Video IP Cores for Altera DE-Series Boards

For Quartus II 12.0

1 Overview

The Altera University Program (UP) Video IP cores facilitate decoding, processing and display of video data. They are designed for use on Altera DE-series boards and work with on-board video-in and VGA chips, as well as Terasic's 5 megapixel CCD camera and LCD screen with touch panel daughtercards. This suite of IP cores comprises: a video decoder, a VGA controller, eleven video-processing cores, two direct-memory-access (DMA) cores, a character buffer and two Video Image Processing (VIP) bridges. The video decoder converts raw video input from video-in chips on Altera DE2/DE2-70/DE2-115 boards, or Terasic's 5 megapixel CCD camera, into packets that can be processed by the video-processing cores. The VGA controller core displays images by creating the timing signals required by VGA compatible monitors attached to the VGA port on the DE-series board, or the Terasic LCD screen with touch panel. The video-processing cores perform basic transformations on the video input, while the VIP bridge cores allow Altera VIP cores to be used together with Altera UP Video IP cores in more advanced applications. The video DMA cores allow video data to be stored to and retrieved from memory. The character buffer core holds ASCII characters and converts them to a video stream, so that they can be displayed on a screen.

The remainder of this manual is organized as follows: Section 2 gives a brief introduction of the cores and gives four examples to assist designers using the video IP cores. Section 3, named Background, describes in detail how the video IP core are connected, the format used to transfer data and the memory layout for stored video. A detailed description of all the UP video cores is given in Section 4.

This manual assumes that the reader is familiar with the Altera Qsys tool and how to use it.

2 Getting Started

In this section, the cores will be briefly described through the use of four examples. The examples use the VGA output and video input ports on the DE-series boards. All examples were created using Qsys, and include a Nios II processor and a 16 KB on-chip memory as a base system.

All of these examples are available in the "/Examples/IP_Core_Demos/Video_Demos_using_Qsys"' directory, which is installed along with the University Program Design Suite package.

2.1 Basic Video Out: Character Display

The first example demonstrates how to display characters on a VGA-compatible screen that is attached to the VGA port on the DE-series board. In this example, we make use of the following four cores: the VGA controller, Clock Signals, Dual-Clock FIFO and Character Buffer for VGA Display.

In this example, drawing characters on the screen is implemented using the system shown in figure 1. To display a character on the screen, users must specify the character location to the Character Buffer IP core. Once specified, the character buffer renders an image of each character and sends it to the Dual-Clock (DC) FIFO core. The DC Fifo buffers part of an image to be displayed on the screen until the VGA IP core is ready to display it. When the VGA cores is ready, the image will be displayed on the screen. It is important to note that in this example the VGA IP core and the character buffer operate at different clock frequencies. This is because the VGA IP Core needs to run at 25 MHz to properly display an image on the screen, while the Character Buffer was connected to the system clock, which runs at 50 MHz. To allow both components to work together, the Clock Signals core generates the appropriate clocks, and the DC FIFO facilitates reliable communication between the two IP cores.

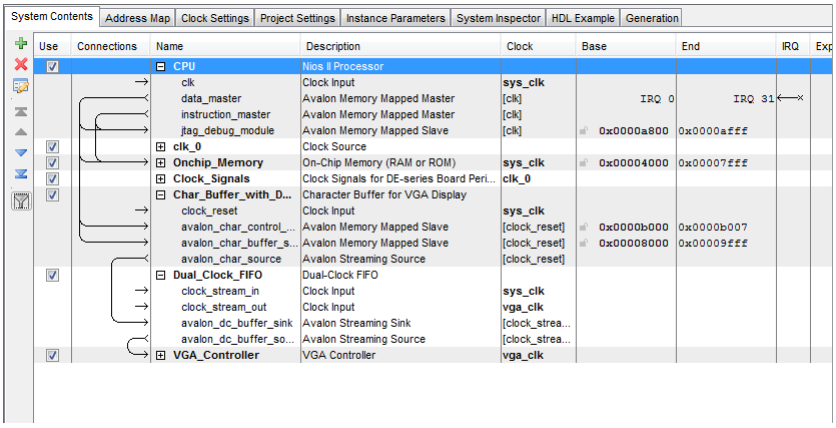


Figure 1. Character display example's Qsys system

Sample programs that run on this system are written in C. To run the example, do the following:

1. Start the Altera Monitor Program software
2. Connect the DE-series board, power it up and connect the USB cable between the board and the host computer
3. Connect a VGA-compatible monitor to the VGA port on the DE-series board and power it up
4. Open a project named "char_buffer.ncf" in the "DE2/Example_1_Char_Buffer/app_software" directory
5. Download the system onto the boards by clicking "Download System" from the Actions menu
6. Select "Compile and Load"
7. Once the program is downloaded onto the system, click the Run button.

2.2 Basic Video Out: Pixel Display

The second example demonstrates how to display graphics on a VGA compatible screen that is attached to the VGA port on the DE-series board. In this example, we make use of the following cores: Pixel Buffer DMA Controller, SRAM/SSRAM controller, RGB resampler and Scaler.

The Pixel Buffer DMA Controller IP core produces video data for the VGA controller, similarly to the Character Buffer. Unlike the Character Buffer IP core which contains memory to serve as a buffer for ASCII characters, the Pixel Buffer DMA Controller does not contain any memory. Instead, it has an Avalon memory-mapped interface that can be connected to any memory device in the Qsys system. In this example, the SRAM/SSRAM will serve as memory for the pixel buffer. The Pixel Buffer DMA Controller reads the SRAM/SSRAM and sends the image stored in memory to the VGA controller.

The image stored in memory is configured to be 320 columns by 240 rows, with 16 bits representing the color of each pixel. This image must be converted into a format of 640 columns by 480 rows with 30 bits per pixel color, because this is the input format required by the VGA controller. The resampler and scaler IP cores are used to perform the conversion.

First, the resampler core is used to change the number of bits needed to represent each pixel. This core extends the 16-bit color value to 30 bits, while maintaining the image resolution of 320 by 240. Then the scaler core converts the resolution of the image from 320 by 240 to 640 by 480. To do this the core replicates each pixel 4 times, allowing the scaler core to send a larger image to the VGA core. Please note that we still use the DC FIFO core to ensure that the scaler running at 50 MHz can reliably send the image data to the VGA core running at 25 MHz. The system used in this example is shown in Figure 2. Note: that the example for the DE0 board uses on-chip memory and stores images with a 80 x 60 resolution and 8-bit grayscale color space, since the board does not have an SRAM chip.

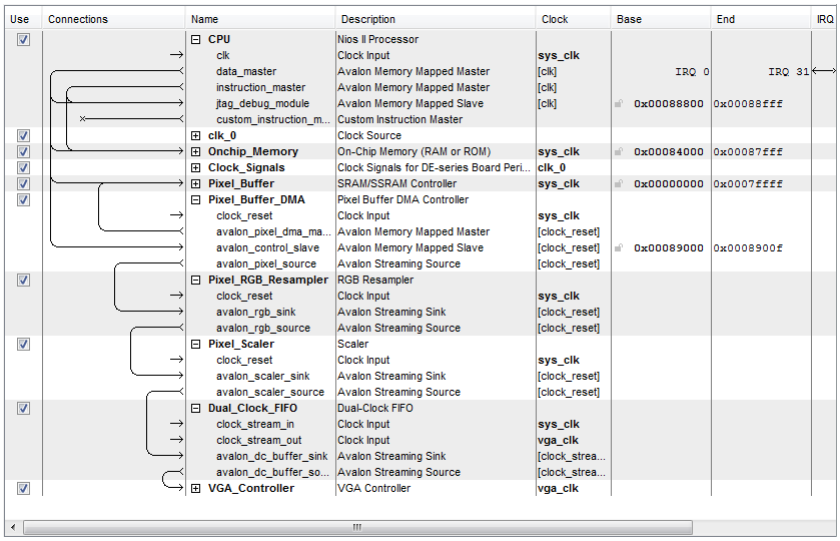


Figure 2. Graphics display example's Qsys system.

Sample programs that run on this system are written in C. To run the example, do the following:

1. Start the Altera Monitor Program software
2. Connect the DE-series board, power it up and connect the USB cable between the board and the host computer
3. Connect a VGA-compatible monitor to the VGA port on the DE-series board and power it up

4. Open a project named "pixel_buffer.ncf" in the "DE2/Example_2_Pixel_Buffer/app_software" directory
5. Download the system onto the boards by clicking "Download System" from the Actions menu
6. Select "Compile and Load"
7. Once the program is downloaded onto the system, click the Run button.

2.3 Video Out

The third example demonstrates how to merge the first two examples to display both characters and graphics simultaneously. In this example, we make use of the following cores: Pixel Buffer DMA Controller, SRAM/SSRAM controller, RGB resampler, Scaler, Character Buffer for VGA Display and Alpha Blender IP core.

The Alpha Blender core combines two video streams into one. The blender has two parts for incoming video streams, named foreground and background. The foreground stream must contain alpha values, along with the regular pixel color values. The alpha values determine the ratio used to combine the foreground and background pixels. The Character Buffer is used as the foreground and is set to automatically send alpha values with each pixel. The scaler is connected to the background port, so that the image resolution and color bits per pixel are identical for the two stream, which is required by the Alpha Blender. Once the video is combined it is sent to the Dual Clock FIFO and then to the VGA controller to be displayed. Figure 3 shows the Qsys system that corresponds to this example.

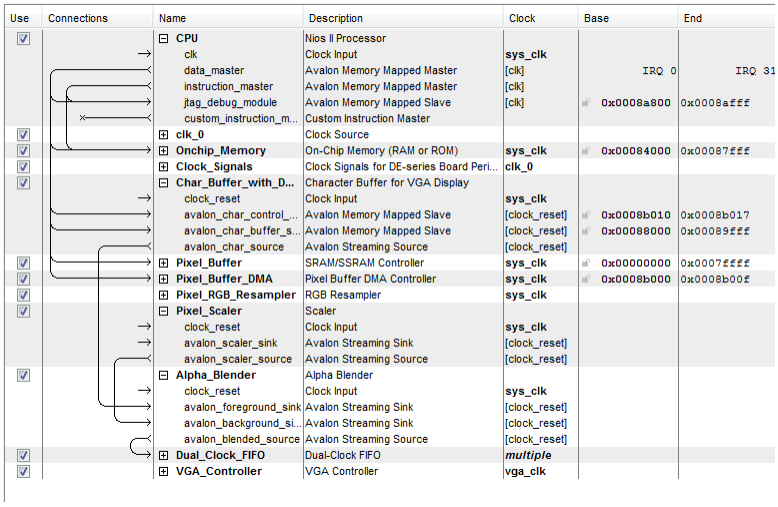


Figure 3. Graphics and character display example's Qsys system.

Sample programs that run on this system are written in C. To run the example, do the following:

1. Start the Altera Monitor Program software
2. Connect the DE-series board, power it up and connect the USB cable between the board and the host computer
3. Connect a VGA-compatible monitor to the VGA port on the DE-series board and power it up

4. Open a project named "char_pixel.ncf" in the "DE2/Example_3_Both_Buffers/app_software" directory
5. Download the system onto the boards by clicking "Download System" from the Actions menu
6. Select "Compile and Load"
7. Once the program is downloaded onto the system, click the Run button.

2.4 Video In

The final example demonstrates how to decode incoming video data and output it using the VGA controller. In this example, we make use of the following cores: Pixel Buffer DMA Controller, SRAM/SSRAM controller, RGB resampler, Scaler, Character Buffer for VGA Display, Alpha Blender IP core and Audio and Video Configurtion IP core, Video-In Decoder, Chroma resampler, Color Space Converter, Clipper, and Video-In DMA controller.

The video analog-to-digital converter (ADC) chip on the DE2/DE2-70/DE2-115 boards converts video from the composite port and streams it into the FPGA. The Audio and Video Configurtion IP core initializes the ADC with the appropriate settings for use with the UP Video IP cores. The Video-In Decoder IP core converts the video data from the ADC into Avalon streaming packets and sends them to the video DMA controller. The video DMA controller writes the video stream to the pixel buffer (SRAM/SSRAM).

The format of the video streamed from the Video-In Decoder is 720 columns by 244 rows with 16 bits per pixel in the 4:2:2 YCrCb color space. The video must be converted into a format of 320 x 240 rows with 16 bits per pixel in the RGB color space, because this is the format required by the pixel buffer. The Chroma Resampler, Color-Space Converter, RGB Resampler, Clipper and Scaler IP cores are used to perform the conversion.

First, the Chroma Resampler converts the pixel from the 4:2:2 YCrCb to the 4:4:4 YCrCb formats, while maintaining the frame resolution of 720 x 244. Then, the Color Space Converter converts between the 4:4:4 YCrCb and the 24-bit RGB color spaces. Next, the RGB Resampler converts the stream between the 24-bit RGB and 16-bit RGB formats. Then, the Clipper trims the stream from the 720 x 244 resolution to a 640 x 240 resolution by dropping the columns and rows around the exterior of the frame. Lastly, the Scaler reduces the stream to 320 x 240 by dropping every other pixel. Now that the video stream is in the correct format, the Video-In DMA Controller IP core transfers the stream to the SRAM/SSRAM. Figure 4 shows the Qsys system that correspondes to this example.

To run the example, do the following:

1. Start the Altera Monitor Program software
2. Connect the DE-series board, power it up and connect the USB cable between the board and the host computer
3. Connect a VGA-compatible monitor to the VGA port on the DE-series board and power it up
4. Connect a NTSC video source to the composite video port on the DE-series board and power it up
5. Open a project named "video_in.ncf" in the "DE2/Example_4_Video_In/app_software" directory
6. Download the system onto the boards by clicking "Download System" from the Actions menu

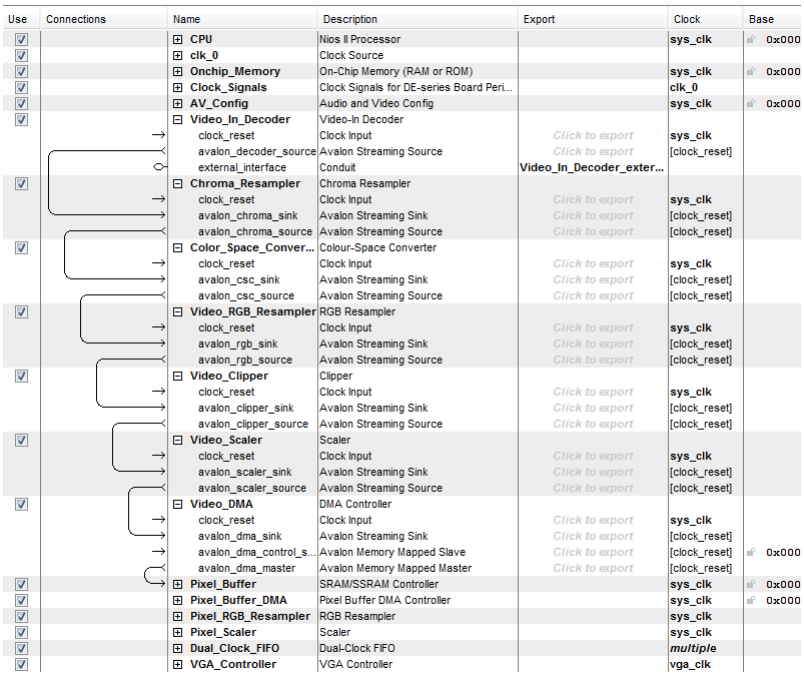


Figure 4. Video in example's Qsys system.

