Image Processing on an FPGA for Low Cost and Low Power Applications on Autonomous Vehicles

By

Benjamin Huntsman

Contents

***Abstract*— A solution to image processing that offers simple expandability and configurability is explained. Using synthesizable System Verilog and simulation coupled with emulation the development was accelerated. This paper explores the design, implementation, and verification of a hardware accelerated image processor.**

Introduction

Computer vision has become an increasingly data intensive subject with images taken for a wide variety of purposes, from filters for cellphone cameras to autonomous vehicles. Computer vision algorithms are complex and continue to become more so. This comes at the cost of higher power requirements for CPUs and slower run times. The implementation of an FPGA based solution for accelerated image processing brings faster processing speeds and a lower power usage ideal for use with autonomous vehicles.

Why System Verilog?

An image processor requires a complex architecture with fast transfer of data between modules. System Verilog offers constructs and easily understandable syntax that helps ensure that the system is more maintainable. Data structures and interfaces simplify the design of each module from its ports to the flow of data within. It also provides the capability of packaging parameters and user defined type definitions for partitioning code into more readable and maintainable sections.

Packages

Packages contain type definitions and parameters to be used in conjunction with a specific module or interface. The package gives a designer one location for configuring that module or interface. For example, the cell processor package contains parameters that if changed will alter the cell processors size or easily reconfigure it to a different color space. It also includes the functions that represent all of the possible operations in the cell processor. (See Packages.sv for more details)

Interfaces

Communications between modules is accomplished through those module's ports. By implementing an interface for this communication, the port definitions for a specific type of communications or for a specific module can be defined in one place and more easily implemented when connecting with other modules. It is possible to have the functions to accomplish these communications be defined within the interface. The cell processor uses an interface to communicate with the image processor. Each cell processor connects to the image processor through its own interface and the ports for communicating are defined for both sides as modports within the interface. Again, giving one place to change port communications for both sides; simplifying the implementation. (See CellProcessor_int.sv for more details)

The Architecture

Cell Processor

An External View

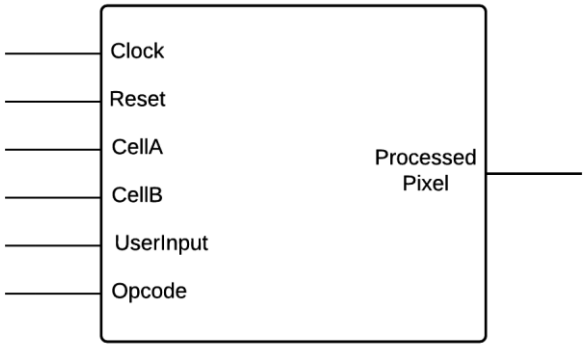

Figure 1: Cell Processor Block Diagram

Externally the cell processor communicates with the image processor via an interface. This interface is made of the input and output ports shown in Figure 1. As the cell processor will operate on whatever data is being held on its inputs every clock it is not necessary to buffer data internally. Currently all the internal logic is combinational. As more complex algorithms are added, it may be necessary to have internal buffering. This would not be for data flow but for holding temporary values instead.

In Code Block 1 it shows the System Verilog syntax for defining the cell processor interface. You will see each of the input and outputs defined and then two modports. The modports are used by the modules that will be connecting using this interface and defines the ports appropriately for each side of the communications path. The two modports are cellPorts, which is for the cell processor, and imagePorts, which is for the image processor.

```
interface cellProcessor_int(input logic clk, rst);
   logic [cellDepth - 1:0]      cellA;
   logic [cellDepth - 1:0]      cellB;
   pixel_t                      userInput;
   logic [opCodeWidth - 1:0]    opcode;
   pixel_t                      processedPixel;

   modport cellPorts (
            input  clk,
            input  rst,
            input  cellA,
            input  cellB,
            input  userInput,
            input  opcode,
            output processedPixel
   );

   modport imagePorts (
            output cellA,
            output cellB,
            output userInput,
            output opcode
    );
endinterface
```

Code Block 1: Cell Processor Interface

An Internal View

The Cell

Internally the cell processor is made of an N x N matrix of pixels, see Figure 2. The value of N is defined in Packages.sv. There are actually four parameters that can be changed to in Packages.sv that will

Pixel (0,0)	Pixel (0,1)	Pixel (0,2)
Pixel (1,0)	Pixel (1,1)	Pixel (1,2)
Pixel (2,0)	Pixel (2,1)	Pixel (2,2)

Figure 2: Cell Processor Model

configure the cell processor to work with whatever color space and cell configuration desired, see Code Block 2. This makes this architecture suitable for a variety of applications.

