XAPP793 (v1.0) September 20, 2012

Implementing Memory Structures for Video Processing in the Vivado HLS Tool

Author: Fernando Martinez Vallina

Summary

This application note describes the main considerations when implementing an image or video processing algorithm with the Vivado™ High-Level Synthesis (HLS) tool. These kinds of algorithms, which are predominantly computation intensive, are natural candidates for hardware implementation with the HLS tool. The techniques described in this application note cover the basics of video algorithm implementation in the HLS tool in terms of synchronization signal handling and memory architectures. Regardless of the exact computation in the user algorithm, these types of applications are memory intensive. The memory architecture expressed in the algorithm has a direct correlation to overall system performance and hardware resource consumption. The recommendations on memory buffer architecture presented in this application note are applicable to Vivado HLS designs on all Xilinx FPGAs.

Introduction

The Vivado HLS tool provides a methodology for implementing video and image processing blocks in Xilinx FPGAs. The HLS tool enables the creation of accelerators targeted at different performance points and different FPGAs from the same algorithmic code. For video and image processing, different performance targets can be specified in terms of maximum image resolution and frames per second.

This application note focuses on the following aspects of video processing IP creation in the HLS tool:

- Overview of the basic video frame format and its description in C/C++
- Memory architecture specification for high-throughput processing

The video processing IP core can be controlled from a processor using application program interfaces (APIs) generated by the HLS tool. For more information on processor control of Vivado HLS IP, refer to *Processor Control of Vivado HLS Designs* [Ref 1].

Programming Environment Specifics

This application note assumes that the user has some general knowledge of the HLS tool and of video IP solutions from Xilinx. For more information on the HLS tool, see *Vivado Design Suite User Guide: High-Level Synthesis* [Ref 2]. Getting started videos for the HLS tool can be found at www.xilinx.com/training/vivado. For more information on video IP from Xilinx, refer to www.xilinx.com.

Video Processing Basics

Video processing can be implemented at different levels of computation granularity:

- Pixel
- Macro-block
- Frame

Regardless of the level of computation granularity best suited for a particular algorithm, the first concern in creating a video processing pipeline is what to do about the synchronization signals. At a fundamental level, all video frames have three regions:

- Vertical blanking

- Horizontal blanking
- Active

The active region is the location of the frame where the image pixels reside. This is the portion of data on which the user algorithm works. It is also the data that is implicitly referred to when a designer starts to think about video processing. The horizontal and blanking regions provide synchronization spacing required for other video equipment to properly detect and display content. Therefore, all video frames have a basic structure similar to Figure 1.

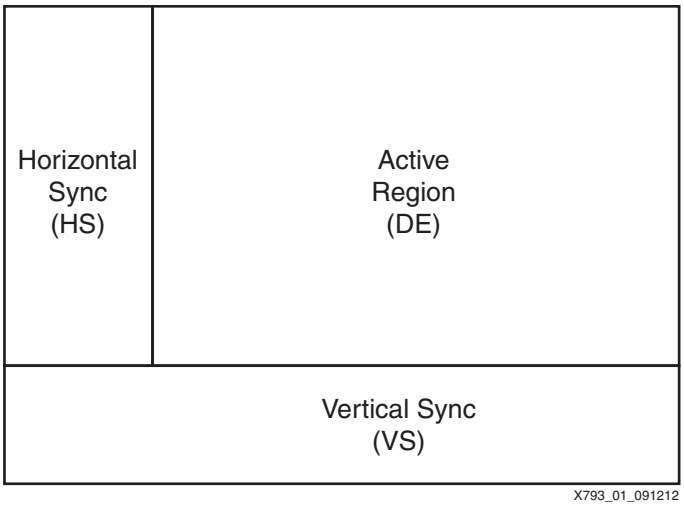

Figure 1: **Basic Video Frame Format**

The synchronization signals shown in Figure 1 (VS, HS, and DE) are required elements of a properly formatted video stream. Although not part of the core user algorithm, these signals must be properly handled so that the results can be accepted by other video processing IP or equipment. In terms of a design for the HLS tool, these signals can be handled either at the system integration level or within the code for the HLS tool.

