New Methodology For FPGA Based Designs Offers Significant Benefits over HDL Based Methods

Kent L. Gilson, Star Bridge Systems, Inc. * Gary DePalma, quickSTART Consulting

HeadStart™ FPGA Designer is a new methodology for FPGA based designs that offers significant benefits in time to market, design performance, and design reuse over HDL based methods. It captures designs graphically at a higher level of abstraction (architecture level), which produces "chip in a day" levels of productivity. Its high level of integration offers faster debugging and shorter time to market. The high level capture enables the recursive synthesis process to design extremely high performance designs that are area efficient. And design reuse is raised to a new level with less effort than ever before.

Introduction

This paper will walk through a FPGA design that was done using only the HeadStart™ FPGA Designer tool set. It is intended to give a feel for how design can be addressed using HeadStart. The design described here has been used often to demonstrate the effectiveness of HeadStart.

The authors are working on a long-term project to model the visual cortex as a system. The human visual recognition system is made up of many simpler layers such as motion detection, shape recognition, and edge detection. As part of this project, an edge detector has been developed as a separate demonstrable subsystem, to be in the final system.

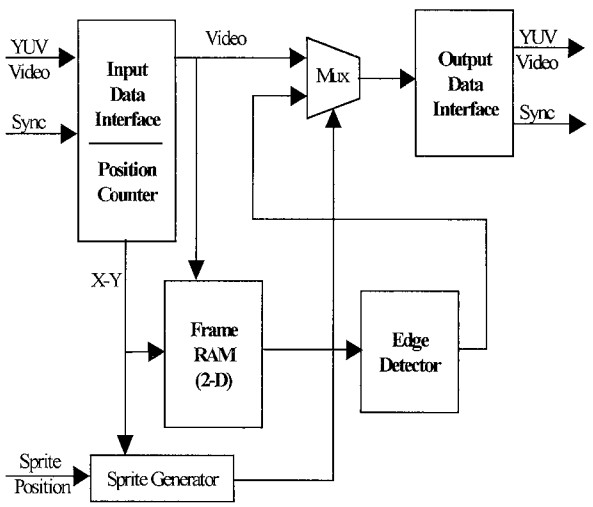
A Real-time Video Edge detector

The application described here is a real time video edge detector. There are two versions, a one-dimensional (1-D) version that detects only vertical edges and a second 2-D detector that detects edges with any orientation. The 1-D version looks only at five pixels in a single line and requires very little memory. The 2-D version is much more memory intensive. Since NTSC video is interlaced, that is sent as two fields with the odd lines in one field and the even lines in the other, the whole frame (both fields) must be stored so that a five-by-five block of pixels can be accessed by the special filter.

The edge detection module was thought to be a good demonstration vehicle because it includes most of the issues which need to be solved in a complete design including interfacing to a previously built (fixed interface) video conversion board, developing the processing blocks, and tuning the coefficients in the blocks to give the best results. However, the design itself is simple enough to be understood by everyone.

The video source chosen is a camcorder that has (U.S.) standard NTSC video output. Because it uses video, any NTSC video source could be substituted later. Results will later be fed into additional processors to simulate the entire pattern recognition process but for development and demonstration purposes they could be viewed on a standard TV or monitor with an NTSC input.

Since NTSC video is an analog signal and the FPGA does not support any analog processing, we need to convert it to digital outside of the FPGA. The demo system utilized an existing board with a Philips chipset that converted NTSC video into an 8 bit wide YUV digital pixel stream for processing. It also detects and presents the horizontal and vertical sync signals. Finally, the board also has a digital-YUV to NTSC converter on it that could be used to display the resulting edge enhanced images on a standard TV monitor. The major portions of the edge detector are shown in figure #1.



Edge Detector Top Level Figure #1

Today, this is more a research project than a development project where the goal is to develop visual recognition algorithms. However, if the recognition system is successful and commercial applications present themselves, it should be possible to move the design to a CMOS ASIC with an on-board Imaging array.

General Approach

As far as the edge-detection is concerned, the design was approached mostly top-down. However, the edge-detector is a bottom-up block in the larger system. We proceeded by building and testing each edge-detection subsection with the yet undersigned blocks stubbed out so testing could be done at the top level. After each subsection was completed and working, we began work on one or more of the stubbed blocks. The first blocks to be designed were the input and output interfaces. These could be built with their video lines connected together thus stubbing out the rest of the design. Live video passed through from a video cam to the monitor with no processing should show no distortion or changes. By building the interface first, we were able to test the ability to correctly input and output the digital video in real time, properly extracting the end of row and frame signals (sync). Also tested in this block was the extraction of odd/even frame signal.

