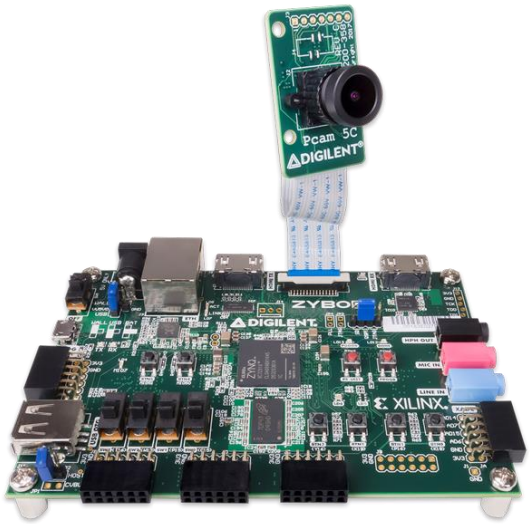
Embedded Vision Demo

Date: August 8, 2019
Author: Thomas Kappenman

1300 NE Henley Court, Suite 3
Pullman, WA 99163
(509) 334 6306 Voice | (509) 334 6300 Fax



Overview

This project demonstrates the use of the Digilent Embedded Vision Bundle, Xilinx Vivado, Xilinx Vivado HLS, and Xilinx SDK to create a video processing system. The Digilent Embedded Vision bundle includes the Zybo Z7-20 Zynq-7000 ARM/FPGA SoC development board and the Pcam 5C 5-megapixel (MP) color camera module.

The Digilent ZYBO Z7 is the newest addition to the popular ZYBO line of ARM/FPGA SoC Platforms. The ZYBO Z7 surrounds the Zynq with a rich set of multimedia and connectivity peripherals to create a formidable single-board computer, even before considering the flexibility and power added by the FPGA. Hardware designers and software developers can seamlessly integrate FPGA and CPU functionality. The ZYBO Z7's video-capable feature set includes a MIPI CSI-2 compatible Pcam connector, HDMI input, HDMI output, and high DDR3L bandwidth.

The Pcam 5C is an imaging module meant for use with FPGA development boards. The module is designed around the Omnivision OV5640 5MP color image sensor. This sensor includes various internal processing functions that can improve image quality, including automatic white balance, automatic black level calibration, and controls for adjusting saturation, hue, gamma and sharpness. Data is transferred over a dual-lane MIPI CSI-2 interface, which provides enough data bandwidth to support common video streaming formats such as 720p (at 60 frames per second) and 1080p (at 30 frames per second). The module is connected to the FPGA development board via a 15-pin flat-flexible cable (FFC).

In this demo, a Pcam 5C supplies real-time high definition video to the HDMI input of a Zybo Z7-20. The Zybo can then perform edge detection, inverted color or grayscale by configuring bottom two switches (sw1 and sw0) on the Zybo Z7-20. The processed signals are sent to the HDMI output of the Zybo Z7-20 for display on an HD monitor. A video of the embedded vision demo can be found at https://youtu.be/vBDIxdp-q8A.

The source files and IP can be downloaded at https://s3-us-west-2.amazonaws.com/digilent-file-share-public/Zybo_Z7_Embedded_Vision_Demo.zip. This project requires Vivado 2017.4 to open.

Required Materials

- Digilent Zybo Z7-20 from Embedded Vision Bundle
- Digilent Pcam 5C from Embedded Vision Bundle
- 15-pin flat-flexible cable included with Pcam 5C
- 720p compatible HDMI Screen
- 5V power supply or USB cable
- MicroSD Card (fat32)

Hardware Setup

1. Copy BOOT.bin to the root of your micro SD card. Insert the SD card into the Zybo Z7 micro SD card slot.
2. Connect an HDMI cable from the HDMI TX port on the Zybo Z7 to your monitor.
3. Attach the Pcam 5C to the 15-pin flat-flexible cable. Connect this cable to J2 on the Zybo Z7. The contact side of the cable should face away from the white tab of the connector.
4. Set JP5, the boot mode jumper, to SD.
5. Connect the power supply to the Zybo Z7. Set the power switch to ON.