To process synchronization signals in the HLS tool, the user must be explicit in the use of these signals in the design C/C++ code. The HLS tool is a general-purpose algorithm synthesis tool and does not have any built-in knowledge about video processing specifics versus any other type of computation. Therefore, the user code for the HLS tool must be in a form similar to this sample code:

```
void video_proc(input_video, output_video){

  for(i=0; i < MAX_FRAME_SIZE; i++){
    if(VS){
      ......
    }
    if(HS){
      ......
    }
    if(DE){
      //Pixel Processing Algorithm
    }
    ....
  }
}
```

This example code shows the additional complexity of integrating sync signal processing as part of the core user algorithm. Sync signal processing makes it difficult to modify an algorithm. It also creates a departure from expressing video processing in terms of loops over rows and columns, which is the traditional way most video processing algorithms are captured in C/C++.

Xilinx provides an IP solution to remove the processing of sync signals from the scope of the design being implemented in the HLS tool. The Video In to AXI4-Stream [Ref 3], [Ref 4] and AXI4-Stream to Video Out [Ref 5] IP cores handle the formatting of pixel streams into valid video streams. By using these cores at the system integration level, the user can capture their video processing algorithm in terms of rows and columns, such as in this sample code:

```
void video_proc(input_pixels, output_pixels, height, width){

  for(i=0; i<height; i++){
    for(j=0; j<width; j++){
      //Pixel processing algorithm
    }
  }
}
```

This example code shows the recommended approach to capturing video processing algorithms in the HLS tool. In terms of hardware, the recommended approach using Xilinx video IP results in the block diagram shown in Figure 2.

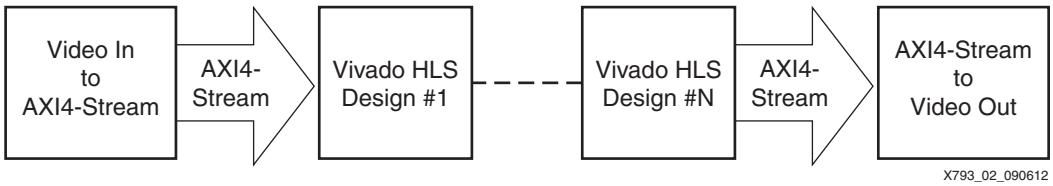

Figure 2: **Vivado HLS Video Processing Pipeline Block Diagram**

The block diagram in Figure 2 shows the recommended way to build video processing pipelines in the HLS tool and how to handle synchronization signals. The other key aspect to video processing in the HLS tool is the instantiation and handling of memory buffers within the user algorithm.

Memory Architecture

Memory buffers are a fundamental feature of video processing algorithms. These memory structures allow the user to have the temporal and spatial data contexts required by an application to work on the current pixel. Three memory styles common in video and DSP applications are shift registers, memory windows, and line buffers. All of these memory structures have implications on latency, order of computation, and functional correctness of the hardware generated by the HLS tool.

The most important implication of these memory structures is their effect on the functional correctness of a design. This implication guarantees that the HLS tool does not automatically insert new memories into user code. The user must explicitly code shift registers, memory window, and line buffer behavior into the code targeted for RTL generation. Only then does the HLS tool build these structures into the RTL.

Shift Register

Shift registers are one of the simplest and most commonly used memory structures in DSP processing. The idea behind a shift register is to provide a one-dimensional temporary data buffer to store the incoming samples from a streaming interface. The duration for which a sample is stored in the shift register is determined by the algorithm being implemented. One example of a typical DSP algorithm that can be implemented with a shift register is a finite impulse response (FIR) filter. The conceptual diagram of an FIR filter with a shift register is shown in Figure 3.