After building the interface blocks, the amount of sync delay could be adjusted to ensure we were capturing the video and not background of the retrace intervals. In the final design, the sync delay would be handled by a constant. However, in the early design stages, we declared these constants as inputs. This way, when we simulate, we can utilize the simulator's slider object that can change the value on an input in real time finding the right constants to position the image in the center of the monitor.

During simulation, the slider is attached to specific interfaces through a user interface definition. It appears on the screen looking like a scroll bar for a window. The user can adjust the sliders to adjust the value of the input (future constants) until the imaging was correct.

Then the input objects were replaced with constants allowing the logic optimizers to eliminate unnecessary hardware. It was interesting to note that even though a common chipset was used for NTSC conversion, the actual delays on input and output differed slightly for best results. This was likely due to the pipelining done inside the interface circuits.

NTSC video is sent as two sequential fields. The first field contains the odd lines of the video and the even lines are sent in the second field. The field rate is approximately 60 Hz where the frame rate is half that. This is called interlaced video and each field holds only half the resolution in the vertical direction. To do 2-D edge detection, we need to "de-interlace" the video by storing it in an array. The interface uses the horizontal and vertical sync to count the x and y directions and use that as an address to store and extract the video from the RAM. These x and y counters are also used as an address when generation output sync.

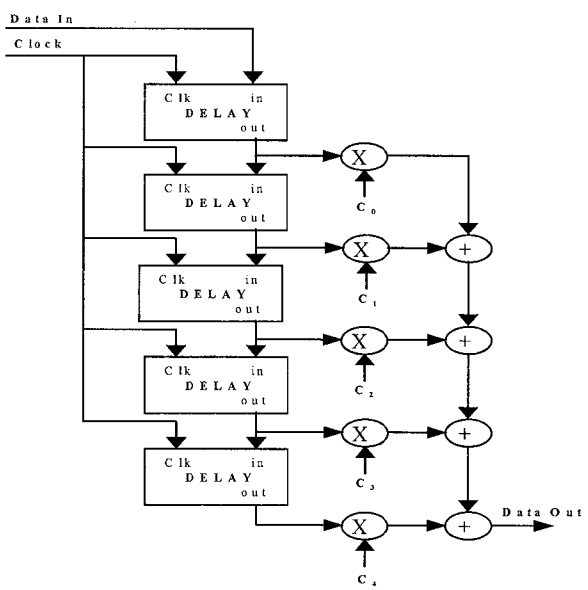
The Sprite Generator

With the ability to get real time video in and out of the chip, we now focused on the method of simultaneously displaying both the incoming video and the edge enhanced view that would be generated by the edge detector. We chose a sprite generator as the best method. A sprite allows us to display the incoming video but replace it inside of the sprite with the edge-enhanced video. For simplicity, we chose a rectangular sprite that could be moved horizontally and vertically on the screen. This way, we can view the incoming video or the edge enhanced video at any point on the screen.

We already had the x and y positional counters in the interface block and all that is needed for the sprite generator was to compare the x and y position with the size and coordinates of the sprite and decide whether or not the current position was inside or outside the sprite. A video rate multiplexer switches the video output between the incoming video and the edge-detected video. The sprite generator was built and tested with the edge detector stubbed out by using a constant intensity and color as the second input to the multiplexer.

The Edge Detector – Real Reuse in Action

The edge detector for the 1-D unit was chosen as a 5-pole finite impulse response (FIR) filter. The Figure #2 shows the block diagram of a 5-pole FIR filter.



A 5-pole FIR filter

Figure #2

A FIR is a good example of a block that could be reused and should be put in the user's library. We would like to generalize the FIR filter in order to increase its value as a reusable object. The first and most obvious generalization is for the FIR to accept different data types (float vs. integer, size) to improve its reusability. With HeadStart, this is the natural design method. It is just a simple case of assigning variable width elements (variant data type) for the input. The library elements are defined with variant types as well. We just wire them up on the graphical capture tool. When the block is used and is connected to an actual data source, in this case the video stream, the blocks will automatically be sized (polymorphed) to the correct size. There is essentially no additional work required to build a data-size polymorphic FIR.

In addition to the variable data size, the constants should have flexible size. It is likely that the best filter will require different sized constants (or a lot of leading zeros) if a fixed-point implementation is being built. Again, we define the inputs as variant data type and the multipliers and will adders will polymorph to the correct size.