3 Background

In this and the following sections, detailed descriptions of each IP core are given. This section contains information common to most IP cores, while section 4 discussed each core individually.

Video is produced by displaying frames (or images) in rapid succession. In a typical video, frames are displayed between 30 and 120 times per second. A frame is a two-dimensional array of pixels as depicted in Figure 5.

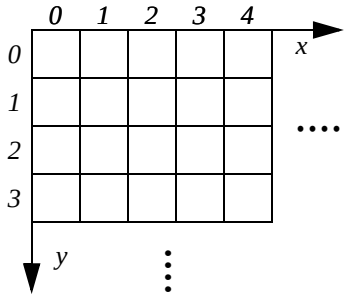


Figure 5. Video frame's screen layout.

The resolution of a frame is defined as the number of pixels in the x and y axes. An example resolution is 640 × 480, which has 640 pixels across the x axis and 480 pixels down the y axis, as shown in Figure 6. Therefore, each pixel location in a frame can be identified by an (x,y) coordinate, with (0,0) being in the top-left corner.

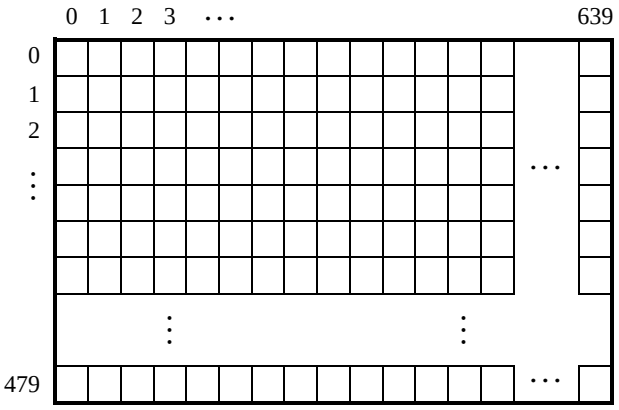


Figure 6. A frame with a 640 × 480 resolution.

The next four sections describe how frames are mapped into memory, how they are transfered between the UP video IP cores, and several formats that are used to represent individual pixels.

3.1 Memory Layout for Video Frames

Frames are mapped to a memory's address space in one of two modes. They are:

- Consecutive mode — the pixel addresses are consecutively laid out in the addressable space. For example, for a 640 × 480 resolution, the pixel at screen coordinate (0, 0) is at the offset 0, (1, 0) is at offest 1, ... (639, 0) is at offset 639, (0, 1) is at offset 640, and so on.

  The address format is shown in Figure 7a. The k value, shown in the figure, is related to the frame's resolution as follows:

  $$k = ceil(log_2(X \times Y))$$

  where X and Y are the resolution in the x, y directions, respectively. For example, for a 640 × 480 resolution, shown in Figure 7b, we have

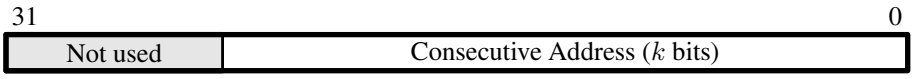
  $$k = ceil(log_2(640 \times 480)) = 19$$

- X-Y mode — the address contains x and y coordinates. The address format is shown in Figure 8a. The values of m and n, shown in the figure, are related to the frame's resolution as follows:

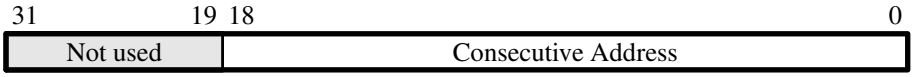
  $$m = ceil(log_2 X)$$
  $$n = ceil(log_2 Y)$$

  where X and Y are the resolution in the x, y directions, respectively. For example, for a 640 × 480 resolution, shown in Figure 8b, we have

  $$m = ceil(log_2 640) = 10$$
  $$n = ceil(log_2 480) = 9$$

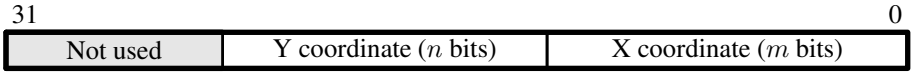


a) Consecutive address format

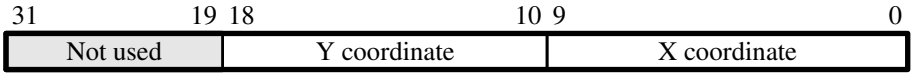


b) Consecutive address format for the 640 × 480 resolution

Figure 7. Address format for the Consecutive mode



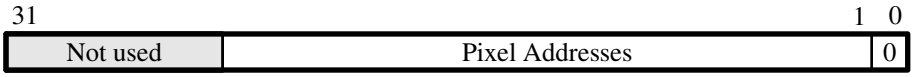
a) X-Y address format



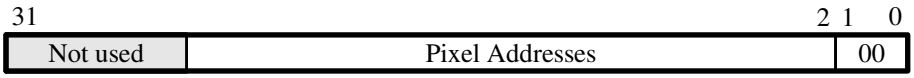
b) X-Y address format for the 640 × 480 resolution

Figure 8. Address format for the X-Y mode

The above addressing examples assume that the color of each pixel is represented with 8 bits. If the pixel color is represented by more than 8 bits, the addresses must be shifted to the left by the appropriate amount. Figure 9 shows generic addressing for 16-bit and 32-bit color pixel formats.



a) 16-bit data width



b) 32-bit data width

Figure 9. Address format based on pixel's data width

3.2 Video Stream Packet Format

The UP video IP cores transfer frames using Avalon Streaming interfaces. Each packet in the stream represents one frame of video data. The video frames are transfered one pixel at a time in row-major order. The first pixel, the top-left pixel in the frame, is signalled by the start-of-packet bit in the Avalon Streaming interface. The last pixel, the bottom-left pixel in the frame, is signalled by the end-of-packet bit in the Avalon Streaming interface. Figure 10 shows this streaming video packet representation.

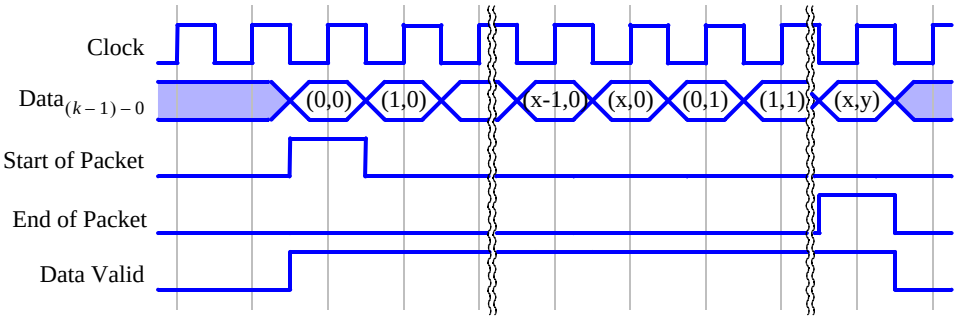


Figure 10. Video frame's streaming packet format.

The format of each pixel in the packet depends on the video frame's color space. The two color spaces used by the UP video IP cores are RGB and YCrCb, each having several different modes. Some of the video IP cores do not require knowledge of the specific color space and mode of the video stream it will process, but do require knowledge of the number of bits per pixel. For these cores, it will be important to know the number of bits per color and the number of color planes of the color space and mode of the incoming packets.

3.3 RGB Color Space

The Red-Green-Blue (RGB) color space contains independent intensity values for each of the primary colors: red, green and blue. The range of the intensity for each color depends on the number of associated bits. The UP video IP cores can use the following RBG color ranges:

- 8-Bit RGB — This format uses 3 bits for red, and 3 bits for green and 2 bits for blue as shown in Figure 11. This mode is defined as 8 bits per color and one color plane.

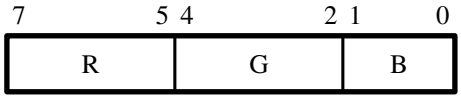


Figure 11. 8-bit RGB Color Space.

- 9-bit RGB — This format uses 3 bits for each color as shown in Figure 12. This mode is defined as 3 bits per color and three color planes.

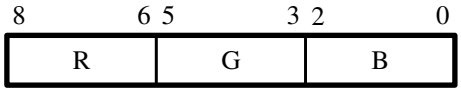


Figure 12. 9-bit RGB Color Space.

- 16-Bit RGB — This format uses 5 bits for red, and 6 bits for green and 5 bits for blue as shown in Figure 13. This mode is defined as 16 bits per color and one color plane.

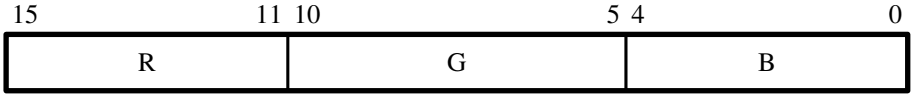


Figure 13. 16-bit RGB Color Space.

- 24-bit RGB — This format uses 8 bits for each color as shown in Figure 14. This mode is defined as 8 bits per color and three color planes.

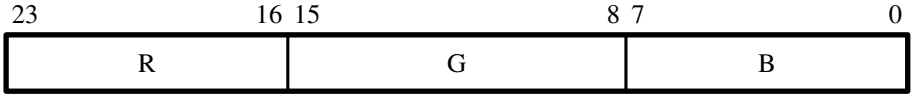


Figure 14. 24-bit RGB Color Space.

- 30-bit RGB — This format uses 10 bits for each color as shown in Figure 15. This mode is defined as 10 bits per color and three color planes.

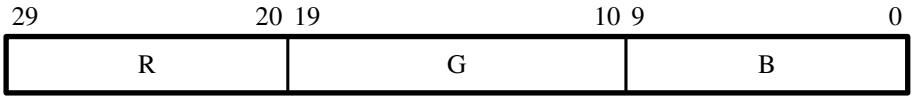


Figure 15. 30-bit RGB Color Space.

- 16-bit RGBA — This format contains alpha values as well as RGB and uses 4 bits of each color as shown in Figure 16. This mode is defined as 4 bits per color and four color planes.

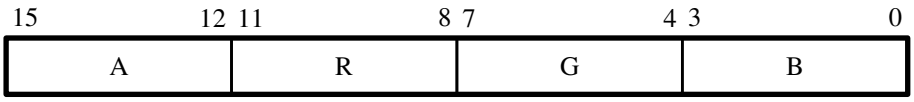


Figure 16. 16-bit RGBA Color Space.

- 32-bit RGBA — This format contains alpha values as well as RGB and uses 8 bits of each color as shown in Figure 17. This mode is defined as 8 bits per color and four color planes.
- 40-bit RGBA — This format contains alpha values as well as RGB and uses 10 bits of each color as shown in Figure 18. This mode is defined as 10 bits per color and four color planes.

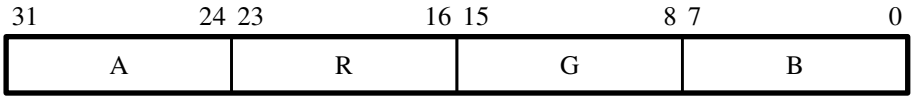


Figure 17. 32-bit RGBA Color Space.

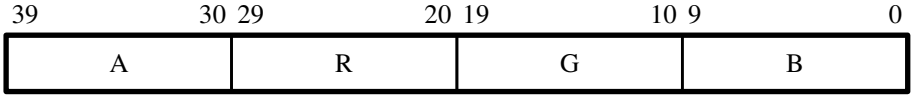


Figure 18. 40-bit RGBA Color Space.

- 8-Bit Grayscale — This is a special case of the RGB color space where are three colors have the same intensity and therefore produces shades of gray. Figure 19 shows to format of the 8-bit Grayscale data. This format is equivalent to YCrCb 4:0:0 color space. This mode is defined as 8 bits per color and one color plane.

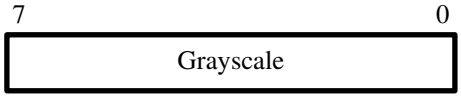


Figure 19. 8-bit Grayscale RGB Color Space.

- Bayer Pattern — This is a special case of the RGB color space where each pixel has a value for only one of the three colors. The pattern of the colors in the frame is shown in 20. Figure 21 shows to format of the bayer pattern data. This mode is defined as 8 bits per color and one color plane.

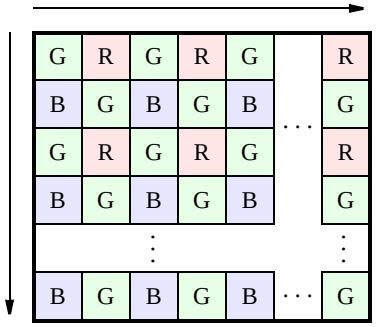


Figure 20. Bayer pattern layout.

3.4 YCrCb Color Space

The Luminance-Chrominance (YCrCb) color space contains information about the brightness (luminance or luma) and color (chrominance or chroma). The color is represented as two components, namely chrominance-red (Cr) and

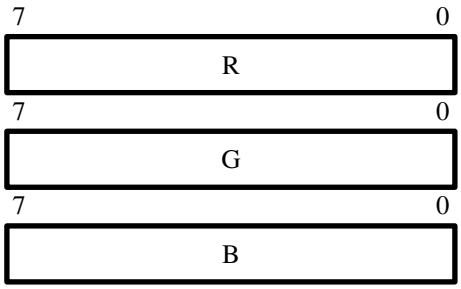


Figure 21. Bayer Pattern RGB Color Space.

chrominance-blue (Cb). The UP video IP cores use 8 bits for each of Y, Cr and Cb. The following lists the YCrCb color space varieties used by the UP video IP cores:

- YCrCb 4:4:4 — This format is the normal YCrCb with all components as shown in Figure 22. This mode is defined as 8 bits per color and three color planes.

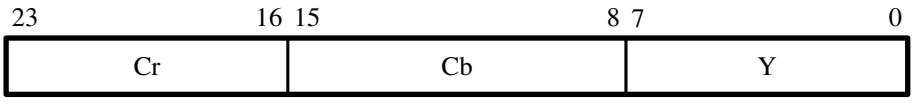


Figure 22. YCrCb 4:4:4 Color Space.

- YCrCb 4:2:2 — This format has only half of the Cr and Cb components. Each consecutive pixel has alternating Cr or Cb components, with the first pixel in the frame starting with the Y and Cb pixel. Figure 23 shows two consecutive pixels for this format. This mode is defined as 8 bits per color and two color planes.

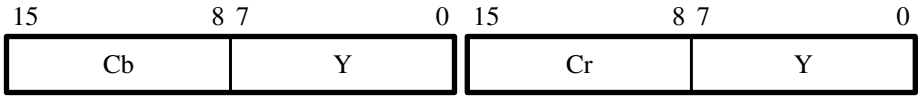


Figure 23. YCrCb 4:2:2 Color Space.

- YCrCb 4:0:0 — This format only as the Y component as shown in Figure 24. This format is equivalent to 8-Bit Grayscale RGB color space. This mode is defined as 8 bits per color and one color plane.

