 object 1 - FS4 0bject 32

The data structure matches the format of the output of the Object Segmentation core. C Model Example Code

An example C file, `run_bitacc_cmodel.c`, is provided and has these characteristics:

- Contains an example of how to write an application that makes a function call to the Object Segmentation C model core function.
- Contains an example of how to populate the video structures at the input and output, including allocation of memory to these structures.
- Reads the Image Characterization statistics from an input file.
- Writes the Object Segmentation metadata to an output file.

After following the compilation instructions in this chapter, you should run the example executable. If invoked with insufficient parameters, this help message is generated:

```
Usage: run_bitacc_cmodel in_file config_file out_file

 in_file        :  Path/name of the input file.
 config_file    :  Path/name of the configuration file.
 out_file       :  Path/name of the output IC Stats file.
```

Config Files

The Object Segmentation model must be initialized using configuration files.

Object Segmentation Config File

During successful execution, the specified config file is parsed by the `run_bitacc_cmodel` example. This is the top-level config file that is specified in the command line arguments. In this file, you must specify:

- Number of frames to process
- Number of feature combinations and feature selects
- Number of horizontal and vertical blocks
- Feature combination config file for each feature combination
- Feature select config file the feature selects

The following example config file provides more information on the formatting of this file.

```
num_frames 2                 # Number of Frames
num_feature_combinations 8   # Number of Feature Combinations 1-8
num_feature_selects 4        # Number of Feature Selects 1-4
num_h_blocks 64              # Number of Horizontal Blocks in IC input
num_v_blocks 64              # Number of Vertical Blocks in IC input
fc1 fc1.cfg                  # Feature Combination config file for FC1
fc2 fc1.cfg                  # Feature Combination config file for FC2
fc3 fc1.cfg                  # Feature Combination config file for FC3
fc4 fc1.cfg                  # Feature Combination config file for FC4
fc5 fc1.cfg                  # Feature Combination config file for FC5
fc6 fc1.cfg                  # Feature Combination config file for FC6
fc7 fc1.cfg                  # Feature Combination config file for FC7
fc8 fc1.cfg                  # Feature Combination config file for FC8
fs fs1.cfg                   # Feature Select config file for FS1-FS4
```

Feature Combination Config File

The feature combination config file contains the data necessary to properly configure one feature combination in the object segmentation model. A separate feature combination config file should be loaded for each feature combination that is used. The config file specifies a set of lower and upper thresholds that are used when testing the global and block statistics in the image characterization input.

The following example config file provides more information on the formatting of this file.

```
# Block Stats
y_mean          100   255           # Block Y Mean
y_var             0 65535           # Block Y Variance
u_mean            0   255           # Block U Mean
u_var             0 65535           # Block U Variance
v_mean            0   255           # Block V Mean
v_var             0 65535           # Block V Variance
LP_y_mean         0   255           # Block Low Frequency Mean
LP_y_var          0 65535           # Block Low Frequency Variance
HP_y_mean         0   255           # Block High Frequency Mean
HP_y_var          0 65535           # Block High Frequency Variance
edge_y_mean       0   255           # Block Edge Content Mean
edge_y_var        0 65535           # Block Edge Content Variance
motion_y_mean     0   255           # Block Motion Mean
motion_y_var      0 65535           # Block Motion Variance
img_sat_mean      0   255           # Block Saturation Mean
img_sat_var       0 65535           # Block Saturation Variance
color_select1     0   255           # Block Color Select 1
color_select2     0   255           # Block Color Select 2
color_select3     0   255           # Block Color Select 3
color_select4     0   255           # Block Color Select 4
color_select5     0   255           # Block Color Select 5
color_select6     0   255           # Block Color Select 6
color_select7     0   255           # Block Color Select 7
color_select8     0   255           # Block Color Select 8

# Global Stats
global_y_mean           3   250     # Global Y Mean
global_y_var            0 65535     # Global Y Variance
global_u_mean           5   225     # Global U Mean
global_u_var            0 65535     # Global U Variance
global_v_mean           0   255     # Global V Mean
global_v_var            0 65535     # Global V Variance
global_LP_y_mean        0   255     # Global Low Frequency Mean
global_LP_y_var         0 65535     # Global Low Frequency Variance
global_HP_y_mean        0   255     # Global High Frequency Mean
global_HP_y_var         0 65535     # Global High Frequency Variance
global_edge_y_mean      0   255     # Global Edge Content Mean
global_edge_y_var       0 65535     # Global Edge Content Variance
global_motion_y_mean    0   255     # Global Motion Mean
global_motion_y_var     0 65535     # Global Motion Variance
global_img_sat_mean     0   255     # Global Saturation Mean
global_img_sat_var      0 65535     # Global Saturation Variance
```