A Pixel

A pixel is made of one or more color channels with each represented by one or more bits. A black and white image can be defined by a single bit per pixel with '0' for black and '1' for white. Another is monochrome which has a single color channel, normally greyscale, with the bits representing the shade of the color. Whereas Red Green Blue color space or RGB uses three color channels denoted by the name of the color space.

```
parameter opCodeWidth   = 4;
parameter channelWidth  = 8;
parameter channelNum    = 3;
parameter cellN         = 3;
parameter pixelDepth    = channelWidth * channelNum;
parameter cellDepth     = pixelDepth * cellN * cellN;
parameter centerPixel   = (cellN * cellN - 1) >> 1;
parameter divShift      = $clog2(cellN * cellN);
parameter boundUp       = 1 << channelWidth + 1;
```

Code Block 2: Cell Processor Configuration Parameters

In order to allow for the processor to work in any color space each function steps through each color channel using an indexed part select instead of using System Verilog's part naming capability. For more information about how this works see Syntax Notes in the header of Packages.sv for more details.

Functions

Each operation given by an opcode needs to have a function defined. Each function has a single cell, two cells, or a cell and a user input given as a pixel for input data. Their output is a single pixel. This is the way that matrix convolution works and as image processing uses some very complex algorithms that will involve some form of convolution or another, this is an essential aspect of the cell processor. It is also necessary that bounds checking are accomplished for each operation. This ensures that operations act appropriately when the color is saturated or not.

Image Processor

An External View

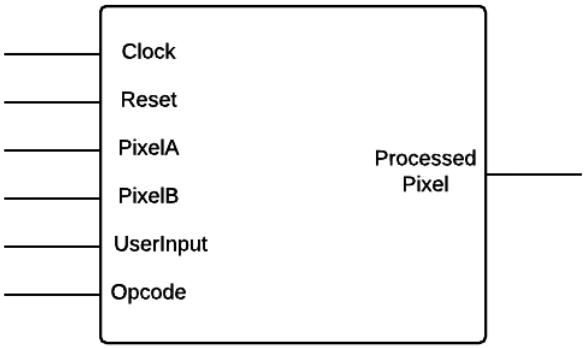

Figure 3: Image Processor Block Diagram

The image processor receives a single pixel at a time for any of its image inputs. This is because it is the smallest piece of data the cell processors will be working with. The image processor assumes that input data is valid and that it is being entered in the appropriate order, that being, from top left of the image to bottom right. Moving left to right and top to bottom. This ordering is necessary as the internal architecture sets up the pixels in this manner. It also assumes that whatever is connected to it is passing data on

appropriate boundaries and understands the delay from input to output and will operate accordingly. Communication with the image processor is also accomplished via an interface as seen in .

```
interface ImageProcessor_int(
                input logic clk, rst);
  pixel_t       pixelA;
  pixel_t       pixelB;
  pixel_t       userInput;
  logic [opCodeWidth - 1:0]   opcode;
  pixel_t       processedPixel;

  modport intPorts (
        input  clk,
        input  rst,
        input  pixelA,
        input  pixelB,
        input  userInput,
        input  opcode,
        output processedPixel
   );

  modport extPorts (
        output pixelA,
        output pixelB,
        output userInput,
        output opcode
   );
endinterface
```

Code Block 3: Image Processor Interface

An Internal View

The architecture used to processor images is visualized in Figure 4 and described hereafter. It is a hybrid of serial and parallel processing. Elements within the image processor and the parameters for using it are found in the image processing package. As the image processor and the cell processor use many of the same elements, the image processor gets most of its parameters from the cell processor package. The only additional parameter is for defining the image width.

The pixels are brought into the processor using an image width wide (or a single row) shift register of pixels. Once the shift register is full that row of the image is shifted into a parallel shift register that we could call the cell processor buffer. This buffer is sized to fit a single row of cells to be used as inputs into the parallel cell processors. As the architecture is setup for normal matrix convolution algorithms, only image width – 2 cell processors are necessary.

The cell processors overlap by two pixels so that their centers are adjacent. As each new row comes into the cell processor buffer, it as if the processors were moving down a row of the image matrix.

Once processed through the cell processors, the resulting row is shifted into another set of parallel shift registers, but this time image width – 2 wide due to the loss of a pixel on either side of the image. This may seem like a lot, but in an image with high resolution it won't even be noticed. This parallel shift register is only used to allow for a clock delay of two clocks. This is also because the loss of the two pixels.

Finally, the row is shifted into a serial shift register to be shifted out of the processor. This architecture is fairly simple, but allows the data to flow in logical manner quickly through the processor. It also allows for the data to flow in and out of the image processor on row boundaries. This simplifies the operation as well.