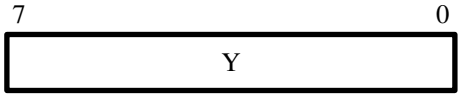


Figure 24. YCrCb 4:0:0 Color Space.

4 Video IP Core Descriptions

In this section, each IP core is described in detail.

4.1 Alpha Blender

The Alpha Blender IP core combines two video streams into one. The two incoming streams, called foreground and background are blended together to create an output stream. The foreground must be in the 40-bit RGBA format, while the background must be in the 30-bit RGB format. The generated output stream is in the 30-bit RGB format. Figure 25 shows the block diagram of the core.

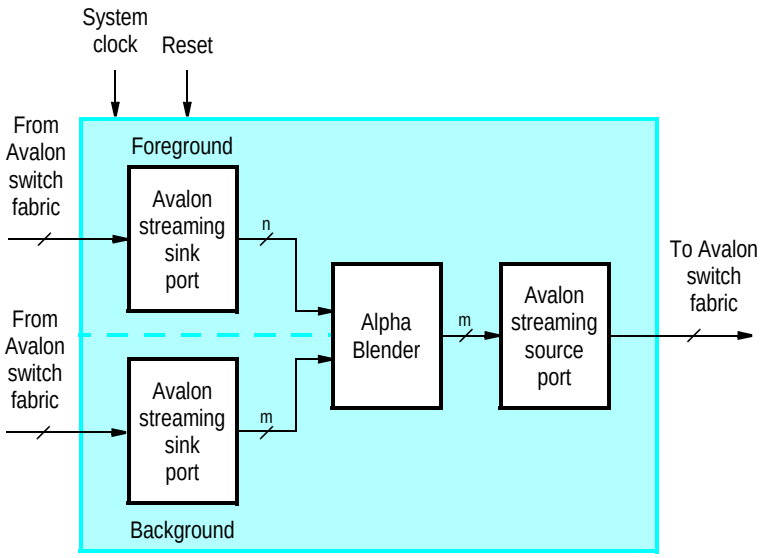


Figure 25. Alpha Blender core's block diagram.

The foreground and background streams are combined using the formula:

$$C_n = \alpha \times C_f + (1 - \alpha)C_b$$

C_n is the new outgoing pixel color, α is a number between 0 and 1, C_f is the incoming foreground pixel colors and C_b is the incoming background pixel color. To blend the streams, this formula is computed three times, once of each color plane, namely, the red, green and blue color planes.

The key parameter in the blending process is α, which is provided as part of the foreground input stream. The foreground input stream consists of a 30-bit RGB value, same as the background stream, and a 10-bit value A. The α parameter is derived by dividing the unsigned 10-bit value A by 1023.

The Alpha Blender has two modes of operation: simple and normal. In the simple mode, the alpha value is rounded to either 0 or 1, which simplifies the blending circuitry. In the normal mode, blending occurs exactly as described above. The mode is selected using the Alpha Blender Qsys Wizard as shown in Figure 26.

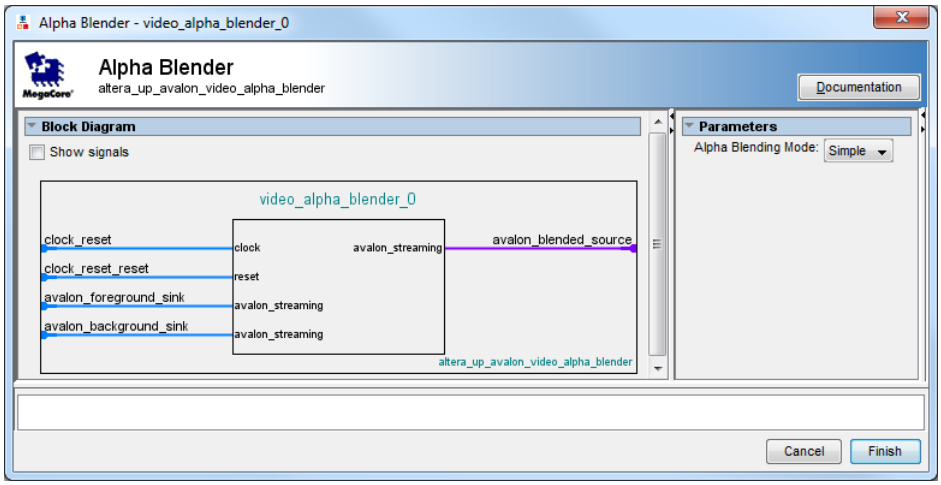


Figure 26. Alpha Blender's Qsys wizard.

4.2 Bayer Pattern Resampler

The Bayer Pattern Resampler converts a video stream from the Bayer Pattern format to the 24-bit RGB format. Four adjacent pixels from the incoming stream are combined into one, as shown in Figure 27. The red and green values from the Bayer Pattern are copied to the new pixel. The averaged value of the two green values from the Bayer Pattern are used in the new pixel. The resulting outgoing stream will have a resolution with half the width and half the height of the incoming stream. Figure 28 shows the block diagram of the Bayer Pattern Resampler.

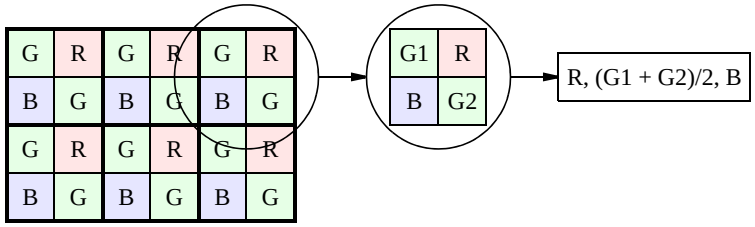


Figure 27. Bayer Pattern Resampler's method of conversion.

4.2.1 Instantiating the Core in Qsys

Designers use the Bayer Pattern Resampler's configuration wizard in Qsys to specify its settings. The following configurations are available and shown in Figure 29:

- Video Source — Specifies the source of the Bayer Pattern, and by extension, it specifies the screen resolution of the incoming stream.

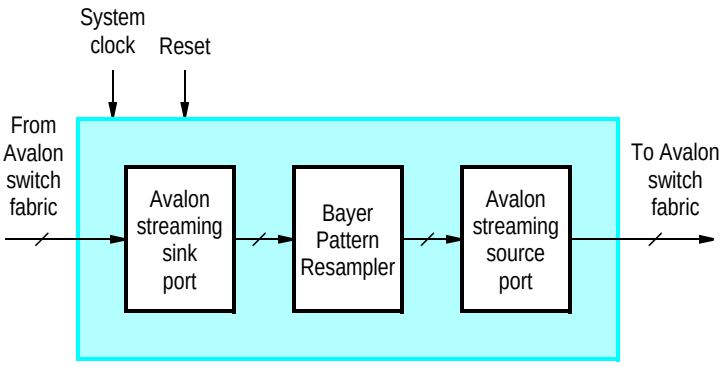


Figure 28. Bayer Pattern Resampler core's block diagram.

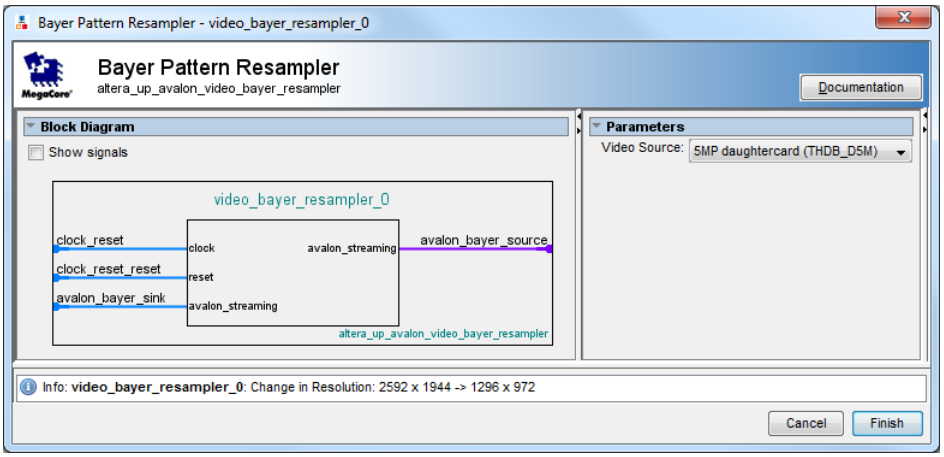


Figure 29. Bayer Pattern Resampler's Qsys wizard.

4.3 Character Buffer for VGA Display

The Character Buffer for VGA Display renders ASCII characters into graphical representation for display. A program running on a Nios II processor can send ASCII character codes to the Character Buffer's Avalon interface, named *avalon_char_slave*. The core stores the characters in its on-chip memory. The DMA controller reads the ASCII characters from the on-chip memory and sends them to the character renderer. The renderer converts the ASCII characters into their graphical representation and send them out via an Avalon Streaming interface. Figure 30 shows the block diagram of the character buffer.

The Character Buffer supports one color mode, which is that characters are drawn in white with a transparent background.

Upon initialization or reset, the Character Buffer sets all the characters to "space", so no characters will be displayed. This "clear screen" operation can take up to 5000 clock cycles.

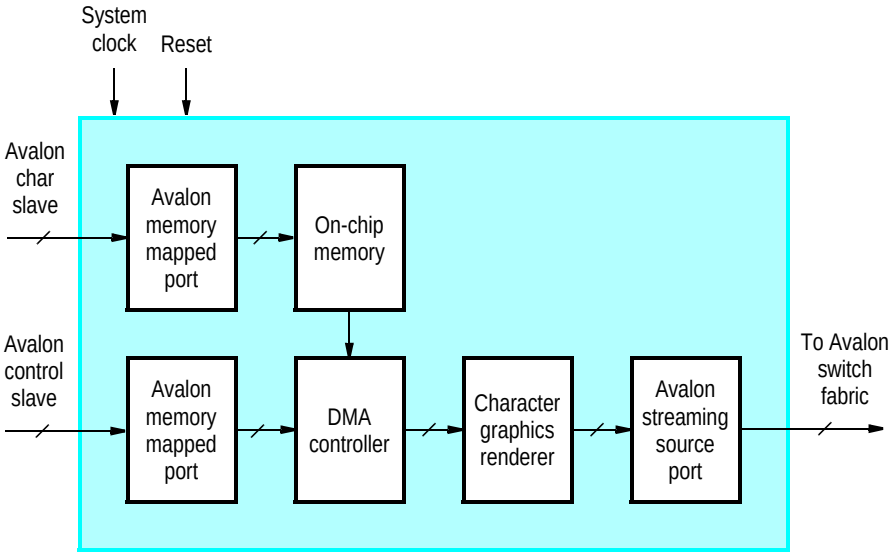
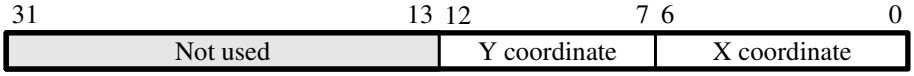
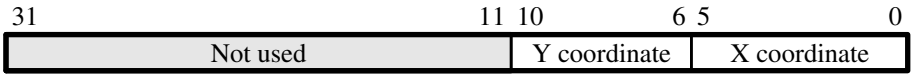


Figure 30. Character Buffer for VGA Display core's block diagram.

The Character Buffer's resolution is defined by the number of characters per line and the number of lines per screen. The Character Buffer supports one resolution per output device. For the on-board VGA DAC, the resolution is 80 characters by 60 lines. For the LCD with touchscreen the resolution is 50 × 30. The core only supportes the X-Y addressing mode, which is shown in Figure 31 for the two valid resolutions.



a) X-Y address format for the 80 × 60 resolution



b) X-Y address format for the 50 × 30 resolution

Figure 31. Character address format

4.3.1 Instantiating the Core in Qsys

Designers use the Character Buffer's configuration wizard in Qsys to specify the desired features. The following configurations are available and shown in Figure 32:

- Video-Out Device — Specifies the device being used, and by extension the screen resolution.
- Enable Transparency — When enabled the output format is set to 40-bit RGBA. This setting must be enabled if the Character Buffer and Pixel Buffer are to be used together.

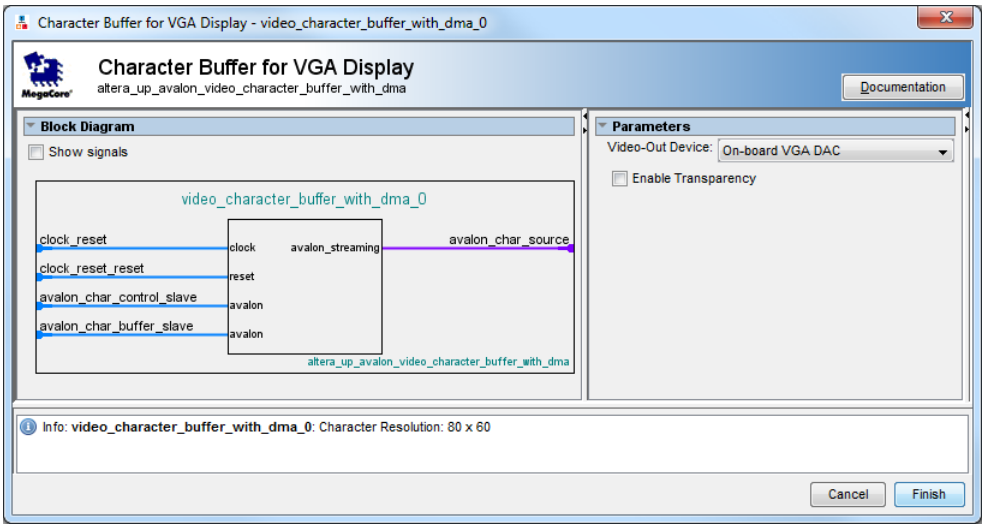


Figure 32. Character Buffer for VGA Display's Qsys wizard.

4.3.2 Software Programming Model

Register Map

Device drivers control and communicate with the Character Buffer through two Avalon memory-mapped interfaces, named *avalon_control_slave* and *avalon_char_slave*. The *avalon_char_slave* interface has a byte data width for ASCII characters and is addressed using the X-Y mode. The *avalon_control_slave* interface consists of the two registers shown in Table 1. The *Control* register provides the ability to clear the screen by writing to the *R* bit, which is bit 16 of this register. The *R* bit remains set to 1 until all characters have been cleared, and then *R* is set to 0. The *Resolution* register, which is read-only, provides two values: the number of characters per line, in bits 15-0, and the number of lines per screen, in bits 31-16.

Table 1. Character Buffer register map

Offset in bytes	Register Name	R/W	Bit Description		
			31...17	16	15...0
0	`Control`	RW	*(1)*	R	*(1)*
4	`Resolution`	R	Lines		Chars

Notes on Table 1:

(1) Reserved. Read values are undefined. Write zero.

Programming with the Character Buffer

The Character Buffer is packaged with C-language functions that are accessible through the hardware abstraction layer (HAL). These functions implement the basic operations that control the Character Buffer.