Feature Select Config File

The feature select config file initializes all of the feature selects that are used. Each feature select is a Boolean equation that uses the feature combinations as terms in the equation. It is implemented as a 1-bit x 256-entry look-up table that uses the feature combinations as addresses into the table. FC1 is mapped to the LSB, FC2 is mapped to the LSB+1, and FC8 is mapped to the MSB.

The feature selects are initialized by loading 256 values that each consist of 4-bits. FS1 corresponds to bit 0, FS2 corresponds to bit 1, FS3 corresponds to bit 2 and FS4 corresponds to bit 3.

The feature select config file consists of 256 entries that are used to initialize the 4-bit x 256 entry feature select look-up table. The first value in the file corresponds to address 0x0 (00000000) in the table. Each subsequent value corresponds to the next address location in the table, incrementing all the way up to the top address of 0xFF (11111111). The following C code illustrates a simple way to calculate the data for a feature select config file.

```
// Initialize the 4 Feature Selection Banks
for(i=0;i<256;i++) {
   fc1 = i & 0x01;
   fc2 = (i >> 1 ) & 0x01;
   fc3 = (i >> 2 ) & 0x01;
   fc4 = (i >> 3 ) & 0x01;
   fc5 = (i >> 4 ) & 0x01;
   fc6 = (i >> 5 ) & 0x01;
   fc7 = (i >> 6 ) & 0x01;
   fc8 = (i >> 7 ) & 0x01;

   fs1[i] = fc1;
   fs2[i] = (fc2 && fc5) || fc7;
   fs3[i] = fc3 || fc4 || fc6;
   fs4[i] = fc1 && fc7 && fc8;

   fs[i] = fs4[i]<<3 | fs3[i]<<2 | fs2[i]<<1 | fs1[i];
}
```

Initializing the Image Characterization Input Data Structure

In the example code wrapper, data is assigned to `image_char_stats_struct` by reading from a file containing image characterization data. The `image_char_stats_utils.h` file provided with the bit accurate C model contains functions to facilitate this file I/O. The `run_bitacc_cmodel` example code uses this function to read from the delivered image characterization data file.

Image Characterization Data Files

The `image_char_stats_utils.h` file declares functions that help access image characterization files. The following functions operate on arguments of type `image_char_stats_struct`, which is defined in `image_char_stats_utils.h`.

```
int alloc_ic_stats_buff(struct image_char_stats_struct* ic_stats);
void free_ic_stats_buff(struct image_char_stats_struct* ic_stats);

int write_ic_stats(FILE *output_fid, struct image_char_stats_struct* stats);
int read_ic_stats(FILE *input_fid, struct image_char_stats_struct* stats);
```

Use the `alloc_ic_stats_buff` and `free_ic_stats_buff` commands to dynamically manage the memory associated with an image characterization data buffer. Use the `write_ic_stats` and `read_ic_stats` functions for file I/O operations.

Initializing the Object Segmentation Metadata Output Data Structure

In the example code wrapper, the object segmentation model writes the results to `obj_seg_metadata_struct`. These results are then written to a file. The `obj_seg_metadata_utils.h` file provided with the bit accurate C model contains functions to facilitate this file I/O.