However, if the data is n bits and the constant is m bits, then the output of the multiplier will be (n+m) bits requiring the adders to be that size plus a small guard band. However, it is most likely that the output of the filter should be the same type and size as the input. Instead of adding up all of these wide words only to throw half the bits away, we can cast the multiplier output with or without rounding. There is a cast entity in the library which takes two inputs, one is the data and the other is connected to a signal only to pick up the data type that you want to cast the result to. We can hook this type line to the data line casting the output of the multiplier back down to the same size as the input.

Data-type polymorphism can be implemented in an HDL to encourage reuse but this is not without significant pain. First, you must define the data as an array with a variable width. Next is the question of where do you define the width, as a parameter or as a constant defined in an included file. To pass it in as a parameter, the next layer of hierarchy up must be modified to include or calculate the size. To use an include file, you have to create one. Usually you want to have a single parameter file for the project. This can cause name collisions if you have implemented more than one filter or if another designer has used the same name in a different design. Finally you must document the parameter and its use. The extra time needed to design for reuse using an HDL methodology goes against the designer's time to market pressures. For this reason, the steps needed to design for reuse are often skipped. With an HDL methodology, reuse costs and your only choice is to pay now or pay later.

HeadStart, on the other hand, both lowers the cost of reuse while it increases the value.

Data-rate Polymorphism – A Step Up in Reusability

Besides generalizing the design for data types, the design should be capable of adapting to the speed requirements. A filter can be used in many applications. An electro-mechanical application such as a switch debounce circuit could have data rates below 100Hz. An audio filter may only need to run at 8kHz and our video filter runs over three orders of magnitude faster in the 20 MHz range. But the algorithm of a filter remains constant. But reuse of a design over a wide range of data-rates is usually not efficient with an HDL methodology.

When you implement multiplication in an HDL, you have two choices, type A*B and hope for the best or use the A*B only for the simulator and import a library element. Most of the library elements are designed for speed and to implement the slower speed applications, you would probably want to share the resource. This requires the designer to modify the RTL to share the resource. There are additional tools today that can help in this area but not at no cost.

HeadStart capture is truly implementation independent. The outputs can be annotated with the required data rate and then the rendering engines can choose the best implementations. Sharing resources is one way but since the multiply library is also based on algorithmic capture, the multiplier can be re-implemented to better fit the data rate by implementing a slow, very small multiplier. Actually, many slow multipliers can actually be more area efficient than a fast one being shared by the various stages in the filter. Additionally, using several small multipliers eliminates both the extra circuitry and the design work needed to share the resources.

What is happening here is called data-rate polymorphism. By capturing the algorithm (not the implementation) with all of its parallelism intact, HeadStart can build very fast circuits even in an FPGA. However it also can reduce the area by eliminating the parallel operations that are not needed to meet the data rate requirements. It is "easy" to convert a parallel design into a serial design without additional information. However, it is nearly impossible to take a specific implementation with serial operations and generalize it to be faster using more parallel operations.

Using Recursion to Add Another Dimension to Reusability

Programmers have used recursion for years to simplify and generalize a programming problem. For example, recursion can be used to design a variable (infinite) precision math library. Why not use recursion for hardware design? Seems reasonable but today's HDLs are not re-

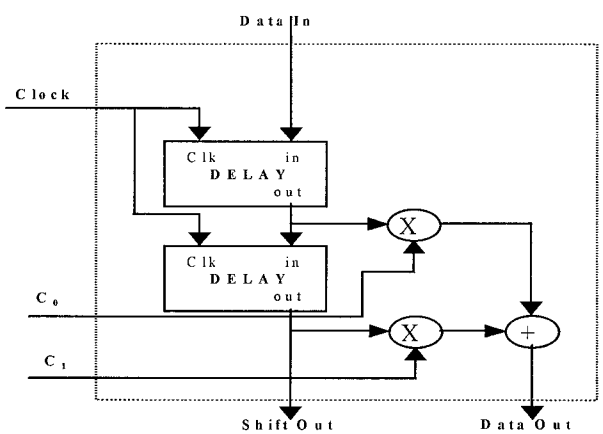
instantiate or call itself. HeadStart capture and synthesis is recursive and recursion is a major technology used to create the data-rate polymorphism. For example, from an algorithmic level, an n-bit adder is the same as an (n-1)-bit adder and a 1 bit adder with the carry passed across. This is the way we define an adder, recursively.

But recursion needs a place to stop. With a concept borrowed from operator overloading used in programming languages, we can have several definitions of an adder as long as the input types are different. During synthesis, the actual choice of which adder is made by the data types attached. Thus a variant input adder recursively reduces the word size until they match the input sizes of a fixed adder. For integers, we only need to define a 1-bit adder and a variant adder to get all possible adders. Remember that this is algorithm level capture so the implementation is not being fixed as it would be with an HDL.