To use the functions, the C code must include the statement:

```
#include "altera_up_avalon_character_buffer.h"
```

alt_up_char_buffer_init

Prototype:	`void alt_up_char_buffer_init(alt_up_char_buffer_dev *char_buffer)`
Include:	`<altera_up_avalon_character_buffer.h>`
Parameters:	`char_buffer` – struct for the character buffer device
Description:	Initialize the name of thestructure.

alt_up_char_buffer_open_dev

Prototype:	`alt_up_char_buffer_dev* alt_up_char_buffer_open_dev(const char *name)`
Include:	`<altera_up_avalon_character_buffer.h>`
Parameters:	`name` – the character buffer component name in Qsys.
Returns:	The corresponding device structure, or NULL if the device is not found
Description:	Opens the character buffer device specified by *name* .

alt_up_char_buffer_draw

Prototype:	`int alt_up_char_buffer_draw(alt_up_char_buffer_dev *char_buffer, unsigned char ch, unsigned int x, unsigned int y)`
Include:	`<altera_up_avalon_character_buffer.h>`
Parameters:	`ch` – the character to draw `x` – the *x* coordinate `y` – the *y* coordinate
Returns:	0 for success, -1 for error (such as out of bounds)
Description:	Draw a character at the location specified by *(x, y)* on the VGA monitor with white color and transparent background.

alt_up_char_buffer_string

Prototype:	`int alt_up_char_buffer_string(alt_up_char_buffer_dev *char_buffer, const char *ptr, unsigned int x, unsigned int y)`
Include:	`<altera_up_avalon_character_buffer.h>`
Parameters:	`ch` – the character to draw `x` – the *x* coordinate `y` – the *y* coordinate
Returns:	0 for success, -1 for error (such as out of bounds)
Description:	Draw a NULL-terminated text string at the location specified by *(x, y)* .

alt_up_char_buffer_clear

Prototype:	`int alt_up_char_buffer_clear(alt_up_char_buffer_dev *char_buffer)`
Include:	`<altera_up_avalon_character_buffer.h>`
Parameters:	–
Returns:	0 for success
Description:	Clears the character buffer's memory.

4.4 Chroma Resampler

The Chroma Resampler converts video streams between the YCrCb color space formats. Converting between the various formats never effects the Y value. Figure 33 shows a block diagram of the core.

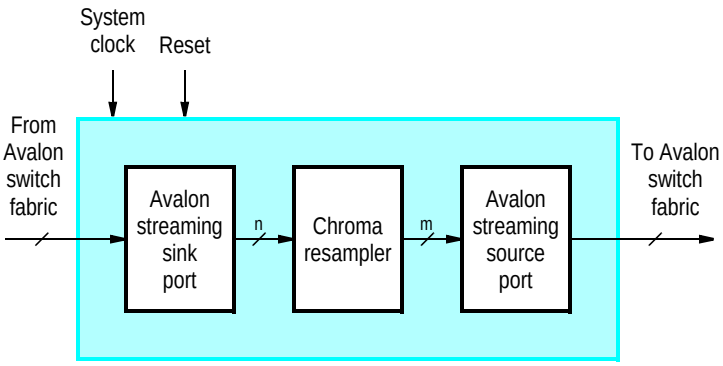


Figure 33. Chroma Resampler core's block diagram.

The following lists the valid conversions and describes how they are performed:

- 4:4:4 to 4:2:2 — Drops half of the Cr and Cb values, alternating every pixel which value is dropped, starting with the Cr value being dropped from the first pixel
- 4:4:4 to 4:0:0 — Drops the Cr and Cb values from every pixel
- 4:2:2 to 4:0:0 — Drops the Cr and Cb values from every pixel
- 4:2:2 to 4:4:4 — Duplicates the missing Cr and Cb values from the last incoming pixel
- 4:0:0 to 4:4:4 — Inserts a value of 128 for both Cr and Cb
- 4:0:0 to 4:2:2 — Inserts a value of 128 for both Cr and Cb, when appropriate

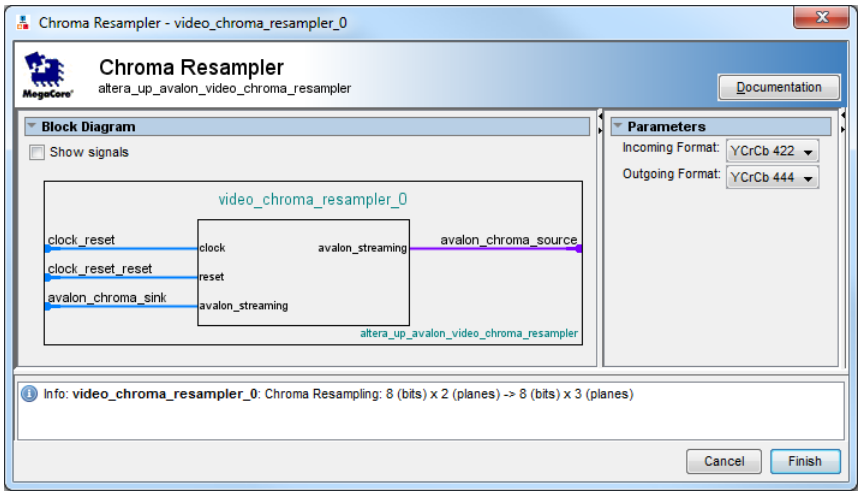


Figure 34. Chroma Resampler's Qsys wizard.

4.4.1 Instantiating the Core in Qsys

Designers use the Chroma Resampler's configuration wizard in Qsys to specify the desired features. The following configurations are available and shown in Figure 34:

- Incoming Format — Specifies the YCrCb format of the incoming stream
- Outgoing Format — Specifies the desired YCrCb format of the outgoing stream

Note: the input and output formats cannot be the same.

4.5 Clipper

The Clipper IP core modifies the resolution of video stream. The clipper can add or drop entire rows and columns of pixels from the top, bottom, right and left sides of video frames. Figure 35 shows a block diagram of the core.

4.5.1 Instantiating the Core in Qsys

Designers use the Clipper's configuration wizard in Qsys to specify the desired features. The following configurations are available and shown in Figure 36:

- Incoming Frame Resolution
  - Width (# of pixels) — Specifies the incoming stream's width
  - Height (# of lines) — Specifies the incoming stream's height
- Reduce Frame Size
  - Columns to remove from the left side — Specifies the number of columns to drop from the left side of the frame

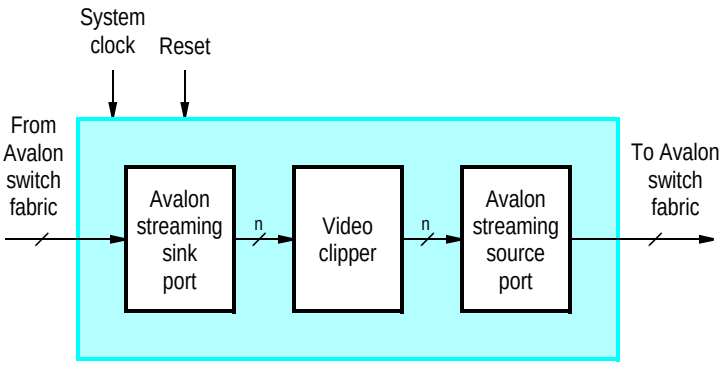


Figure 35. Clipper core's block diagram.

  - Columns to remove from the right side — Specifies the number of columns to drop from the right side of the frame
  - Rows to remove from the top — Specifies the number of rows to drop from the top of the frame
  - Rows to remove from the Bottom — Specifies the number of rows to drop from the bottom of the frame
- Enlarge Frame Size
  - Columns to add from the left side — Specifies the number of columns to add to the left side of the frame
  - Columns to add from the right side — Specifies the number of columns to add to the right side of the frame
  - Rows to add from the top — Specifies the number of rows to add to the top of the frame
  - Rows to add from the Bottom — Specifies the number of rows to add to the bottom of the frame
  - Added pixel value for plane 1 — Specifies the pixel value, in hexadecimal, for the first color plane, when columns or rows are added to the frame
  - Added pixel value for plane 2 — Specifies the pixel value, in hexadecimal, for the second color plane, when columns or rows are added to the frame
  - Added pixel value for plane 3 — Specifies the pixel value, in hexadecimal, for the third color plane, when columns or rows are added to the frame
  - Added pixel value for plane 4 — Specifies the pixel value, in hexadecimal, for the fourth color plane, when columns or rows are added to the frame
- Pixel Format
  - Color Bits — Specifies the number of bits per color plane
  - Color Planes — Specifies the number of color planes per pixel

4.6 Color-Space Converter

The Color-Space Converter converts video streams between the YCrCb and RGB color spaces. The converter can take an input stream in either the YCrCb (4:4:4 or 4:0:0) or 24-bit RGB color space and produces a stream in the opposite color space. Figure 37 shows the block diagram of the core.

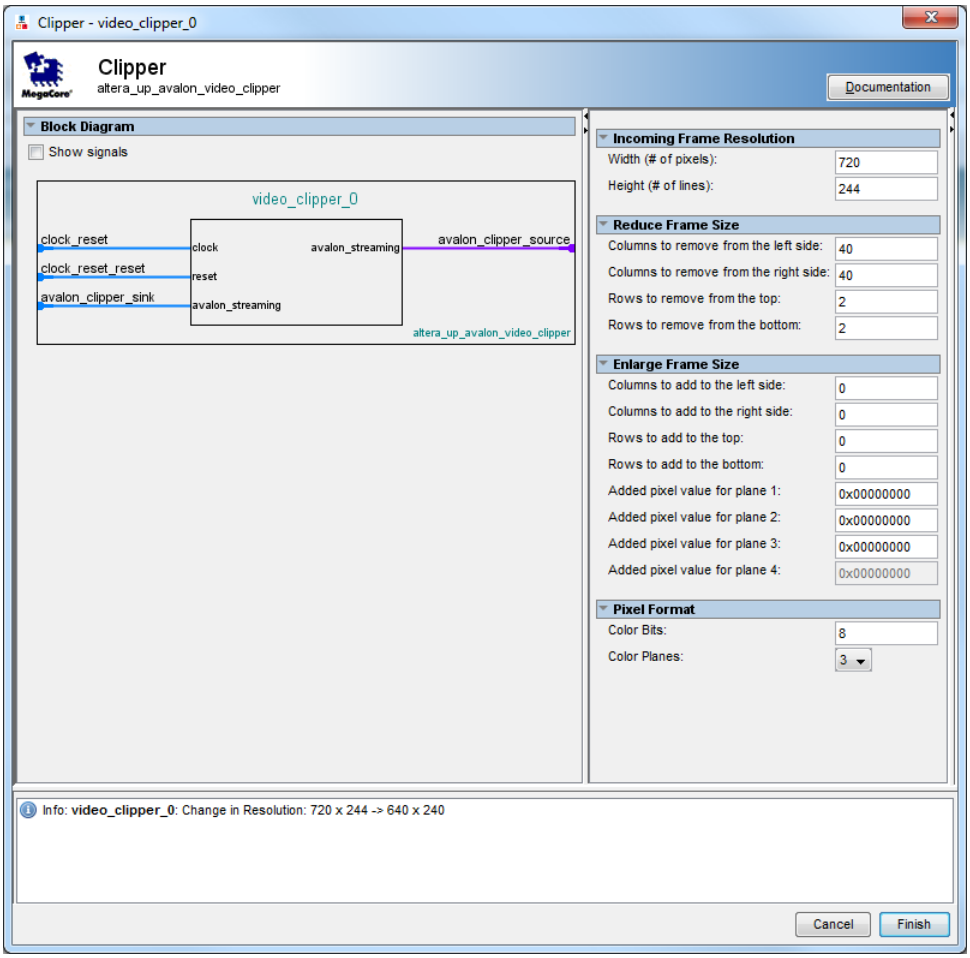


Figure 36. Character Buffer for VGA Display's Qsys wizard.

The following lists the formulas used to perform the conversions:

- 4:4:4 YCrCb to 24-bit RGB

$$\begin{aligned} R &= 1.164(Y-16) + 1.596(C_r-128) \\ G &= 1.164(Y-16) - 0.813(C_r-128) - 0.392(C_b-128) \\ B &= 1.164(Y-16) + 2.017(C_b-128) \end{aligned}$$

- Grayscale (Y) to 24-bit RGB

$$\begin{aligned} R &= Y \\ G &= Y \\ B &= Y \end{aligned}$$

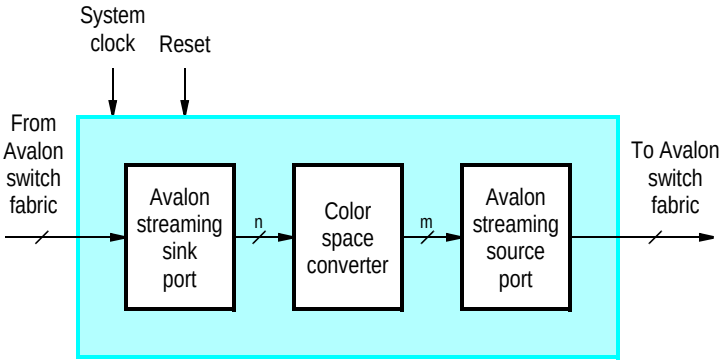


Figure 37. Color-Space Converter core's block diagram.

- 24-bit RGB to 4:4:4 YCrCb

$$\begin{aligned} Y &= 0.257 \times R + 0.504 \times G + 0.098 \times B + 16 \\ C_r &= 0.439 \times R - 0.368 \times G - 0.071 \times B + 128 \\ C_b &= -0.148 \times R - 0.291 \times G + 0.439 \times B + 128 \end{aligned}$$

- 24-bit RGB to 4:0:0 YCrCb

$$Y = 0.257 \times R + 0.504 \times G + 0.098 \times B + 16$$

4.6.1 Instantiating the Core in Qsys

Designers use the Color-Space Converter's configuration wizard in Qsys to specify the desired features. The following configurations are available and shown in Figure 38:

- Color-Space Conversion — Specifies the desired conversion and by extension the incoming and outgoing color formats

4.7 DMA Controller for Video

The DMA Controller IP core stores and retrieves video frames to and from memory. The DMA controller has two modes of operation: "from stream to memory" and "from memory to stream". When in the "from stream to memory" mode, the core stores frames from an incoming stream to an external memory. The core uses its Avalon memory-mapped master interface to send the data to the memory. When in the "from memory to stream" mode, the DMA controller uses its Avalon memory-mapped master interface to read video frames from an external memory. Then, it sends those video frames out via its Avalon streaming interface. Figure 39 shows a block diagram of the core. The Avalon streaming sink interface is only present when the controller is in the "from stream to memory" mode. The Avalon streaming source interface is only present when the controller is in the "from memory to stream" mode.

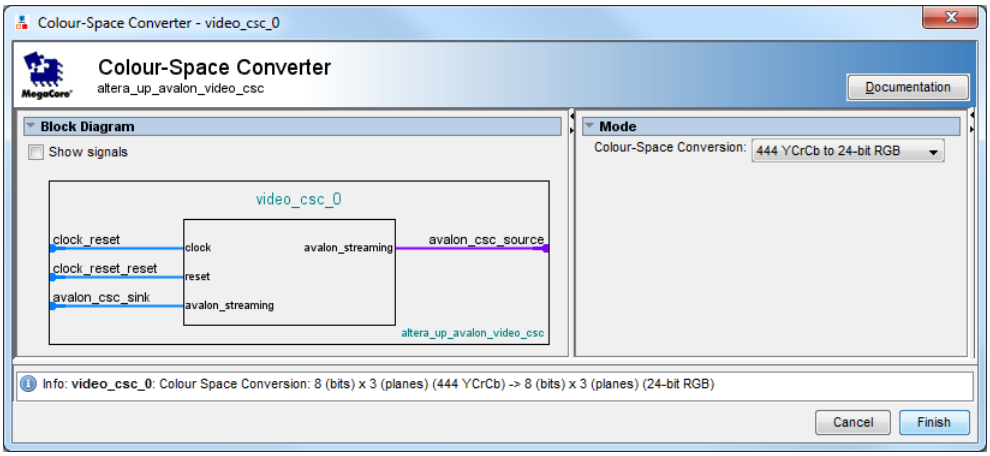


Figure 38. Color-Space Converter's Qsys wizard.