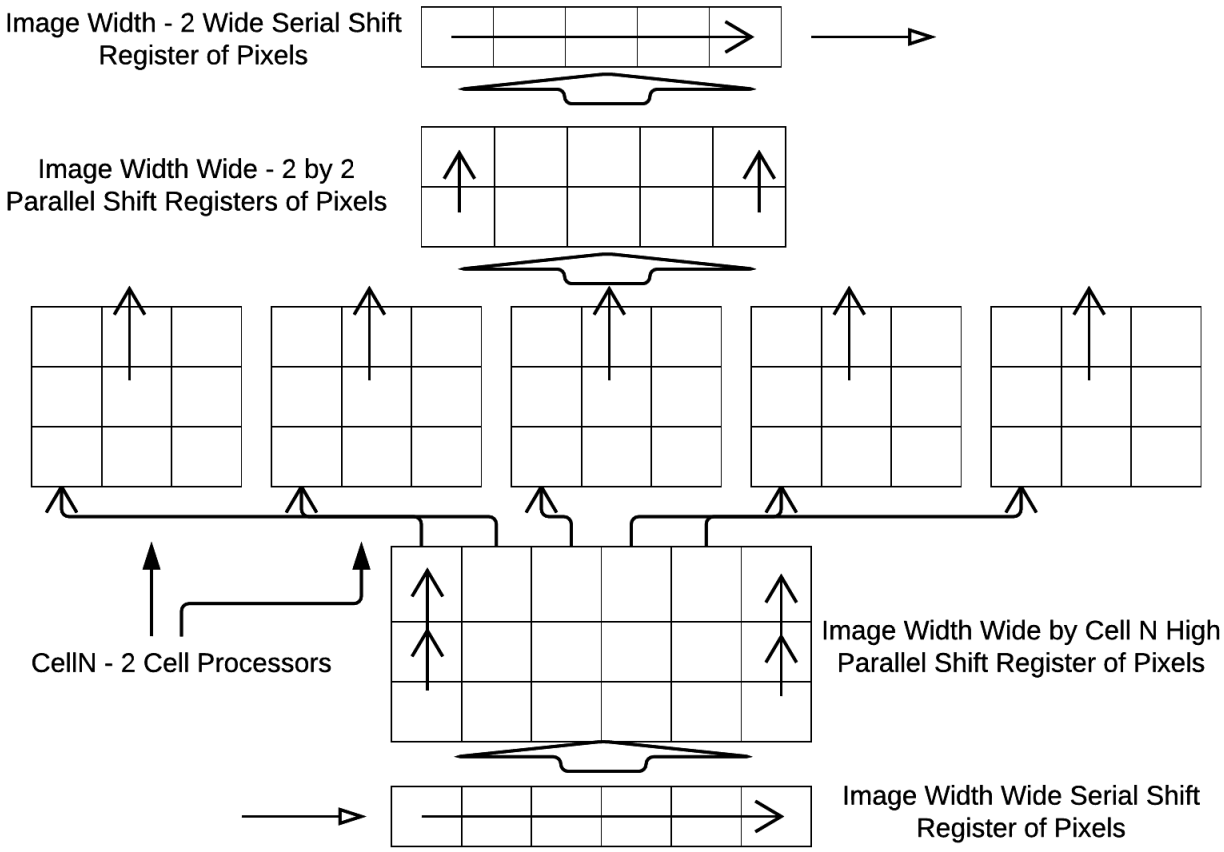

Figure 4: Image Processor Architecure

Validation

Simulation

Xilinix's XSim and Mentor Graphics' Questa Sim were both used in initial simulation testing of the basic design. This continued to be method of choice until a framework was made to be able to use the design in emulation. Although it is possible to verify a design in simulation, there are too many differences between simulation and how something tests after synthesis.

To run the simulation, you will need to install Xilinx's Vivado Suite to get XSim or use Questa Sim on PSU's Linux Red Hat server. The processor for simulation is different for both, but all the necessary elements are found in the project and the instructions follow:

XSim

Questa Sim

Emulation

Mentor Graphics has graced the school with a very expensive and very useful piece of equipment called the Veloce Emulator. Once the framework was finished for using Veloce, all testing and continued development was conducted on the Veloce emulator. The emulator allows for checking for synthesizability and validation of design, where simulation would only validate. As this design is something that will actually be run on an FPGA, synthesizability is a requirement with everything that is done.

The emulator provides several modes. The two that were used in this project were Standalone, and TBX. Standalone mode runs everything directly on the emulator itself and does not take advantage of the advanced capabilities of the emulator. TBX, on the other hand, utilizes co-simulation capabilities. This means that the test bench is written and ran on the Veloce server and the design under test (DUT) is on the emulator. There are a couple of options on how to do this, but the fact is that it allows for a more dynamic testing environment. These two methods and how to use them with this design are described below.

Standalone Mode

TBX

File Structure

Source Code

The source code is found in the main folder under the folder named ‘src’. The system Verilog files for the image processor and cell processor are here. The files needed for working with the Nexys 4 Xilinx board can be found in the ‘Xilinx’ folder with files for a system built to work with any FPGA are found in ‘FPGATestDesign’. The folder named ‘Veloce’ contains files specific to working with the Veloce emulator.

Documentation

All documentation is found in the main folder under the folder named ‘doc’. It contains reference designs, reference articles and papers, and, of course, the documentation for this design.

Version Control

The version control system chosen for this project was Git. Once the project is complete later this summer, I will open up the Git repository for others to retrieve the project. Until then, I believe it is wise to keep it private so that others do not meddle with it or copy the work. I am hoping to get a paper published out of this and possible win a competition, so I don’t want the design taken by someone else at this point either.