Object Segmentation Metadata Files

The `obj_seg_metadata_utils.h` file declares functions that help access object segmentation metadata files. The following functions operate on arguments of type `obj_seg_metadata_struct`, which is defined in `obj_seg_metadata_utils.h`.

```
int alloc_obj_seg_metadata_buff(struct obj_seg_metadata_struct* obj_seg_meta);
void free_obj_seg_metadata_buff(struct obj_seg_metadata_struct* obj_seg_meta);

int write_obj_seg_metadata(FILE *output_fid,
                           struct obj_seg_metadata_struct meta);
int read_obj_seg_metadata(FILE *output_fid,
                          struct obj_seg_metadata_struct* meta);
```

Use the `alloc_obj_seg_metadata_buff` and `free_obj_seg_metadata_buff` commands to dynamically manage the memory associated with an object segmentation metadata buffer. Use the `write_obj_seg_metadata` and `read_obj_seg_metadata` functions for file I/O operations.

C-Model Example I/O Files

Input Files

- **<in_filename>** (for example, `ic_stats_in.txt`)
  - Image Characterization statistics
- **<config_file>** (for example, `objseg_config_512.cfg`)
  - Object Segmentation configuration

Output Files

- **<out_filename>** (for example, `objseg_out.txt`)
  - Object Segmentation metadata

Compiling the Object Segmentation v2.0 C Model With Example Wrapper

Linux (32-bit and 64-bit)

To compile the example code, perform these steps:

1. Set your $LD_LIBRARY_PATH environment variable to include the root directory where you unzipped the model zip file, as shown in this example:

   ```
   setenv LD_LIBRARY_PATH <unzipped_c_model_dir>:${LD_LIBRARY_PATH}
   ```

2. Copy these files from the /lin (for 32-bit) or from the /lin64 (for 64-bit) directory to the root directory:
   - libstlport.so.5.1
   - libIp_v_objseg_v2_0_bitacc_cmodel.so
3. In the root directory, compile using the GNU C Compiler with this command:

```
gcc -x c++ run_bitacc_cmodel.c -o run_bitacc_cmodel -L.
-lIp_v_objseg_v2_0_bitacc_cmodel -Wl,-rpath,.
```

4. This results in the creation of the executable `run_bitacc_cmodel`, which can be run using:

```
./run_bitacc_cmodel ic_stats_512.txt objseg_config_512.cfg
objseg_out.txt
```

Windows (32-bit and 64-bit)

Precompiled library `v_scaler_v4_0_bitacc_cmodel.lib`, and top-level demonstration code `run_bitacc_cmodel.c` must be compiled with an ANSI C compliant compiler under Windows. Here, an example is provided using Microsoft Visual Studio.

In Visual Studio create a new, empty Win32 Console Application project. As existing items, add:

- **libIp_v_objseg_v2_0_bitacc_cmodel.lib** to the "Resource Files" folder of the project
- **run_bitacc_cmodel.c** to the "Source Files" folder of the project
- **v_objseg_v2_0_bitacc_cmodel.h** to "Header Files" folder of the project
- **yuv_utils.h** to the "Header Files" folder of the project
- **rgb_utils.h** to the "Header Files" folder of the project
- **video_utils.h** to the "Header Files" folder of the project
- **image_char_stats_utils.h** to the "Header Files" folder of the project
- **obj_seg_metadata_utils.**h to the "Header Files" folder of the project

After the project is created and populated, it must be compiled and linked (built) to create a Win32 or Win64 executable. To perform the build step, choose **Build Solution** from the Build menu. An executable matching the project name is created in the Debug or Release subdirectories under the project location based on whether "Debug" or "Release" is selected in the "Configuration Manager" in the Build menu.

Running the Delivered Executables

Included in the zip file are precompiled executable files to use with Win32, Win64, Linux32 and Linux64 platforms.