Vivado HLS & Data Pipelining

The video processing IP blocks used in this demo are created using Vivado HLS. The role of these high-level synthesis tools is to extract the best possible circuit implementation from a C/C++ code that is functionally correct and meets the requirements. It analyzes data dependencies determining which operations could and should execute in each clock cycle. Depending on the targeted clock frequency and FPGA, some operations might take more cycles to complete. This step is called **scheduling**. Next, the hardware resources are determined to implement the scheduled operation best. This is called **binding.** The last step in the synthesis is the **control logic extraction** which creates a finite state machine that controls when the different operations should execute in the design.

An example of optimization performed by the tool for multi-cycle operations is **pipelining**. Imagine the following C statement:

```
x=a*b+c;
```

If the clock period is too small for the multiplication and addition to complete in one clock cycle, it will be scheduled for two cycles. For every set of inputs a, b, and c it takes two cycles to obtain the result. It follows that in cycle 2 the multiplier does not perform any operation; it only provides the result calculated in the previous cycle.

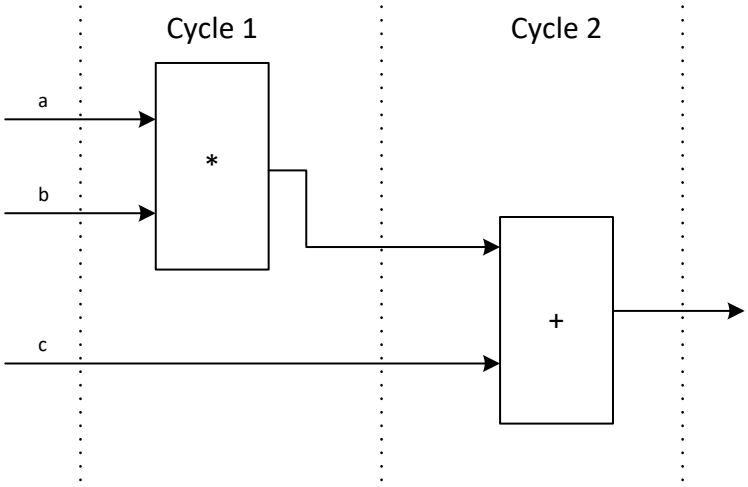


Figure 1: Sequential Execution

This inefficiency becomes more apparent, when this statement is executed in a loop, ie. the circuit processes more than one set of input data.
If there was a storage element between cycles, the result from cycle 1 would be saved, and the multiplier would be free to perform a calculation for the next set of inputs. This concept is called pipelining and it is a major optimization opportunity increasing the throughput tremendously.

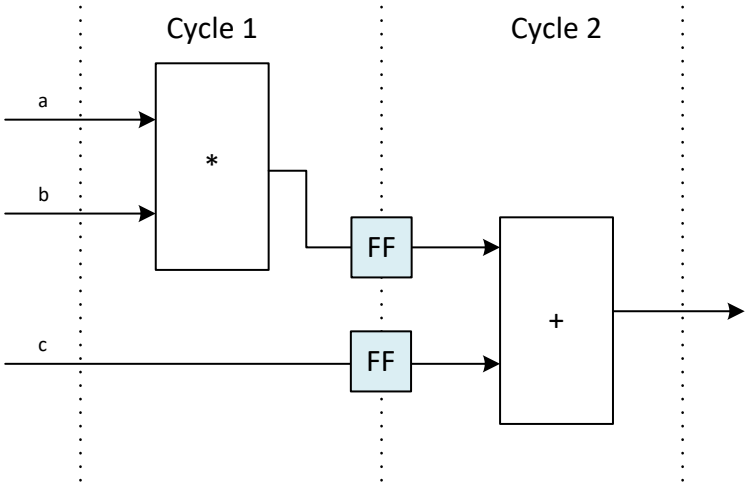


Figure 2: Pipelined Execution

Operation

This demo covers the entire process of capturing video data, processing it, and outputting it. A simplified diagram of the video pipeline is shown below.