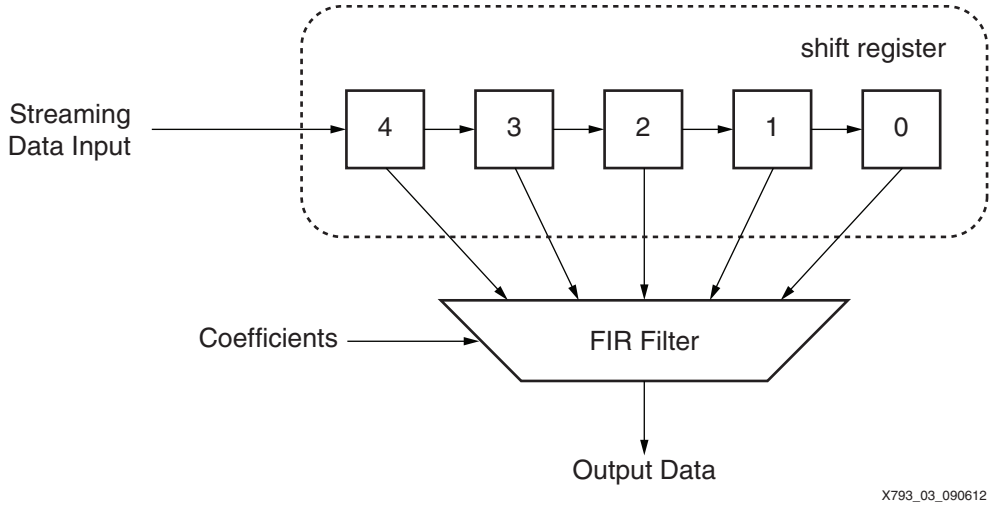

Figure 3: **FIR Conceptual Diagram**

As shown in Figure 3, each data element from the streaming interface enters the shift register at position 4 and is reused an additional four times by the FIR filter before being discarded. By definition, a streaming data interface produces a specific data sample only once. Therefore, the data sample must be stored before it can be used.

In the HLS tool, an array is the simplest and most straightforward way of declaring a shift register. For the example in Figure 3, there are two ways of expressing shift registers depending on how the input streaming interface is defined, as shown below.

```
int A[5];
int new_data[5];
int i;
for(i = 0; i < 4; i++){
#pragma HLS unroll
  A[i] = A[i+1];
}
A[4] = new_data[k];
```

```
int A[5];
hls::stream<int> new_data;
for(i = 0; i < 4; i++){
#pragma HLS unroll
  A[i] = A[i+1];
}
A[4] = new_data.read();
```

The code examples above show how a shift register can be created from array A. The `for` loop in the code examples dictates the data movement in the shift register from element 4 down to element 0. In hardware, all elements in a shift register move simultaneously on every clock cycle. This desired hardware property can be exposed to the HLS tool by fully unrolling the loop specifying the data movement in A. Although there are several ways to unroll a loop in the HLS tool, the examples above use the following:

```
#pragma HLS unroll
```

By placing a pragma directly in the code, the user is specifying that A will behave as a shift register across all possible performance and implementation targets. This removes any ambiguity that can result from performance-based unrolling strategies in Vivado HLS. Pragmas placed in the user code are automatically executed on every implementation.

The difference between the previous code examples is in the method for obtaining new_data from the input interface. The first uses implicit streaming by pulling data from the kth location of array new_data. The advantage of this approach is in not requiring alterations to the original C/C++ testbench code. Also, during code development, the user has full access to the new_data array for debugging any problems that might occur when a shift register is inserted into an algorithm.

After everything is working correctly, new_data can be transformed into a streaming interface using either of the following commands:

```
set_directive_interface -mode ap_fifo foo new_data  (Vivado HLS directive)
```

or

```
#pragma HLS interface ap_fifo port=new_data  (Vivado HLS pragma)
```

In the case of streaming interfaces, the HLS tool supports `ap_fifo` and `ap_hs` as native handshaking/streaming protocols. Refer to *Vivado Design Suite User Guide: High-Level Synthesis* [Ref 2] for additional documentation on these modes.