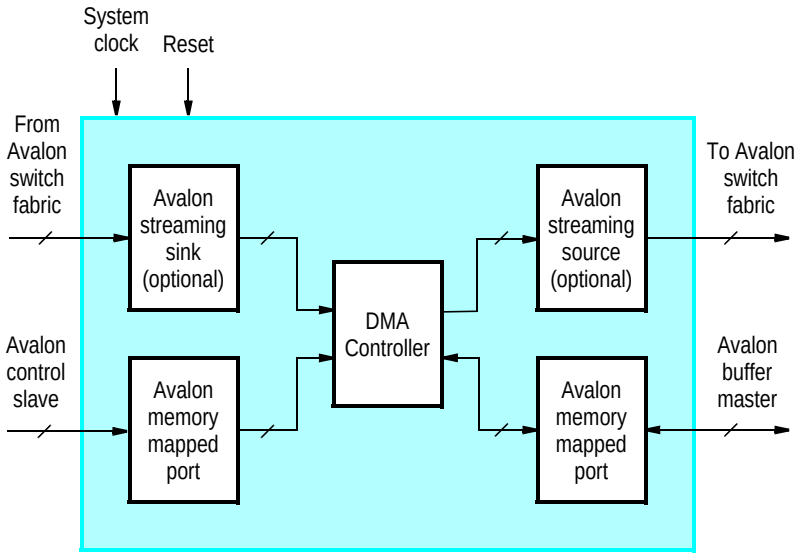


Figure 39. DMA Controller for Video core's block diagram.

The DMA controller's Avalon memory-mapped slave interface, named *avalon_dma_control_slave*, is used to communicate with the controller's internal registers. These internal registers and their functions are described in section 4.7.2.

The DMA controller can use either the consecutive or X-Y addressing modes to read and write frames from and to memory. Also, the controller can store and retrieve pixels of any format and the core will automatically adjust its address for the choosen format's word length.

4.7.1 Instantiating the Core in Qsys

Designers use the DMA Controller's configuration wizard in Qsys to specify the desired features. The following configurations are available and shown in Figure 40:

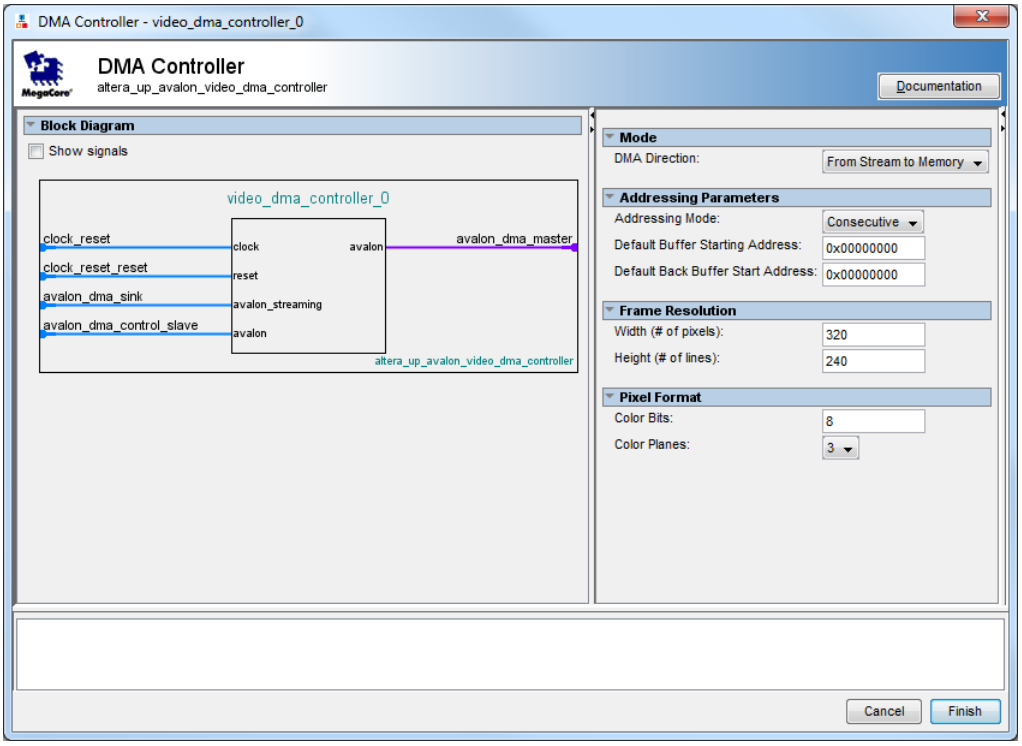


Figure 40. DMA Controller's Qsys wizard.

- Mode
  - DMA Direction — Specifies whether a video stream is to be stored to or retrieved from memory
- Addressing Parameters
  - Addressing Mode — Specifies the addressing mode
  - Default Buffer Start Address — The start address of the buffer upon reset
  - Default Back Buffer Start Address — The start address of the back buffer upon reset (can be equal to the *Default Buffer Start Address*, if no back buffer is desired)
- Frame Resolution
  - Width (# of pixels) — Specifies the incoming stream's width
  - Height (# of lines) — Specifies the incoming stream's height
- Pixel Format
  - Color Bits — Specifies the number of bits per color plane
  - Color Planes — Specifies the number of color planes

4.7.2 Software Programming Model

4.7.3 Register Map

Device drivers control and communicate with the DMA controller's Avalon memory-mapped interface, named *avalon_control_slave*. The *avalon_control_slave* provides an interface for controlling the DMA 's operation, and for obtaining status information. It consists of four registers, as shown in Table 2.

Table 2. DMA Controller register map

Offset in bytes	Register Name	R/W	Bit Description							
			31...24	23...16	15...12	11...8	7...6	5...2	1	0
0	Buffer	R	Buffer's start address							
4	BackBuffer	R/W	Back buffer's start address							
8	Resolution	R	Y		X					
12	Status	R	m	n	*(1)*	CB	CP	*(1)*	A	S

Notes on Table 2:

(1) Reserved. Read values are undefined. Write zero.

The *Buffer* register holds the 32-bit address of the start of the memory buffer. This register is read-only, and shows the address of the first pixel of the frame currently being output. The *BackBuffer* register allows the start address of the frame to be changed under program control. To change the frame being displayed, the desired frame's start address is first written into the *BackBuffer* register. Then, a second write operation is performed on the *Buffer* register. The value of the data provided in this second write operation is not used by the controller. Instead, it interprets a write to the *Buffer* register as a request to *swap* the contents of the *Buffer* and *BackBuffer* registers. The swap does not occur immediately. Instead, the swap is done after the DMA controller reaches the last pixel associated with the frame currently being output. While the controller is not yet finished outputting the current frame, bit *S* of the *Status* register will be set to 1. After the current screen is finished, the swap is performed and bit *S* is set to 0.

The *Resolution* register in Table 2 provides the *X* resolution of the screen in bits 15-0, and the *Y* resolution in bits 31-16. Finally, the *Status* register provides information for the DMA controller. The fields available in this register are shown in Table 3.

Table 3. Status register bits

Bit number	Bit name	R/W	Description
31 - 24	m	R	Width of Y coordinate address
23 - 16	n	R	Width of X coordinate address
11 - 8	CB	R	Number of color bits minus one
7 - 6	CP	R	Number of color planes minus one
1	A	R	Addressing mode: 0 (X,Y), or 1 (consecutive)
0	S	R	Swap: 0 when swap is done, else 1

4.8 Dual-Clock FIFO

The Dual-Clock FIFO buffers video data and help transfer a stream between two clock domains. Video streams into the core at the input clock frequency. The data is buffered in a FIFO memory. Then, the data is read out of the FIFO

at the output clock frequency and streamed out of the core. Figure 41 shows the block diagram of the core.

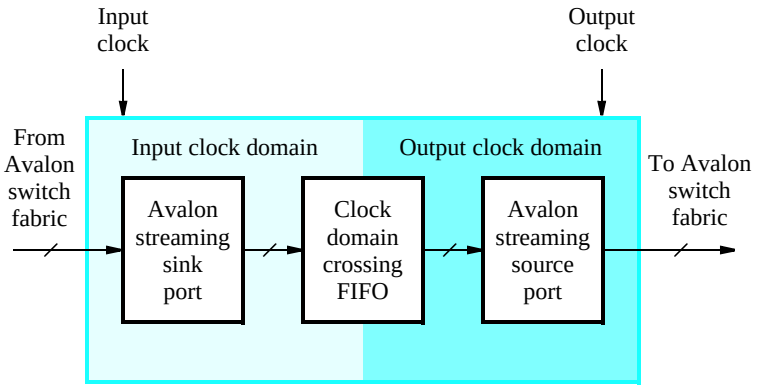


Figure 41. Dual-Clock Buffer core's block diagram.

4.8.1 Instantiating the Core in Qsys

Designers use the Dual-Clock FIFO's configuration wizard in Qsys to specify the desired features. The following configurations are available and shown in Figure 42:

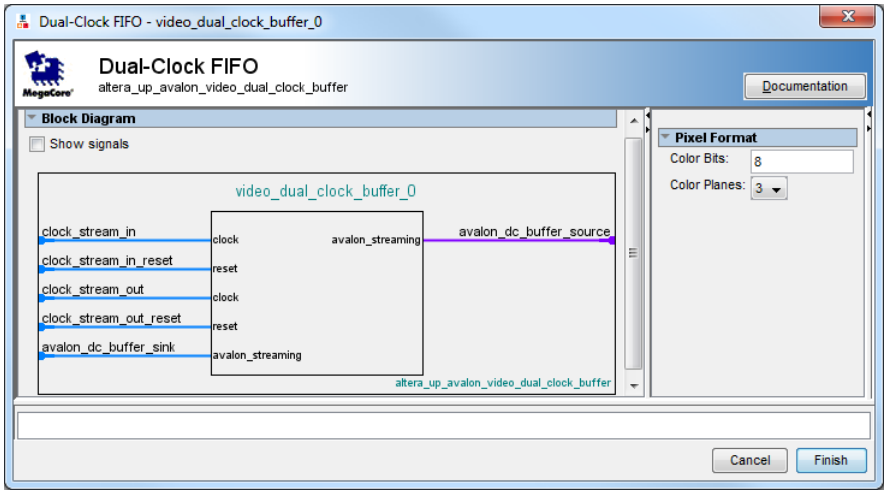


Figure 42. Dual-Clock FIFO's Qsys wizard.

- Color Bits — Specifies the number of bits per color plane
- Color Planes — Specifies the number of color planes per pixel

4.9 Edge Detection

The Edge Detection is an example of an image processing algorithm which highlights edges found in video frames. This core only accepts 8-bit grayscale formatted video. The video stream is first processed by a Gaussian smoothing filter to reduce noise in the images. Then, the stream is processed through a Sobel operator, which computes the gradient of the image intensity. Next, the stream is processed by a non-maximum suppression filter, which finds the directions of the gradients. Finally, the stream is processed through a hysteresis filter to determine which gradients are edges. Figure 43 shows the block diagram of the core.

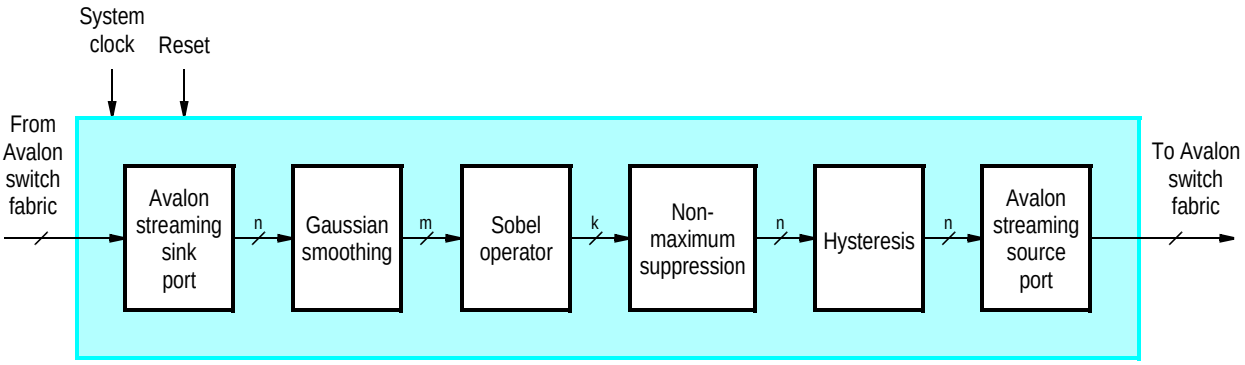


Figure 43. Edge Detection core's block diagram.

4.9.1 Instantiating the Core in Qsys

Designers use the Edge Detection's configuration wizard in Qsys to specify the desired features. The following configurations are available and shown in Figure 44:

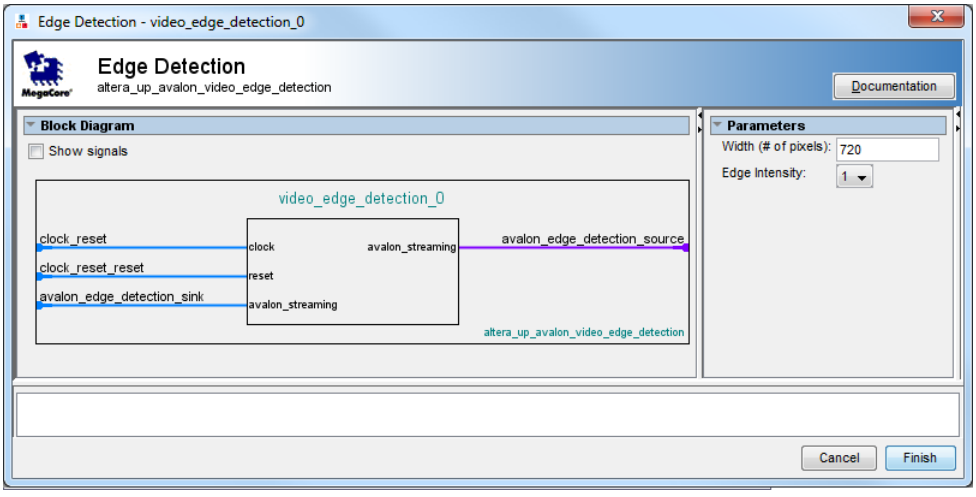


Figure 44. Edge Detection's Qsys wizard.