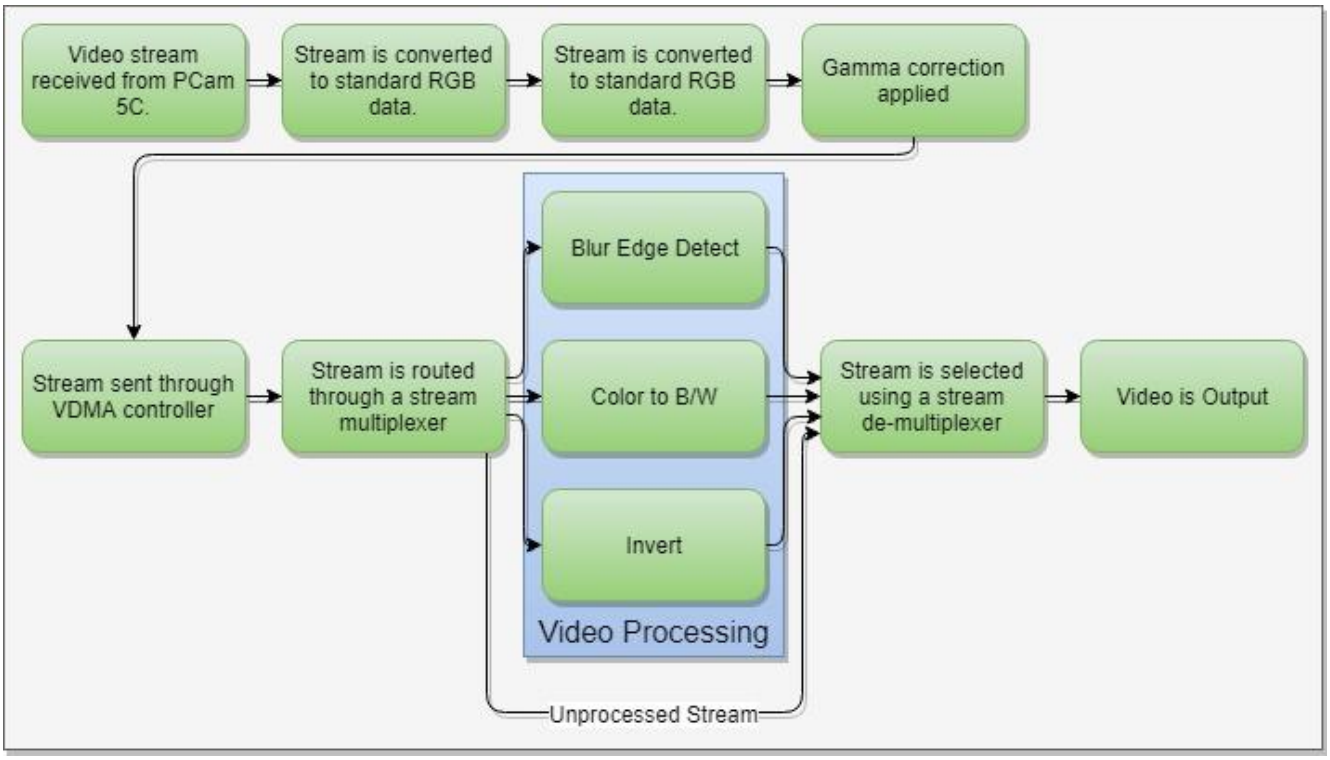


Figure 3: Video Pipeline Operation Flowchart

Video Pipeline IP Cores

The IP cores discussed in this section are used in Vivado's IP Integrator (IPI). The IP cores designed by Digilent are open-source and are available for download on Digilent's official Github.