When using implicit streaming, the user must be aware of the conceptual differences between the C and the HLS tool-generated RTL simulation model. The transformation changing new_data from an array into a streaming interface is only visible in the RTL simulation model after the design has been synthesized by the HLS tool. Output from the C and RTL simulation models only match if new_data is accessed in a sequential and linear order. It is the responsibility of the user to ensure that the array is accessed in this way.

In contrast, the second code example introduces explicit streaming with the `hls::stream` class. The `hls::stream` class enforces streaming FIFO ordered read and write semantics at the C simulation level. Also, using this class automatically results in a streaming interface at the RTL level. Refer to *Vivado Design Suite User Guide: High-Level Synthesis* for additional documentation on `hls::stream`. After streaming I/O has been added to an algorithm with implicit streaming, it is recommended to lock the behavior in place with explicit streaming. Changing from implicit to explicit streaming helps the user avoid coding changes at the interface level that can affect the functional correctness of the algorithm.

Memory Windows

In video and image processing, a memory window is defined as a neighborhood of N pixels centered on pixel P. The memory window can also be viewed as a collection of shift registers, which forms a 2-dimensional data storage element. This kind of memory is usually implemented as flip-flops for these reasons:

- It has fewer data elements. Only the pixels required to compute some characteristic of P are stored. An example of this is the 3 x 3 memory windows used in edge detection.
- All pixels in the neighborhood must be simultaneously available when computing the value of P.

The most basic definition of a memory window in C/C++ is a 2-D array. For example, a 3 x 3 memory window B can be defined as:

```
int B[3][3];
```

One of the characteristics of a memory window is that all data elements are available simultaneously. In the HLS tool, this characteristic can be achieved through partitioning the array into its individual elements, else the array will be implemented as a block RAM. Array partitioning can be set through either a directive or a pragma. In the case of B, the partitioning command is either of the following:

```
set_directive_array_partition -type complete -dim 0 foo B (Vivado HLS
directive)
```

or

```
#pragma HLS ARRAY_PARTITION variable=B complete dim=0 (Vivado HLS pragma)
```

where:

- `complete` splits a dimension of an array into individual elements.
- `dim = 0` is a partitioning command that is applied to every dimension in the array.

Using either the directive or pragma approach for array partitioning results in B[3][3] being decomposed into nine independent registers at the RTL level.

Moving data in a window style memory can be treated the same as a shift register. The vertical motion across a pixel frame can be constrained to the line buffer, which is used to supply data to the window. With this simplifying assumption, the horizontal movement of the window is shown below.

```
for(i = 0; i < 3; i++){
#pragma HLS unroll
  B[i][0] = B[i][1];
  B[i][1] = B[i][2];
}
```

An example of how to copy data from a line buffer D to a window B is shown below.

```
for(i = 0; i < 3; i++){
#pragma HLS unroll
  B[i][2] = D[i][col];
}
```

Line Buffer

A line buffer is a multi-dimensional shift register capable of storing several lines of pixel data. Typically, line buffers are implemented as block RAMs to avoid the communication latency to off-chip DRAM memories. Also, a line buffer requires simultaneous read and write access, which takes full advantage of the dual-port nature of block RAMs. Although a memory window is a subset of a line buffer, a line buffer cannot be used directly in most video and image processing algorithms. The path traversal of an incoming pixel to the algorithm computation kernel is shown in Figure 4.