But recursion can be used in another way. In designing a system, I might need to have a filter with a different number of poles. Without recursion , I would have to define each filter separately or add a FOR loop to instantiate the filter and go back to the problems of documentation and variable collisions mentioned above. With HeadStart, we defined the FIR filter recursively based on the number of inputs

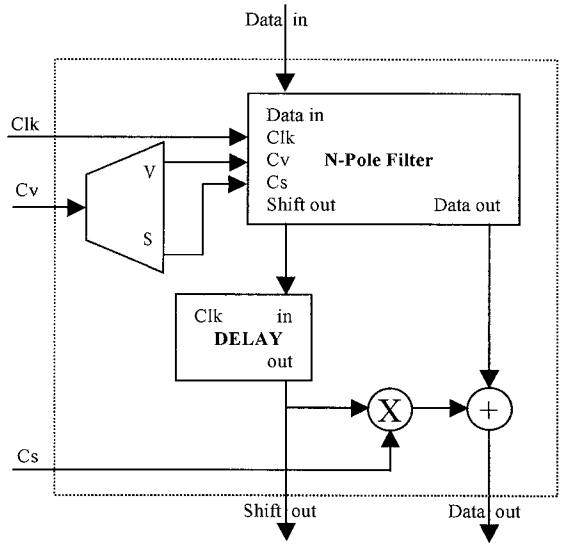
We started by looking at the design of an n-pole filter as shown earlier. It is clear that in this 5 pole filter, there is a four-pole filter plus an additional delay, adder and multiply block. To define recursion on the n-pole filter, we define a two pole filter (Figure #3) which accepts two constants, both scalars of any data width. It used the same data-size polymorphic elements as before. However we added an additional output which is the last stage of the delay chain so that we could add additional stages.

Figure #3



A 2-pole FIR filter

In the recursive filter, we will supply it with a vector of constants. The number of scalars in the vector will determine the number of poles. Next we employ a library element called a data-exposer. A data-exposer accepts a more complex data type and splits it into two simpler components. For our vector, it strips the first element of the vector off and outputs it as a constant. It also outputs a vector with one less component. If the input to the data-exoposer is a two element vector, both outputs will be scalar. It is a simple matter to design an n-pole filter recursively as shown (Figure #4).



Recursive N-Pole FIR Filter

Figure # 4

Finally, it would be nice if the filter could be defined to have a single input (vector) for the constants and not have to require the user to input one constant as a scalar and the others in a vector. To solve this we add one more layer of recursion for the top level. This accepts all n constants as a vector and splits one scalar off. It then instantiates the first n-pole filter with two constant inputs, a scalar and a vector. This top level FIR filter is shown in Figure #5.

Even though all three blocks have the same name in the library (FIR) the recursive synthesis process can easily differentiates which to use because they all have number of inputs or different input types. With this type of recursion we can capture and test our design without having to make a decision on the number of poles in each filter.

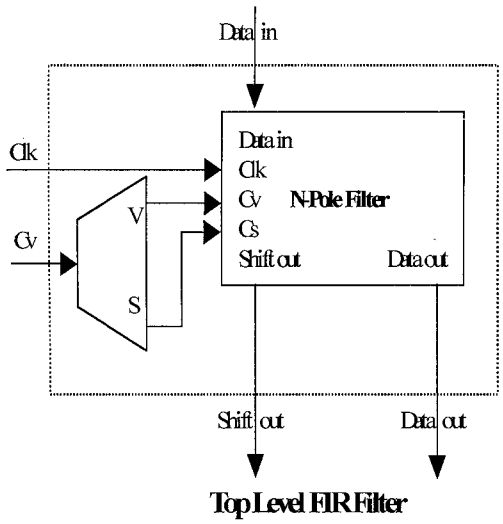


Figure # 5

About the authors

Kent L. Gilson is Chairman and Chief Technology Officer of Midvale, Utah based Star Bridge Systems, Inc. Star Bridge's high-performance, reconfigurable Hypercomputers are sold in the capability computing market. The company's HeadStart™ and Alpha Series Emulator™ products for system-level design, synthesis and verification will be introduced into the EDA market in early 2002. Kent is a leading proponent of reconfigurable computing and has won national awards from Xilinx Corporation and the National Association of Music Merchants for his reconfigurable computing products. Kent can be reached at Star Bridge Systems 801-984-4444 or at SBS@starbridgesystems.com.

Gary DePalma has more than 20 years experience in the electronics industry. He has held engineering and management positions at companies that include Bell Labs, Silicon Compilers, and Intel. He is currently a principal at quickSTART Consulting where he works with new ventures to help develop innovative products. Gary can be reached at garydep@ieee.org.