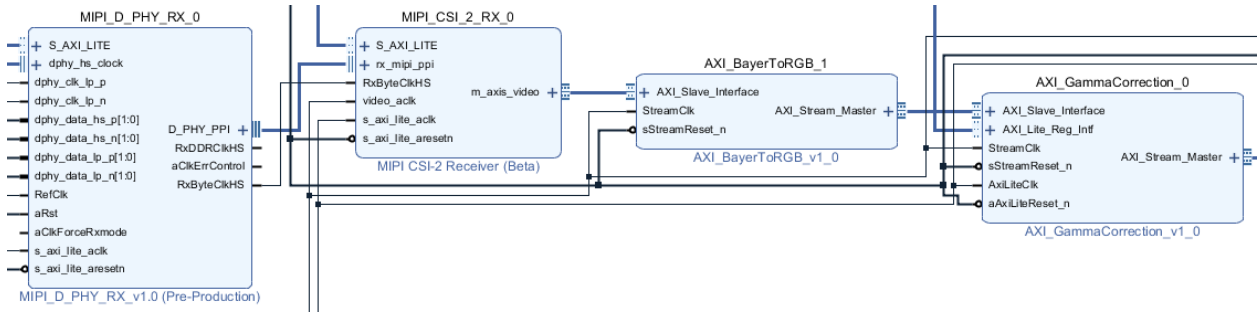


Figure 4: MIPI RX and Preprocessing

- **MIPI_D_PHY_RX** – Implements the physical layer of the interface, MIPI D-PHY. Data is de-serialized and the two data lanes merged together.
- **MIPI_CSI_2_RX** – Implements the protocol layer of the interface, MIPI Camera Serial Interface 2 (CSI-2). It interprets data words as pixels, lines, and frames. Video data is formatted as RAW RGB data and packed into an AXI-Stream interface forming the input side of the video processing pipeline.
- **AXI_BayerToRGB** – Interpolates the Bayer-filtered RAW RGB data into standard RGB data (32 bits: 2 bit padding, 30 bits red, green, and blue).
- **AXI_GammaCorrection** – Applies gamma correction to the video stream while also reducing the RGB color depth to 8 bits per color (24 bits: 8 red, 8 green, 8 blue). This block applies one of five gamma factors to the incoming image: 1, 1/1.2, 1/1.5, 1/1.8, and 1/2.2.