- Width (# of pixels) — Specifies the incoming stream's width
- Edge Intensity — A multiplicative factor for increasing the brightness of the detected edge. The 2x factor is useful for display edges on a small screen, such as the LCD with touchscreen

4.10 Pixel Buffer DMA Controller

The Pixel Buffer DMA Controller's block diagram is shown in Figure 45. The DMA controller uses its Avalon memory-mapped master interface to read video frames from an external memory. Then, it sends those video frames out via its Avalon streaming interface. The controller's Avalon memory-mapped slave interface, named *avalon_control_slave*, is used to communicate with the controller's internal registers. These internal registers and their functions are described in section 4.10.2.

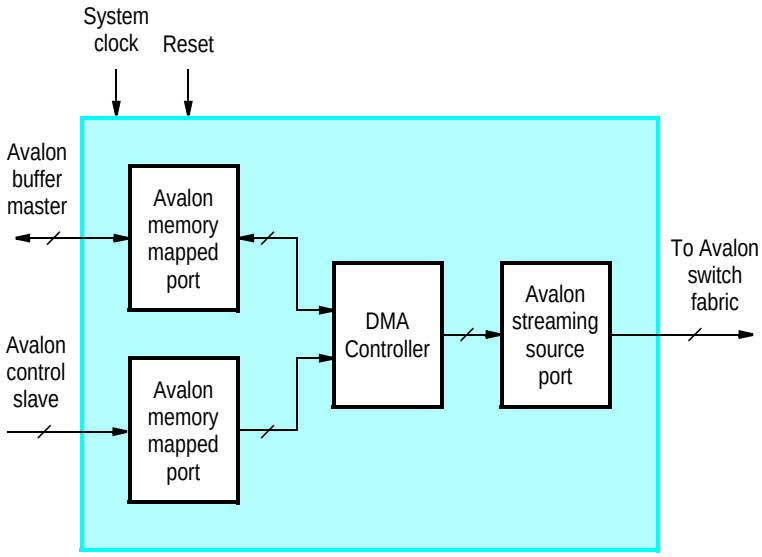


Figure 45. Pixel Buffer DMA Controller core's block diagram.

The Pixel Buffer DMA controller can used either the consecutive or X-Y addressing modes to read and write frames from and to memory. Also, the controller can store and retrieve pixels of any format and the core will automatically adjust its address for the choosen format's word length.

4.10.1 Instantiating the Core in Qsys

Designers use the Pixel Buffer DMA Controller's configuration wizard in Qsys to specify the desired features. The following configurations are available and shown in Figure 46:

- Addressing Mode — Choose between the Consecutive and the X-Y addressing modes
- Default Buffer Start Address — The start address of the buffer upon reset
- Default Back Buffer Start Address — The start address of the back buffer upon reset (can be equal to the

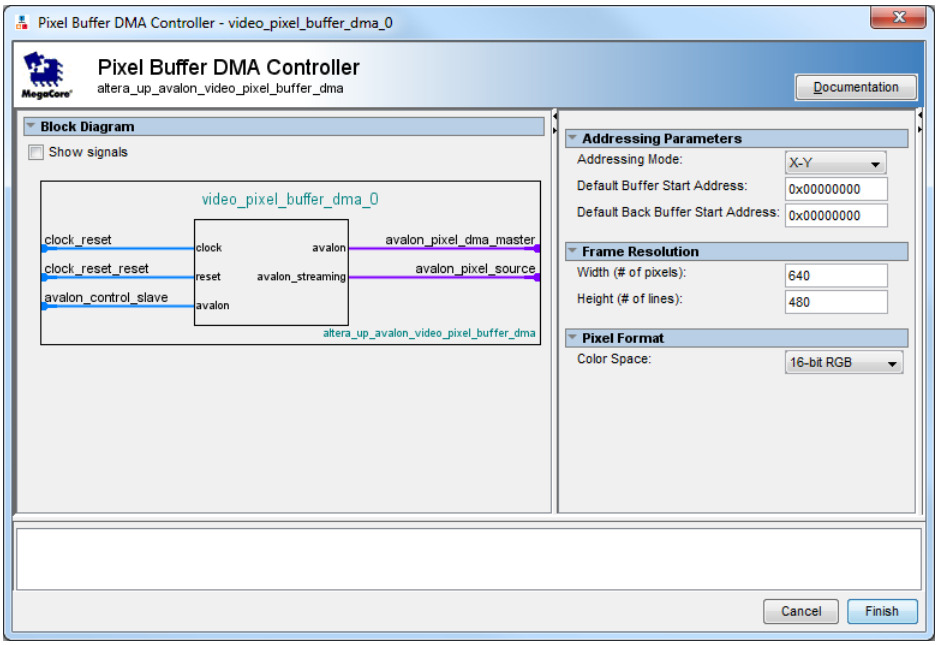


Figure 46. Pixel Buffer DMA Controller's Qsys wizard.

Default Buffer Start Address, if no back buffer is desired)

- Width (# of pixels) — Specifies the incoming stream's width
- Height (# of lines) — Specifies the incoming stream's height
- Color Space — Specifies the choosen RGB color space format

4.10.2 Software Programming Model

4.10.3 Register Map

Device drivers control and communicate with the Pixel Buffer the Avalon memory-mapped interfaces, named *avalon_control_slave*. The *avalon_control_slave* provides an interface for controlling the pixel buffer's operation, and for obtaining status information. It consists of four registers, as shown in Table 4.

The *Buffer* register holds the 32-bit address of the start of the memory buffer. This register is read-only, and shows the address of the first pixel of the frame currently being output. The *BackBuffer* register allows the start address of the frame to be changed under program control. To change the frame being displayed, the desired frame's start address is first written into the *BackBuffer* register. Then, a second write operation is performed on the *Buffer* register. The value of the data provided in this second write operation is not used by the pixel buffer. Instead, it interprets a write to the *Buffer* register as a request to *swap* the contents of the *Buffer* and *BackBuffer* registers. The swap does not occur immediately. Instead, the swap is done after the Pixel Buffer reaches the last pixel associated with the frame currently being output. While the Pixel Buffer is not yet finished outputting the current frame, bit *S* of the *Status* register will be set to 1. After the current screen is finished, the swap is performed and bit *S* is set to 0.

Table 4. Pixel Buffer register map									
Offset in bytes	**Register Name**	**R/W**	**Bit Description**						
			31...24	**23...16**	**15...8**	**7...4**	**3**	**2**	**1**	**0**
0	`Buffer`	R	Buffer's start address							
4	`BackBuffer`	R/W	Back buffer's start address							
8	`Resolution`	R	Y		X					
12	`Status`	R	m	n	*(1)*	B	*(1)*		A	S

Notes on Table 4:

(1) Reserved. Read values are undefined. Write zero.

The *Resolution* register in Table 4 provides the *X* resolution of the screen in bits 15-0, and the *Y* resolution in bits 31-16. Finally, the *Status* register provides information for the Pixel Buffer. The fields available in this register are shown in Table 5.

Table 5. Status register bits			
Bit number	**Bit name**	**R/W**	**Description**
31 - 24	m	R	Width of Y coordinate address
23 - 16	n	R	Width of X coordinate address
7 - 4	B	R	number of bytes of color: 1 (greyscale, 8-bit color), 2 (9-bit and 16-bit color), 3 (24-bit color) or 4 (30-bit and 32-bit color)
1	A	R	Addressing mode: 0 (X,Y), or 1 (consecutive)
0	S	R	Swap: 0 when swap is done, else 1

4.10.4 Programming with the Pixel Buffer

The Pixel Buffer is packaged with C-language functions accessible through the hardware abstraction layer (HAL). These functions implement the basic operations that are needed for the Pixel Buffer. An example of C code that these functions is given at the end of this section.

To use the functions, the C code must include the statement:

```
#include "altera_up_avalon_pixel_buffer.h"
```

alt_up_pixel_buffer_open_dev

Prototype: `alt_up_pixel_buffer_dev* alt_up_pixel_buffer_open_dev(const char *name)`
Include: `<altera_up_avalon_pixel_buffer.h>`
Parameters: `name` – the pixel buffer component name in Qsys.
Returns: The corresponding device structure, or NULL if the device is not found
Description: Opens the pixel buffer device specified by *name* .

alt_up_pixel_buffer_draw

Prototype: `int alt_up_pixel_buffer_draw(alt_up_pixel_buffer_dev *pixel_buffer, unsigned int color, unsigned int x, unsigned int y)`
Include: `<altera_up_avalon_pixel_buffer.h>`
Parameters: `pixel_buffer` – the pointer to the VGA structure
`color` – the RGB color to be drawn
`x` – the *x* coordinate
`y` – the *y* coordinate
Returns: 0 for success, -1 for error (such as out of bounds)
Description: Draw a pixel at the location specified by *(x, y)* on the VGA monitor.

alt_up_pixel_buffer_change_back_buffer_address

Prototype: `int alt_up_pixel_buffer_change_back_buffer_address(alt_up_pixel_buffer_dev *pixel_buffer, unsigned int new_address)`
Include: `<altera_up_avalon_pixel_buffer.h>`
Parameters: `pixel_buffer` – the pointer to the VGA structure
`new_address` – the new start address of the back buffer
Returns: 0 for success
Description: Changes the back buffer's start address.

alt_up_pixel_buffer_swap_buffers

Prototype: `int alt_up_pixel_buffer_swap_buffers(alt_up_pixel_buffer_dev *pixel_buffer)`
Include: `<altera_up_avalon_pixel_buffer.h>`
Parameters: `pixel_buffer` – the pointer to the VGA structure
Returns: 0 for success
Description: Swaps which buffer is being sent to the VGA Controller.

alt_up_pixel_buffer_check_swap_buffers_status

Prototype: `int alt_up_pixel_buffer_check_swap_buffers_status(alt_up_pixel_buffer_dev *pixel_buffer)`
Include: `<altera_up_avalon_pixel_buffer.h>`
Parameters: `pixel_buffer` – the pointer to the VGA structure
Returns: 0 if complete, 1 if still processing
Description: Check if swapping buffers has completed.

alt_up_pixel_buffer_clear_screen

Prototype: `void alt_up_pixel_buffer_clear_screen(alt_up_pixel_buffer_dev *pixel_buffer, int backbuffer)`
Include: `<altera_up_avalon_pixel_buffer.h>`
Parameters: `pixel_buffer` – the pointer to the VGA structure
`backbuffer` – set to 1 to clear the back buffer, otherwise set to 0 to clear the current screen.
Returns: 0 if complete, 1 if still processing
Description: This function clears the screen or the back buffer.

alt_up_pixel_buffer_draw_box

Prototype: `void alt_up_pixel_buffer_draw_box(alt_up_pixel_buffer_dev *pixel_buffer, int x0, int y0, int x1, int y1, int color, int backbuffer)`
Include: `<altera_up_avalon_pixel_buffer.h>`
Parameters: `pixel_buffer` – the pointer to the VGA structure
`x0`, `x1`, `y0`, `y1` – coordinates of the top left (x0,y0) and bottom right (x1,y1) corner of the box
`color` – color of the box to be drawn
`backbuffer` – set to 1 to select the back buffer, otherwise set to 0 to select the current screen.
Returns: 0 if complete, 1 if still processing
Description: This function draws a box of a given color between points (x0,y0) and (x1,y1).

alt_up_pixel_buffer_draw_hline

Prototype: `void alt_up_pixel_buffer_draw_hline(alt_up_pixel_buffer_dev *pixel_buffer, int x0, int x1, int y, int color, int backbuffer)`
Include: `<altera_up_avalon_pixel_buffer.h>`
Parameters: `pixel_buffer` – the pointer to the VGA structure
`x0`, `x1`, `y` – coordinates of the left (x0,y) and the right (x1,y) end-points of the line
`color` – color of the line to be drawn
`backbuffer` – set to 1 to select the back buffer, otherwise set to 0 to select the current screen.
Returns: 0 if complete, 1 if still processing
Description: This function draws a horizontal line of a given color between points (x0,y) and (x1,y).

alt_up_pixel_buffer_draw_vline

Prototype: `void alt_up_pixel_buffer_draw_vline(alt_up_pixel_buffer_dev *pixel_buffer, int x, int y0, int y1, int color, int backbuffer)`
Include: `<altera_up_avalon_pixel_buffer.h>`
Parameters: `pixel_buffer` – the pointer to the VGA structure
`x`, `y0`, `y1` – coordinates of the top (x,y0) and the bottom (x,y1) end-points of the line
`color` – color of the line to be drawn
`backbuffer` – set to 1 to select the back buffer, otherwise set to 0 to select the current screen.
Returns: 0 if complete, 1 if still processing
Description: This function draws a vertical line of a given color between points (x,y0) and (x,y1).

alt_up_pixel_buffer_draw_rectangle

Prototype: `void alt_up_pixel_buffer_draw_rectangle(alt_up_pixel_buffer_dev *pixel_buffer, int x0, int y0, int x1, int y1, int color, int backbuffer)`
Include: `<altera_up_avalon_pixel_buffer.h>`
Parameters: `pixel_buffer` – the pointer to the VGA structure
`x0`, `x1`, `y0`, `y1` – coordinates of the top left (x0,y0) and bottom right (x1,y1) corner of the rectangle
`color` – color of the rectangle to be drawn
`backbuffer` – set to 1 to select the back buffer, otherwise set to 0 to select the current screen.
Returns: 0 if complete, 1 if still processing
Description: This function draws a rectangle of a given color between points (x0,y0) and (x1,y1).

alt_up_pixel_buffer_draw_line

Prototype:	`void alt_up_pixel_buffer_draw_line(alt_up_pixel_buffer_dev *pixel_buffer, int x0, int y0, int x1, int y1, int color, int backbuffer)`
Include:	`<altera_up_avalon_pixel_buffer.h>`
Parameters:	`pixel_buffer` – the pointer to the VGA structure `x0`, `x1`, `y0`, `y1` – coordinates (x0,y0) and (x1,y1) correspond to end points of the line `color` – color of the line to be drawn `backbuffer` – set to 1 to select the back buffer, otherwise set to 0 to select the current screen.
Returns:	0 if complete, 1 if still processing
Description:	This function draws a line of a given color between points (x0,y0) and (x1,y1).

4.10.5 Pixel Buffer core C Example using Device Drivers

The example program using HAL for the Pixel Buffer DMA Controller is shown in Figure 47.

```
#include "altera_up_avalon_pixel_buffer.h"

int main(void)
{
	alt_up_pixel_buffer_dev * pixel_buf_dev;

	// open the Pixel Buffer port
	pixel_buf_dev = alt_up_pixel_buffer_open_dev ("/dev/Pixel_Buffer");
	if ( pixel_buf_dev == NULL)
		alt_printf ("Error: could not open pixel buffer device \n");
	else
		alt_printf ("Opened pixel buffer device \n");

	/* Clear and draw a blue box on the screen */
	alt_up_pixel_buffer_clear_screen (pixel_buf_dev);
	alt_up_pixel_buffer_draw_box (pixel_buf_dev, 0, 0, 319, 239, 0x001F, 0);
}
```

Figure 47. An example of C with Device Driver Support code that uses Pixel Buffer Core.