Linux (32-bit and 64-bit)

1. Set your $LD_LIBRARY_PATH environment variable to include the root directory where you unzipped the model zip file, as shown in this example:

```
setenv LD_LIBRARY_PATH <unzipped_c_model_dir>:${LD_LIBRARY_PATH}
```

2. Copy these files from the /lin or /lin64 directory to the root directory:
   - libstlport.so.5.1
   - libIp_v_objseg_v2_0_bitacc_cmodel.so
   - run_bitacc_cmodel
3. Execute the model:

```
./run_bitacc_cmodel ic_stats_512.txt objseg_config_512.cfg
objseg_out.txt
```

Windows (32-bit and 64-bit)

1. Copy `run_bitacc_cmodel.exe` from the /win32 or /win64 directory to the root directory.
2. Execute the model:

```
./run_bitacc_cmodel ic_stats_512.txt objseg_config_512.cfg
objseg_out.txt
```

Object Segmentation Metadata Output

Image Characterization Statistics Input

The image characterization statistics input is described in the Image Characterization Statistics Input Structure section. For additional information on the image characterization statistics input structure, see the *LogiCORE IP Image Characterization Product Guide* (PG015).

Metadata Output

This section includes an example of the object segmentation metadata output. The comments explain how to read this output. For more information on the object segmentation metadata structure, see Object Segmentation Metadata Output Structure.

The first 32 lines of the data are the metadata frame header. The header contains a frame index, the number of objects in the image frame, and the image characterization global statistics for the frame. Following the frame header are 32 object descriptions for Feature Select 1 (FS1). Each object description contains:

- The coordinate information for a box that bounds the object (X start/stop, Y start/stop)
- The centroid of the box (X, Y)
- The object density (the number of blocks inside the box that belong to the object)
- An object identifier (Cyclic Redundancy Check (CRC) of the other five lines of object information)

If less than 32 objects are found for FS1, the remaining object descriptions are still present, but the contents are set to 0. The FS1 object descriptions are followed by the FS2 object descriptions, which are followed by the FS3 objects descriptions, which are followed by the FS4 object descriptions. If fewer then four Feature Selects are instantiated, then only that number of Feature Select object descriptions are output. For example, if two Feature Selects are instantiated, then only the object descriptions for FS1 and FS2 are output.

Multiple object segmentation metadata structures can be in one output file. The next metadata structure begins directly after the end of the previous metadata structure.

```
# Metadata Frame Header
-1            # Frame Struct Valid (0xFFFFFFFF)
1             # Frame Index
12            # Total Number of Objects (FS4 thru FS1)
16777226      # FS 4 = 1, FS 3 = 0, FS 2 = 0, FS 1 = 10
              # Global Statistics
-1786936427   # Low Frequency Mean = 0x95, V Mean = 0x7D,
              # U Mean = 0x83, Y Mean = 0x95
117446148     # Saturation Mean = 0x07, Motion Mean = 0x00,
              # Edge Mean = 0x16, High Frequency Mean = 0x04
3539068       # U Variance = 0x0036, Y Variance = 0x007C
7798842       # Low Frequency Variance = 0x0077, V Variance = 0x003A
49348637     # Edge Variance = 0x02F1, High Frequency Variance = 0x001D
262144        # Saturation Variance = 0x0004, Motion Variance = 0x0000
0             # Frame Header Padding (22 lines)
…
0
# Metadata Feature Select #1
65537         # FS #1, Object 1
17826056      # XStop = 0x110, XStart = 0x108
4718656       # YStop = 0x048, YStart = 0x040
4456716       # YCentroid = 0x044, XCentroid = 0x10C
1             # Object Density (number of blocks in the object)
18677828      # Object Identifier
65538         # FS #1, Object 2
32506344
```

```
10485912
10224108
1
30212255
65539          # FS 1, Object 3
30933416
13107376
12321216
13
28115158
65540          # FS 1, Object 4
33554912
15728816
13631984
16
35717300
65541          # FS 1, Object 5
33554936
12058800
11796988
1
34406576
65542          # FS 1, Object 6
14155776
22020280
17039468
210
9240576
65543          # FS 1, Object 7
19398872
14680248
13369600
19
17105268
65544          # FS 1, Object 8
19923240
12583096
12321068
1
21823669
65545          # FS 1, Object 9
25690424
21495992
```

```
16777568
85
29425852
65546          # FS 1, Object 10
26214792
14155984
13894028
1
27066591
65536          # FS 1, Object 11
0
0
0
0
0
#FS 1 Objects 12 - 32 are repeats of Object 11
# Metadata Feature Select #2
131073         # FS 2, Object 1
33554432
33554432
16777472
4096
16912641
131072         # FS 2, Object 2
0
0
0
0
0
#FS 2 Objects 3 - 32 are repeats of Object 2
# Metadata Feature Select #3
196608         # FS 3, Object 1
0
0
0
0
0
#FS 3 Objects 2 - 32 are repeats of Object 1
# Metadata Feature Select #4
262145         # FS 4, Object 1
33554432
33554432
16777472
```

```
4096
17043713
262144        # FS 4, Object 2
0
0
0
0
0
#FS 4 Objects 3 - 32 are repeats of Object 2
```