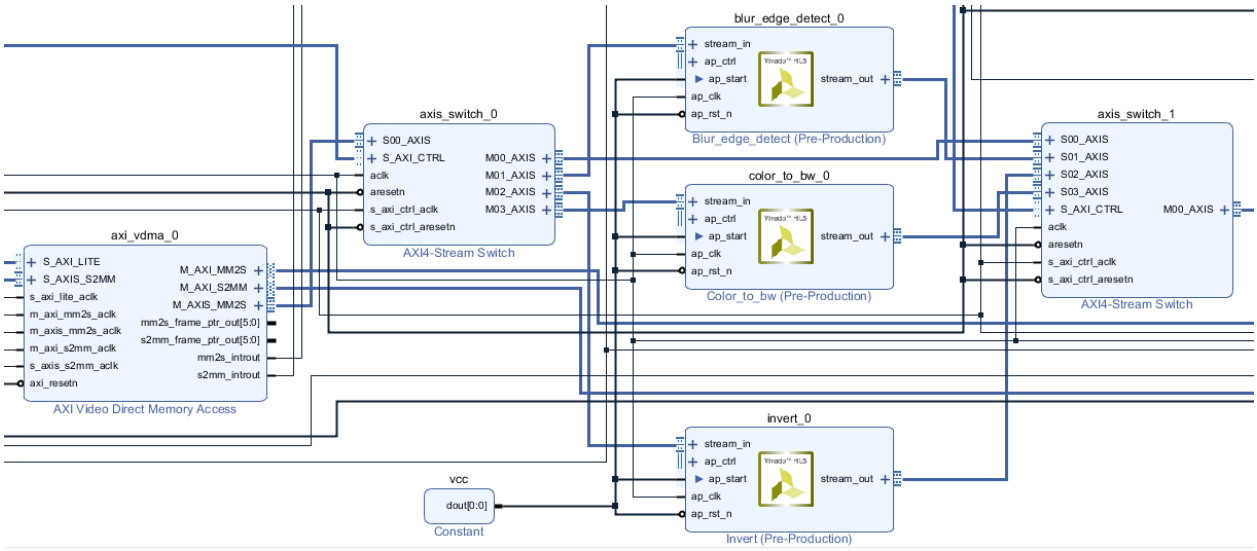


Figure 5: Video Stream Mux/Demux

- **AXI_Video Direct Memory Access** – Implements three frame buffers in DDR memory allowing access from within the processor, if needed. This block is connected to the Zynq memory subsystem via two high performance memory access ports. Doubling the input frame rate of 30Hz to the output frame rate of 60Hz is achieved with frame buffering.
- **AXI4-Stream Switch 0** – Muxes the stream to one of 4 locations:
- **Blur edge detect** – A HLS IP core designed to show edge detection. This core first applies a gaussian blur filter onto the image to get rid of noise. It then applies a Sobel filter on the X and Y direction and combines these images.
- **Color to BW** – A HLS IP core that converts RGB data to grayscale.
- **Invert** – A HLS IP core that inverts the colors.
- **AXI4-Stream Switch 1** – De-muxes the video stream to one source.

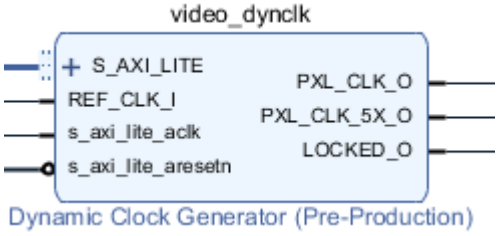


Figure 6: Dynamic Clock Generator

- **Dynamic Clock Generator** – Generates the pixel clock used by the HDMI output. The pixel clock frequency changes based on the output resolution set by the processor.

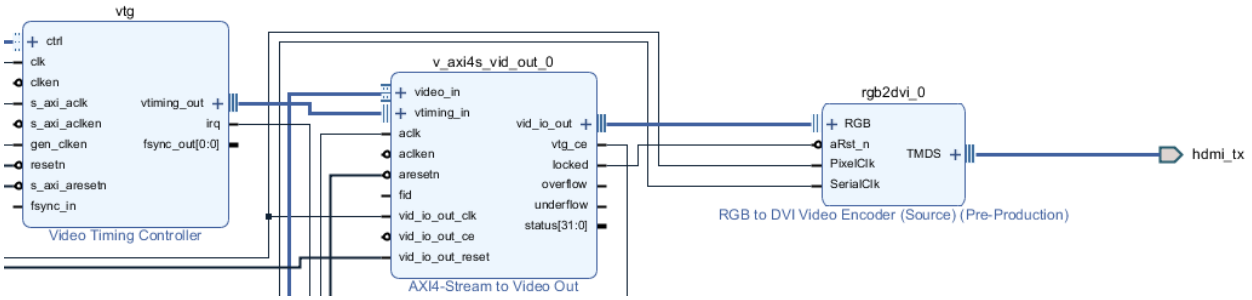


Figure 7: Video Output

- **Video Timing Controller** – Generates the video control signals used by the HDMI output. These signals include VSync, HSync, VBlank, HBlank, field id, and active video.
- **AXI4-Stream to Video Out** – Converts the AXI video stream to a video output interface. This block uses the video signals from the video timing controller and pairs them with the RGB data coming in from the video stream demux.
- **RGB to DVI** – Encodes RGB data as DVI to be output on the HDMI Source connector.