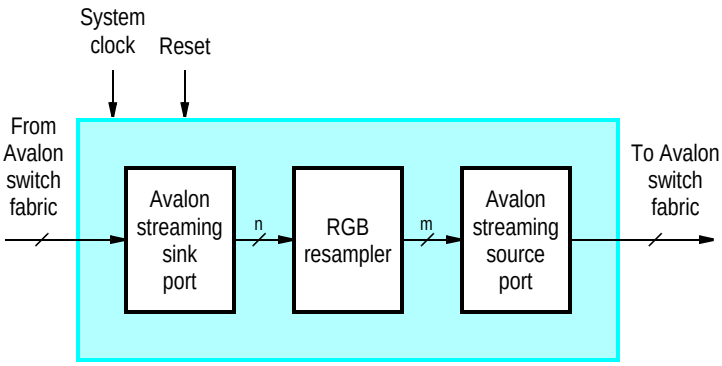


Figure 48. RGB Resmapler core's Block Diagram

4.11 RGB Resampler

The RGB Resampler converts video streams between the RGB color space formats. The core can convert between any RGB format, except the Bayer pattern format. Figure 48 shows a block diagram of the core.

Although the core can convert to any RGB format, converting to the 8-bit Grayscale format should be avoided. The manner used to convert to the grayscale format is very rudimentary. To acheive a better quality conversion, use the Color Space Converter IP core.

4.11.1 Instantiating the Core in Qsys

Designers use the RGB Resampler's configuration wizard in Qsys to specify the desired features. The following configurations are available and shown in Figure 49:

- Incoming Format — Specifies the RGB format of the incoming stream
- Outgoing Format — Specifies the desired RGB format for the outgoing stream
- Alpha Value for Output — Specifies the desired alpha value for the output stream, when required and not present in the input stream

Note: the input and output formats cannot be the same.

4.12 Scaler

The Scaler IP core modifies the resolution of video stream. The scaler converts an incoming video stream's resolution by adding or dropping entire rows and/or columns of pixels. Figure 50 shows a block diagram of the core.

The scaler adds pixels by replicating columns and/or rows. For example, if the scaler is increasing a stream's width by factor of 2, it will output every incoming pixel twice. Another example, is when a stream's height is to be increased by a factor of 4, the scaler will buffer each row and output it four times before continuing on to the next row.

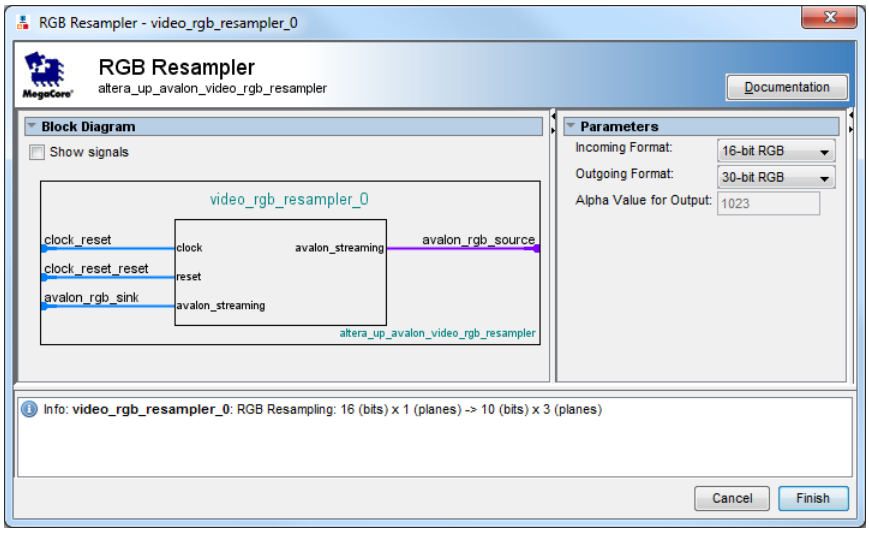


Figure 49. RGB Resampler's Qsys wizard.

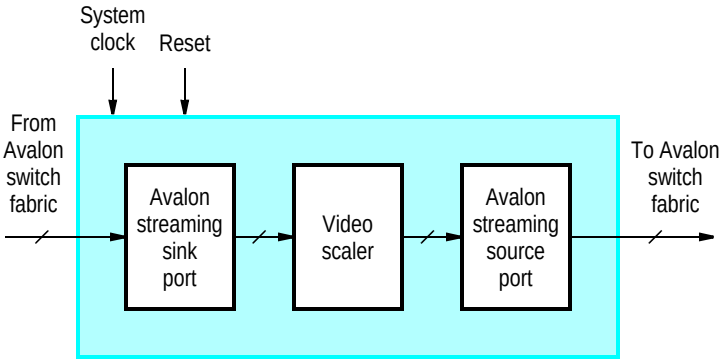


Figure 50. Scaler core's block diagram.

The scaler reduces a stream's resolution by removing entire columns and/or rows. For example, if the scaler is decreasing a stream's width by factor of 4, it will output one incoming pixel, drop the next three pixels, and continue repeating this pattern for the entire frame. When a stream's height is to be decreased by a factor of 2, the scaler will output one row and drop next, again repeating for the entire frame.

4.12.1 Instantiating the Core in Qsys

Designers use the Scaler's configuration wizard in Qsys to specify the desired features. The following configurations are available and shown in Figure 51:

- Scaling Parameters
  - Width Scaling Factor — Specifies the scaling factor for the video stream's width

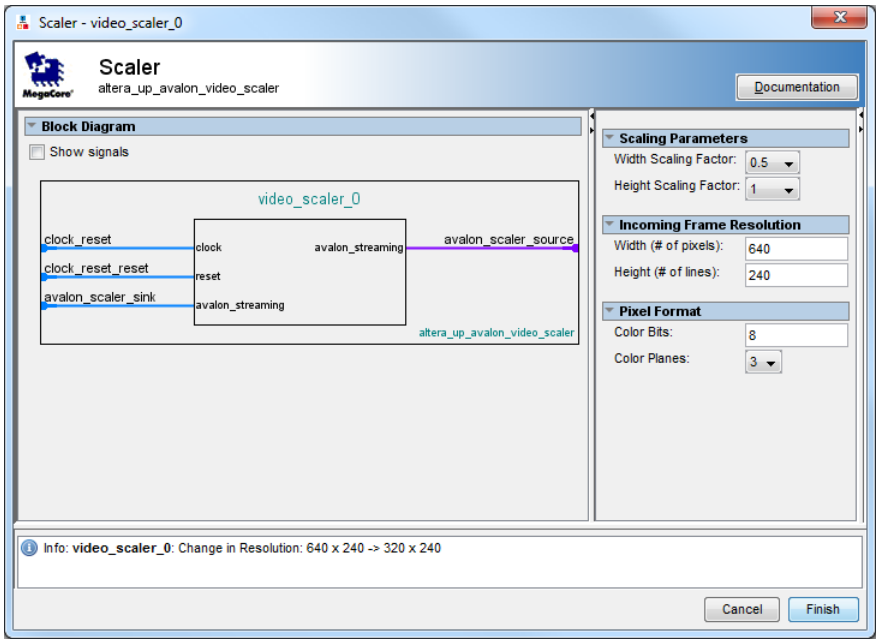


Figure 51. Scaler's Qsys wizard.

  - Height Scaling Factor— Specifies the scaling factor for the video stream's height

- Incoming Frame Resolution
  - Width (# of pixels) — Specifies the incoming stream's width
  - Height (# of lines) — Specifies the incoming stream's height

- Pixel Format
  - Color Bits — Specifies the number of bits per color plane
  - Color Planes — Specifies the number of color planes

4.13 Test-Pattern Generator

The Test-Pattern Generator IP core generates a video stream. The core generates a constant image in the 24-bit RGB format, which is outputted via its Avalon Streaming interface. The constant image is generated by changing two values in the hue-saturation-value (HSV) color space. On the x-axis, the hue changes from 0° to 360°. On the y-axis, the saturation changes from 0 to 1. Figure 52 shows a block diagram of the core.

4.13.1 Instantiating the Core in Qsys

Designers use the Test-Pattern Generator's configuration wizard in Qsys to specify the desired features. The following configurations are available and shown in Figure 53:

- Outgoing Frame Resolution

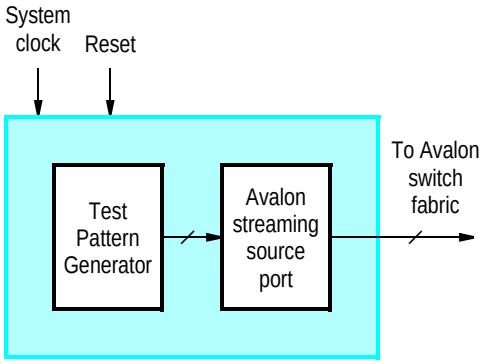


Figure 52. Test Pattern Generator core's block diagram.

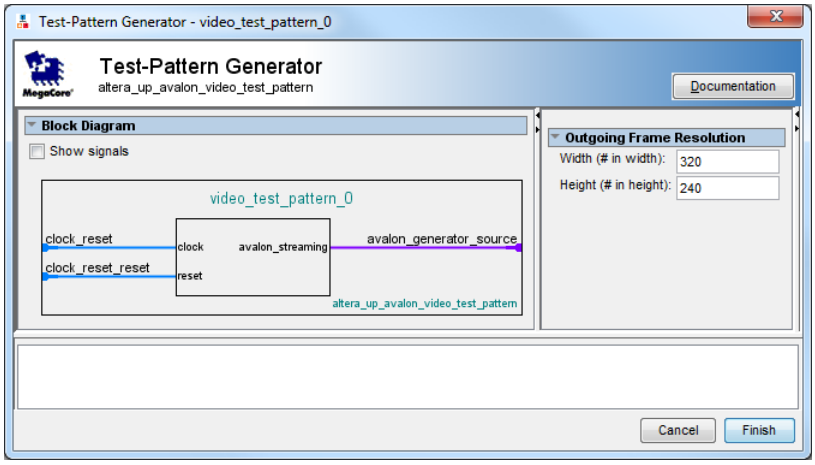


Figure 53. Test-Pattern Generator's Qsys wizard.

- Width (# of pixels) — Specifies the outgoing stream's width
- Height (# of lines) — Specifies the outgoing stream's height

4.14 VGA Controller

The VGA controller IP core generates the timing signals required by the on-board VGA DAC on the DE-series boards and Terasic's LCD with touchscreen daughtercards. Data is provided to the VGA Controller via its Avalon Streaming Interface. The controller takes the incoming data, adds the appropriate VGA timing signals and then sends that information to either the on-board VGA DAC or the LCD with touchscreen daughtercard. Figure 52 shows a block diagram of the core.

The VGA Controller generates the timing signals required for the VGA DAC and LCD daughtercard, including horizontal and vertical synchronization signals. The timing information generated by the VGA Controller produces screen resolutions of 640×480, 800×480 and 800×600 pixels for the VGA DAC, the LCD with touchscreen

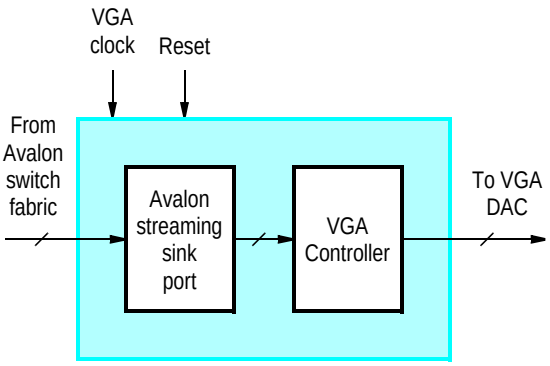


Figure 54. VGA Controller core's block diagram.

(TRDB_LTM) and the 8 inch LCD on the tPad, respectively. To generate the timing information correctly, a 25 MHz clock has to be provided to the VGA Controller, except for the 8 inch LCD, when a 40 MHz clock must be provided instead. The ***Clock Signals for DE-series Boards*** core, also provided by the Altera University Program, can generate the required 25 MHz and 40 MHz clocks; see its documentation for more details.

4.14.1 Instantiating the Core in Qsys

Designers use the VGA Controller's configuration wizard in Qsys to specify the desired features. The following configurations are available and shown in Figure 55:

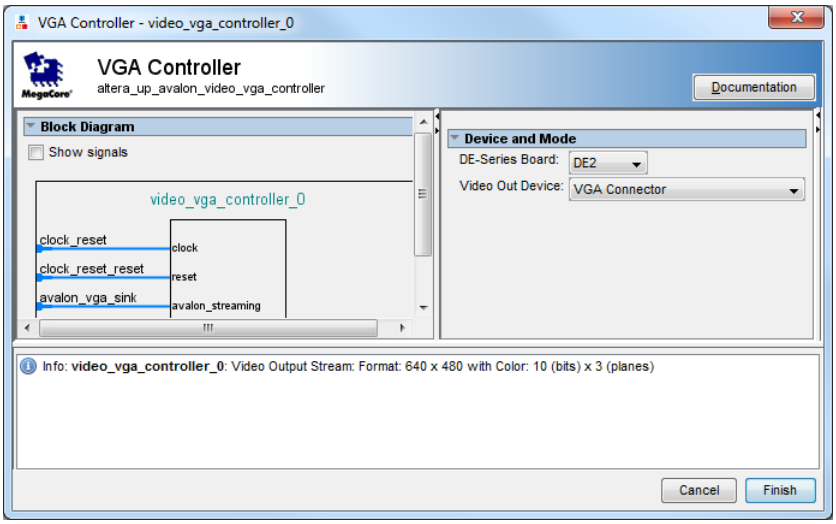


Figure 55. VGA Controller's Qsys wizard.

- DE-Series Board — Specifies the Altera DE-series board that the system is being designed for
- Video Out Device — Specifies the VGA compatible device being used, and by extension the screen resolution

4.15 Video-In Decoder

The Video-In decoder core, as shown in Figure 1, takes video input from one of three video sources: composite-video port on DE2/DE2-70 board, a 1.3 Megapixel CCD camera, or a 5 Megapixel CCD camera. As shown on the left-hand side figure, the video data enters the decoder from the video source. It is than decoded, and syncronized from the video source's clock domain to the system's clock domain. In the final stage of the core, the video data is converted into Altera University Program video streaming packets.

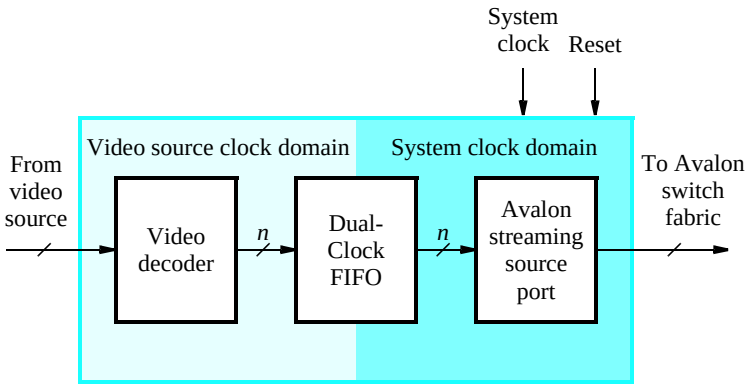


Figure 56. Video-In Decoder Core's Block Diagram

Each outgoing packet represents a video frame. The size of the packet and the color format used depends on the video source, as listed in Table 6. The on-board video source supplies data from either an NTSC or PAL device in a 4:2:2 YCrCb format. The CCD cameras provide video data in the Bayer pattern format.

The Video-In Decoder's Avalon streaming source port should be connected to the Avalon streaming sink port of the next video processing core using Qsys. In typical video processing flows, the next core should be either the Chroma Resampler, in the case of the on-board video source, or the Scaler, in the case of a CCD camera source.