XILINX®

Appendix E

Additional Resources

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see the Xilinx® Support website at:

http://www.xilinx.com/support.

For a glossary of technical terms used in Xilinx documentation, see:

http://www.xilinx.com/support/documentation/sw_manuals/glossary.pdf.

Note: The glossary also contains acronyms.

List of Acronyms

Table E-1: **List of Acronyms**

Acronym	Description
AMBA	Advanced Microcontroller Bus Architecture
API	Application Program Interface
AXI	Advanced eXtensible Interface
DSP	Digital Signal Processing
EDK	Embedded Development Kit
FF	Flip-Flop
FPGA	Field Programmable Gate Array
GPP	General Purpose Processor
GUI	Graphical User Interface
HDL	Hardware Description Language
I/O	Input/Output
IP	Intellectual Property
ISE	Integrated Software Environment
LSB	Least Significant Bit
LUT	Lookup Table
MHz	Mega Hertz

Table E-1: **List of Acronyms**

Acronym	Description
MM2S	Memory Map to Stream
MSB	Most Significant Bit
OSD	On Screen Display
R	Read
R/W	Read/Write
RAM	Random Access Memory
S2MM	Stream to Memory Map
VHDL	VHSIC Hardware Description Language (VHSIC an acronym for Very High-Speed Integrated Circuits)
XPS	Xilinx Platform Studio (part of the EDK software)
XST	Xilinx Synthesis Technology

Solution Centers

See the Xilinx Solution Centers for support on devices, software tools, and intellectual property at all stages of the design cycle. Topics include design assistance, advisories, and troubleshooting tips.

References

These documents provide supplemental material useful with this user guide:

- *AMBA® AXI4-Stream Protocol Specification*
- UG761, *AXI Reference Guide*
- DS768, *AXI Interconnect IP Data Sheet*
- PG015, *LogiCORE IP Image Characterization Product Guide*

To search for Xilinx documentation, go to http://www.xilinx.com/support

Technical Support

Xilinx provides technical support at www.xilinx.com/support for this LogiCORE™ IP product when used as described in the product documentation. Xilinx cannot guarantee timing, functionality, or support of product if implemented in devices that are not defined in the documentation, if customized beyond that allowed in the product documentation, or if changes are made to any section of the design labeled DO NOT MODIFY.

See the IP Release Notes Guide (XTP025) for more information on this core. For each core, there is a master Answer Record that contains the Release Notes and Known Issues list for the core being used. The following information is listed for each version of the core:

- New Features
- Resolved Issues
- Known Issues

Ordering Information

The Object Segmentation v2.0 core is provided under the Xilinx End User License Agreement and can be generated using the Xilinx® CORE Generator™ system v13.3 or higher. The CORE Generator system is shipped with the Xilinx ISE® Design Suite development software. Contact your local Xilinx sales representative for pricing and availability of additional Xilinx LogiCORE IP modules and software. Information about additional Xilinx LogiCORE IP modules is available on the Xilinx IP Center.

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
10/19/11	1.0	Initial Xilinx release.

Notice of Disclaimer