HLS Video Processing Cores

The Vivado HLS workflow is an iterative approach with simulations as verification steps inserted along the way to make sure the design meets the requirements and is functionally correct right from the initial stages. Vivado HLS can:

- Compile, execute and debug the C/C++ algorithm.
- Synthesize into RTL implementation.
- Provide analysis features.
- Generate and execute RTL simulation testbenches.
- Export the RTL implementation as an IP module.

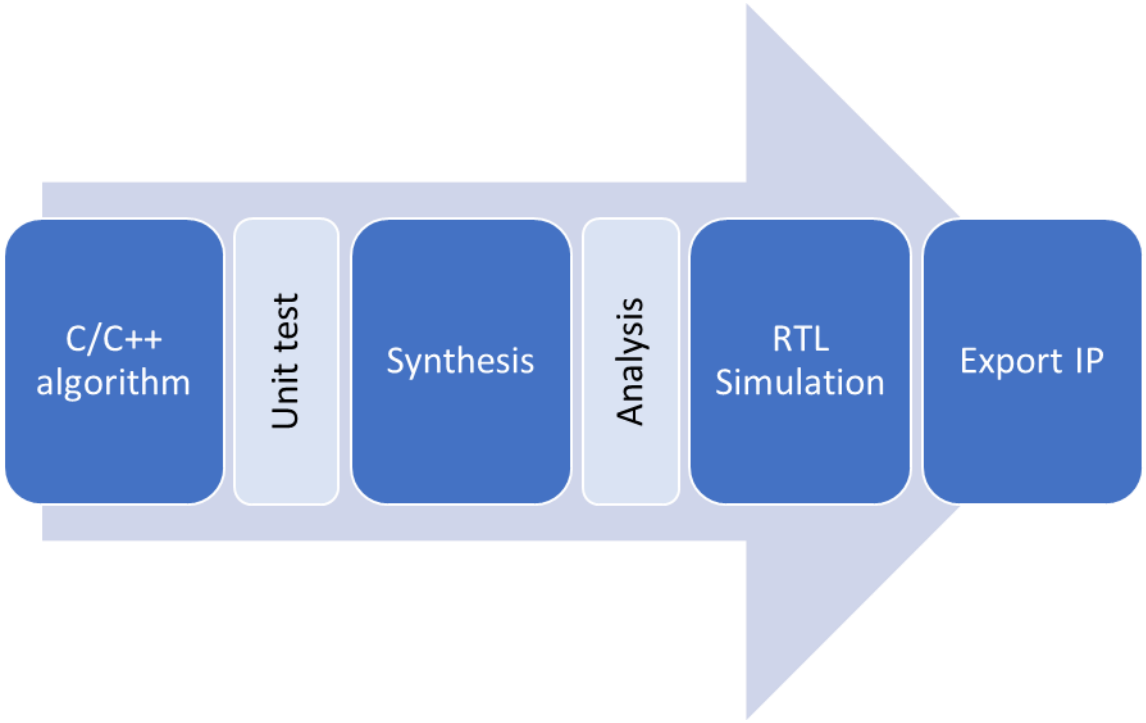


Figure 1: Vivado HLS Workflow Chart

The hls_video API was used to generate the video processing IP in this design. Each function is given the DATAFLOW directive to tell Vivado HLS that this will constantly stream data. The rgb img matrixes (img0, img1, etc) are given STREAM directives. This directive generates a FIFO for each variable and informs Vivado HLS that these operations will be performed in parallel.