Table 6. Video In Decoder Packet Formats

Input Device	Color Format	Video Packet Size
On-Board Video (NTSC)	YUV 4:2:2	720 × 244
On-Board Video (PAL)	YUV 4:2:2	720 × 288
1.3 Megapixel Camera (TRDB_DC2)	Bayer Pattern	1280 × 1024
5 Megapixel Camera (TRDB_D5M)	Bayer Pattern	2592 × 1944

When using the Video-In Decoder IP core, users should also include the Audio and Video Configuration IP core in their system. The configuration core can initialize the video input device for use with the decoder core.

4.15.1 Instantiating the Core in Qsys

Designers use the Video-In Decoder's configuration wizard in Qsys to specify the desired features. The following configurations are available and shown in Figure 57:

- Video-In Source — Choose the source being used, and by extension the video format outputted by this core

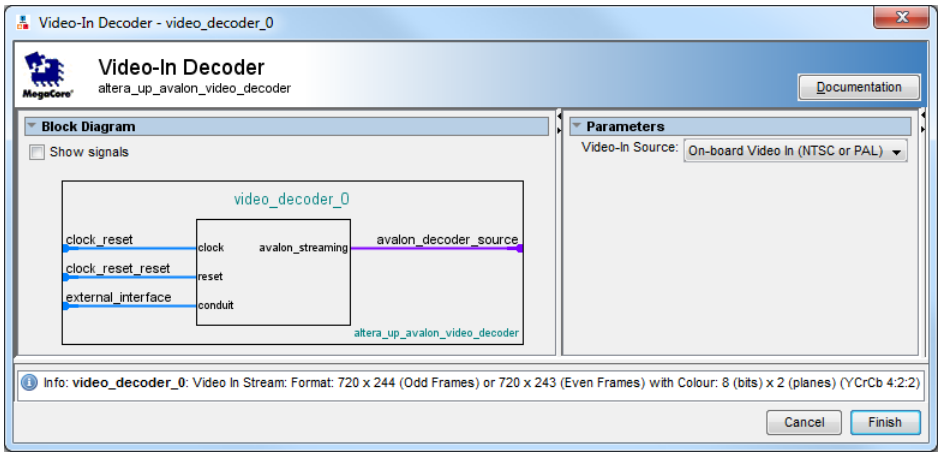


Figure 57. Video-In Decoder's Qsys wizard.

4.16 Video-Stream Router

The Video-Stream Router IP core allows users to select alternate paths for video streams. The router has two modes: split and merge. In split mode, the incoming stream will be routed to one of the outgoing stream, based upon the value of the external bit, *path selector*, as shown in Figure 58. In merge mode, one of the two incoming streams will be routed to the outgoing stream, based upon the value of the external bit, *path selector*, as shown in Figure 59.

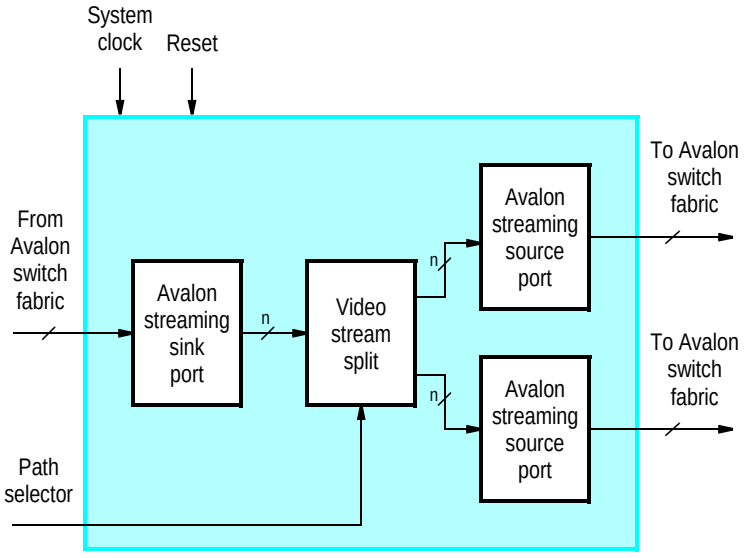


Figure 58. Splitting stream router's block diagram.

When using two routers in tandem, one in split mode and one in merge mode, using two separate path select signals may cause instability in the system. This instability is due to the possibility that the two cores could get stuck in a

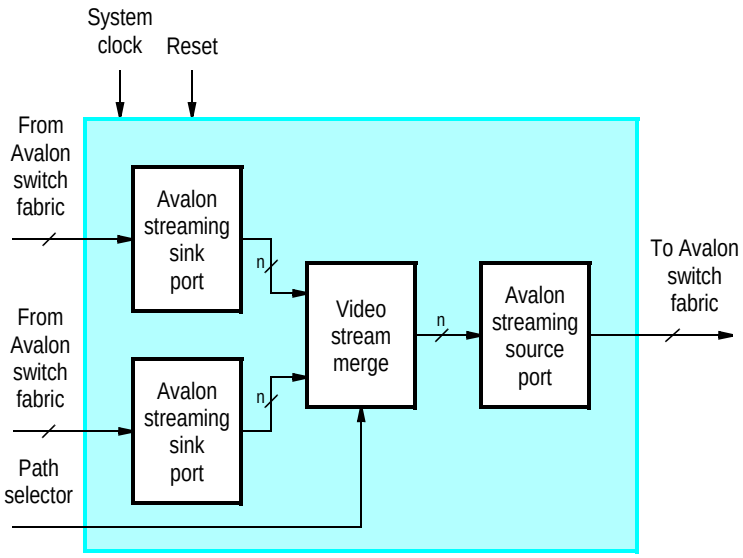


Figure 59. Merging stream router's block diagram.

state where they are trying to route the stream along opposite paths. To alleviate this issue, the cores should include the optional Avalon streaming interface used for synchronization. When using the synchronization interface, the split-mode router will control the path used by both cores. As a result, the merge-mode router will no longer include the external path select signal.

4.16.1 Instantiating the Core in Qsys

Designers use the Video-Stream Router's configuration wizard in Qsys to specify the desired features. The following configurations are available and shown in Figure 60:

- Mode
  - Stream Router Type — Specifies the type of router, either split or merge
  - Enable Synchronization Stream — Enables the synchronization stream interface
- Pixel Format
  - Color Bits — Specifies the number of bits per color plane
  - Color Planes — Specifies the number of color planes per pixel

4.17 VIP Bridges

The VIP Bridge IP cores convert video streams between the Altera VIP format and UP video stream format. There are two bridge cores, named "Raw to VIP" and "VIP to Raw".

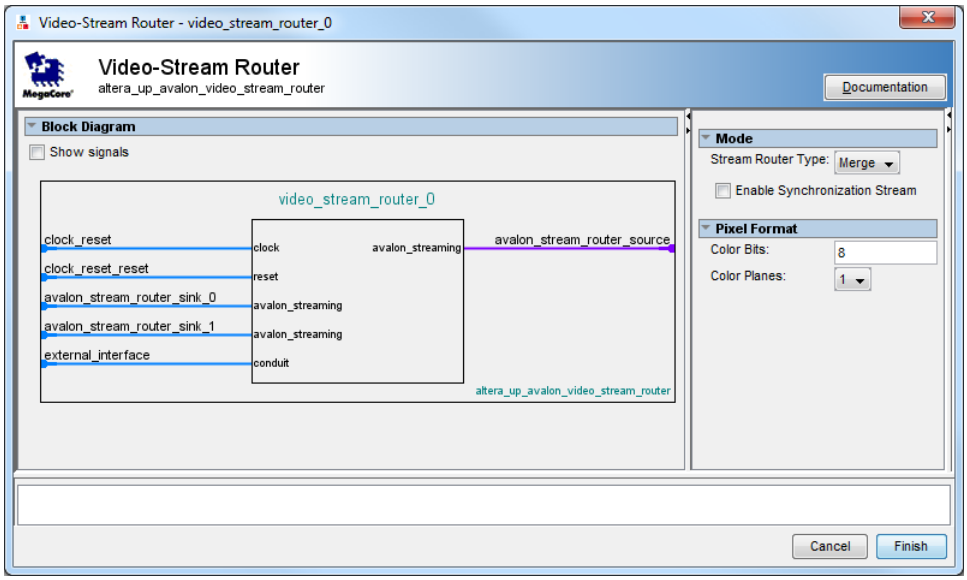


Figure 60. Video-Stream Router's Qsys wizard.

4.17.1 Raw to VIP

The Raw-to-VIP bridge take an input stream in the University Program (UP) video stream format and converts it to the VIP format. Then, the VIP-formatted data is outputted via the Avalon streaming source interface. Figure 61 shows a block diagram of the core. The UP video stream format is essentially raw video data, as described in Section 3.2.

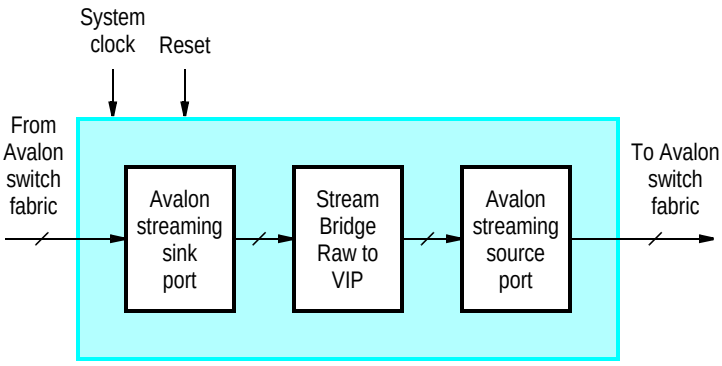


Figure 61. Video-Stream Bridge: Raw to VIP core's block diagram.

Instantiating the Core in Qsys

Designers use the Video-Stream Bridge: RAW to VIP's configuration wizard in Qsys to specify the desired features. The following configurations are available and shown in Figure 62:

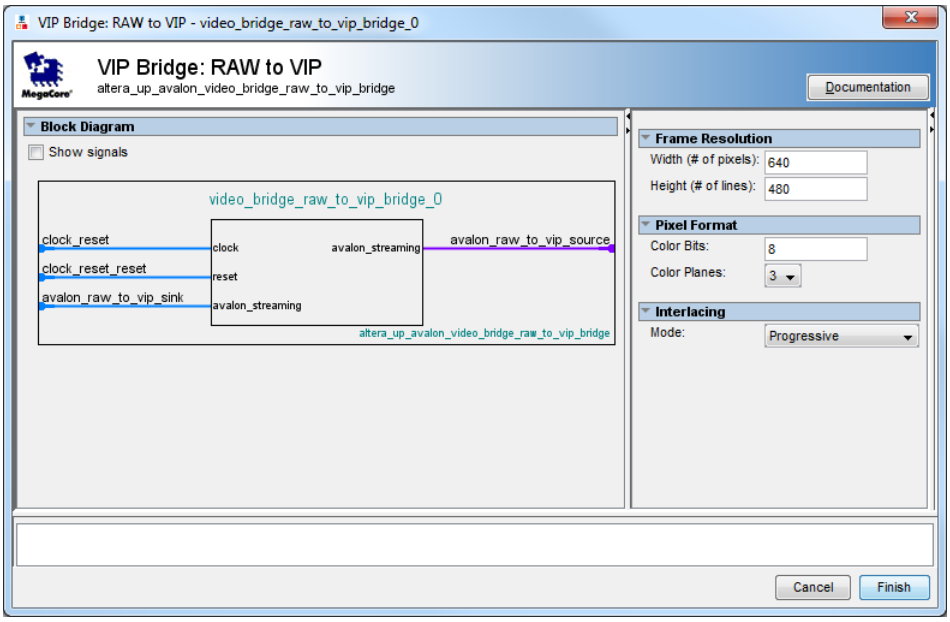


Figure 62. Video-Stream Bridge: Raw to VIP core's Qsys wizard.

- Frame Resolution
  - Width (# of pixels) — Specifies the incoming stream's width
  - Height (# of lines) — Specifies the incoming stream's height
- Pixel Format
  - Color Bits — Specifies the number of bits per color plane
  - Color Planes — Specifies the number of color planes per pixel
- Interlacing
  - Mode — Specifies the ing0ing stream's scan type

4.17.2 VIP to Raw

The VIP-to-Raw bridge takes an input stream in the VIP format, and converts it to the UP video-stream format. Then, the UP video-stream-formatted data is outputted via the Avalon streaming source interface. Figure 63 shows a block diagram of the core.

Instantiating the Core in Qsys

Designers use the Video-Stream Bridge: VIP to Raw's configuration wizard in Qsys to specify the desired features. The following configurations are available and shown in Figure 64:

- Pixel Format

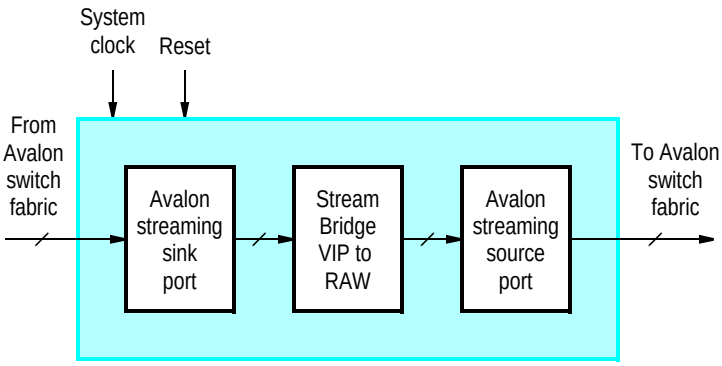


Figure 63. Video-Stream Bridge: VIP to RAW core's block diagram.

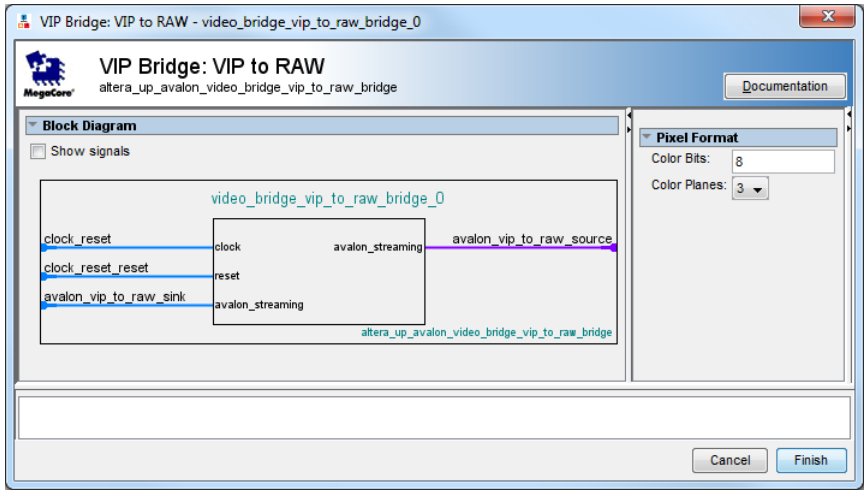


Figure 64. Video-Stream Bridge: VIP to RAW core's Qsys wizard.

- Color Bits — Specifies the number of bits per color plane
- Color Planes — Specifies the number of color planes per pixel