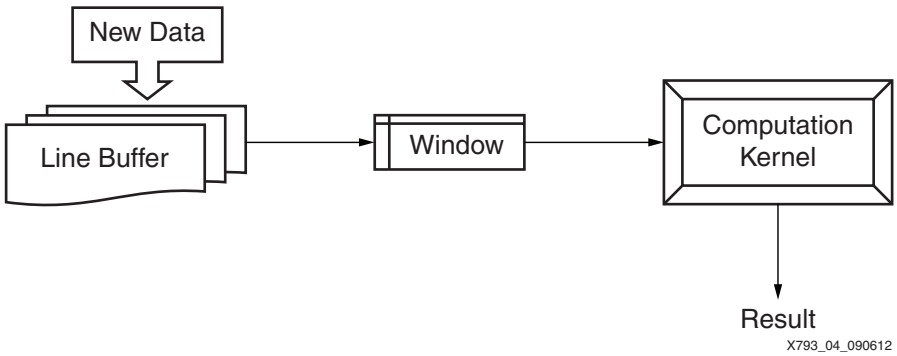

Figure 4: Pixel Memory Traversal to a Computation Kernel

As in the case of a window, a line buffer is declared in C/C++ as a multi-dimensional array. In determining the height of the line buffer, one simple rule is to keep it at the same height as the memory window. For example, the corresponding line buffer to window B in the FPGA is declared as:

```
rgb_pixel D[3][MAX_COLS]
```

Declaring a full three lines of buffer storage is not efficient in terms of FPGA resource use, but it is the easiest to code in C/C++. The area inefficiency is a result of window processing algorithms not requiring a full N lines of data to begin the computation. Storing the full N lines instead of N-1 lines and several pixels results in an overhead of several block RAMs. Depending on device utilization, the example C code shown in this document can be further optimized to reduce block RAM consumption.

Along with a data movement specification similar to that of a shift register, a line buffer requires the user to rethink the boundaries of the processing loop in the algorithm. The algorithm must

take into account two effects of line buffering. The first effect is the time required to fill the line buffer with sufficient data to compute the first output value. The second effect is that the algorithm will have to run for more iterations than there are available input samples to produce all output samples.

Continuing with the example of line buffer D, the computation loop of the algorithm is extended, as shown below.

```
for(row = 0; row < MAX_ROW; row++){
  for(col = 0; col < MAX_COL; col++){
    ....
  }
}
```

to

```
For(row = 0; row < MAX_ROW+1; row++){
  for(col = 0; col < MAX_COL+1; col++){
    ....
  }
}
```

The extension of the boundary condition by 1 accounts for the offset between input and output data created by the insertion of a line buffer. With the processing loop extended beyond the boundaries of the incoming data set, the user must add gating conditions to ensure the correct functionality of the algorithm.

The input to the algorithm is bounded by the original boundaries in the first code example above. Following the data flow of the second example, all input data is initially written to the line buffer. This implies that the gating condition must be placed on the input data write to the line buffer. An example of an input data gating condition is shown below.

```
if(col < MAX_COL & row < MAX_ROW){
  D[2][col] = new_data[k];
}
```

Writing results to the output interface must also be delayed to account for the line buffer delay. An example of a condition to gate the output is shown below.

```
if(col > 0 & row > 0){
  output_pixel[k] = result;
}
```

Data movement in a line buffer is a vertical shift in which new rows are added as older rows are discarded. The horizontal move of the line buffer along the column axis happens automatically as a result of the inner processing loop over col. The vertical shift of the line buffer can be coded as shown below.

```
if(col < MAX_COL){
  D[0][col] = D[1][col];
  D[1][col] = D[2][col];
}
```

Conclusion

By coding memory architectures and usage models as part of the algorithm, the user has full control over the type of hardware being created by the Vivado HLS tool. This allows the user to quickly explore the impact of different memory layouts on both area and performance. The coding examples and tool settings presented in this application note enable the user of the HLS tool to implement memory structures typically used in the implementation of video processing algorithms.

References

This application note uses the following references:

1. XAPP745, *Processor Control of Vivado HLS Designs*
2. UG902, *Vivado Design Suite User Guide: High-Level Synthesis*
3. PG043, *LogiCORE IP Video In to AXI4-Stream v2.00.a*
4. UG761, *AXI Reference Guide*
5. PG044, *LogiCORE IP AXI4-Stream to Video Out v2.00.a*

Revision History

The following table shows the revision history for this document.

Date	Version	Description of Revisions
09/20/12	1.0	Initial Xilinx release.

Notice of Disclaimer