- **Blur Edge Detect** – AXI-Stream interface is interpreted as hls::Mat (matrix) data type for easy interfacing with other HLS OpenCV functions. It is then turned to grayscale. The greyscale image matrix is then given a gaussian blur and sent through two Sobel filters, one for the X direction, one for the Y direction, and then added together. It is then converted to RGB values, then converted back to AXI video stream format and sent to the output.

```
void blur_edge_detect(stream_t &stream_in, stream_t &stream_out)
{
	int const rows = MAX_HEIGHT;
	int const cols = MAX_WIDTH;

	rgb_img_t img0(rows, cols);
	rgb_img_t img1(rows, cols);
	rgb_img_t img2(rows, cols);
	rgb_img_t img3(rows, cols);
	rgb_img_t img4(rows, cols);

	hls::AXIvideo2Mat(stream_in, img0);
	hls::CvtColor<HLS_RGB2GRAY>(img0, img1);
	hls::GaussianBlur<5,5>(img1, img2, (double)5, (double)5);
	mysobelxy(img2,img3);
	hls::CvtColor<HLS_GRAY2RGB>(img3, img4);
	hls::Mat2AXIvideo(img4, stream_out);
}

void mysobelxy(rgb_img_t &src, rgb_img_t &dst)
{
	int const rows = MAX_HEIGHT;
	int const cols = MAX_WIDTH;

	rgb_img_t img0(rows, cols);
	rgb_img_t img1(rows, cols);
	rgb_img_t img2(rows, cols);
	rgb_img_t img3(rows, cols);

	hls::Duplicate(src, img0, img1);
	mysobel(img0, img2, 1);//Sobel transform in X direction
	mysobel(img1, img3, 0);//Sobel transform in Y direction
	hls::AddWeighted(img2, 1, img3, 1, 0, dst);
}
```

- **Color to BW** - AXI-Stream interface is interpreted as hls::Mat data type, then turned to greyscale. This matrix is then converted back to AXI video stream format and sent to the output.

```
void color_to_bw(stream_t &stream_in, stream_t &stream_out)
{

      int const rows = MAX_HEIGHT;
      int const cols = MAX_WIDTH;

      rgb_img_t img0(rows, cols);
      rgb_img_t img1(rows, cols);

      hls::AXIvideo2Mat(stream_in, img0);
      hls::CvtColor<HLS_RGB2GRAY>(img0, img1);
      hls::Mat2AXIvideo(img1, stream_out);
}
```

- **Invert Colors** - The stream is first converted into matrix format. The matrix is subtracted by the scalar matrix “pix”. This essentially inverts the colors. The matrix is then converted back to AXI video stream format and sent to the output.

```
void invert(stream_t &stream_in, stream_t &stream_out)
{

      int const rows = MAX_HEIGHT;
      int const cols = MAX_WIDTH;

      rgb_img_t img0(rows, cols);
      rgb_img_t img1(rows, cols);

      rgb_pix_t pix(250,250,250);

      hls::AXIvideo2Mat(stream_in, img0);
      hls::SubRS(img0, pix, img1);
      hls::Mat2AXIvideo(img1, stream_out);
}
```

Software

The processor code is written in Xilinx SDK. The program first initializes the hardware used in this project, then scans for changes in the switches to route the video stream to where it needs to go. When stored in flash or an SD card, this code is run on startup using the First Stage Bootloader (fsbl) provided by Xilinx.

- **Initialization**

```
int main()
{
	init_platform();

	ScuGicInterruptController irpt_ctl(IRPT_CTL_DEVID);
	PS_GPIO<ScuGicInterruptController> gpio_driver(GPIO_DEVID, irpt_ctl, GPIO_IRPT_ID);
	PS_IIC<ScuGicInterruptController> iic_driver(CAM_I2C_DEVID, irpt_ctl, CAM_I2C_IRPT_ID, 100000);
	SWITCH_GPIO sw_gpio(GPIO_SW_DEVID);
	AXI_SWITCH src_switch(SRC_AXIS_SW_DEVID);
	AXI_SWITCH dst_switch(DST_AXIS_SW_DEVID);

	SWITCH_CTL axis_switch_ctl(src_switch, dst_switch, sw_gpio, XPAR_AXIS_SWITCH_0_NUM_MI, 0);

	OV5640 cam(iic_driver, gpio_driver);
	AXI_VDMA<ScuGicInterruptController> vdma_driver(VDMA_DEVID, MEM_BASE_ADDR, irpt_ctl,
			VDMA_MM2S_IRPT_ID,
			VDMA_S2MM_IRPT_ID);
	VideoOutput vid(XPAR_VTC_0_DEVICE_ID, XPAR_VIDEO_DYNCLK_DEVICE_ID);

	pipeline_mode_change(vdma_driver, cam, vid, Resolution::R1280_720_60_PP,
OV5640_cfg::mode_t::MODE_720P_1280_720_60fps);


	xil_printf("Video init done.\r\n");
```

1. The main platform is initialized. This enables caches then resets the PL (FPGA).
2. Interrupt controller is initialized.
3. The GPIO (cam enable pin) and IIC controllers are initialized and are connected to the interrupt controller in software.
4. The switches and stream mux and demux are initialized and added to the switch ctl object.
5. The camera is then initialized using the IIC and cam enable controllers.
6. The VDMA is initialized and tied to the interrupt controller.
7. The dynamic clock generator and video timing controller are initialized.
8. The video pipeline has its video formats configured and started. The chosen resolution is 1280 x 720.

- **Pipeline mode change**

```
void pipeline_mode_change(AXI_VDMA<ScuGicInterruptController>& vdma_driver, OV5640& cam, VideoOutput& vid, Resolution res, OV5640_cfg::mode_t mode)
{
	//Bring up input pipeline back-to-front
	{
		vdma_driver.resetWrite();
		MIPI_CSI_2_RX_mWriteReg(XPAR_MIPI_CSI_2_RX_0_S_AXI_LITE_BASEADDR, CR_OFFSET, (CR_RESET_MASK & ~CR_ENABLE_MASK));
		MIPI_D_PHY_RX_mWriteReg(XPAR_MIPI_D_PHY_RX_0_S_AXI_LITE_BASEADDR, CR_OFFSET, (CR_RESET_MASK & ~CR_ENABLE_MASK));
		cam.reset();
	}

	{
		vdma_driver.configureWrite(timing[static_cast<int>(res)].h_active, timing[static_cast<int>(res)].v_active);
		Xil_Out32(GAMMA_BASE_ADDR, 3); // Set Gamma correction factor to 1/1.8
		//TODO CSI-2, D-PHY config here
		cam.init();
	}

	{
		vdma_driver.enableWrite();
		MIPI_CSI_2_RX_mWriteReg(XPAR_MIPI_CSI_2_RX_0_S_AXI_LITE_BASEADDR, CR_OFFSET, CR_ENABLE_MASK);
		MIPI_D_PHY_RX_mWriteReg(XPAR_MIPI_D_PHY_RX_0_S_AXI_LITE_BASEADDR, CR_OFFSET, CR_ENABLE_MASK);
		cam.set_mode(mode);
		cam.set_awb(OV5640_cfg::awb_t::AWB_ADVANCED);
	}

	//Bring up output pipeline back-to-front
	{
		vid.reset();
		vdma_driver.resetRead();
	}

	{
		vid.configure(res);
		vdma_driver.configureRead(timing[static_cast<int>(res)].h_active, timing[static_cast<int>(res)].v_active);
	}

	{
		vid.enable();
		vdma_driver.enableRead();
	}
}
```

This function changes the resolution that the video stream is received and sent.

1. The video pipeline is reset
2. The gamma value is set and the camera is reinitialized.
3. The mode is set, this can be 720p60, 1080p15, 1080p30, and disabled.
4. The auto white balance is enabled (This can also be configured).
5. The video output controller is reset and configured to output the specified resolution.
6. The video output controller is enabled.

Conclusion

This project scratches the surface of what is possible with the Digilent Embedded Vision Bundle. With the processing power of an ARM A9, the high-speed parallel processing of the programmable logic, and the premade open-source IP cores that Digilent offers, video processing can be taught and experienced by anyone with a computer. With the high demand of computer vision experience in the market, the Digilent Embedded Vision Bundle is the perfect choice for any student, professor, or professional who wants to accelerate their learning in the field.