

Real-Time Image Manipulation Using Soft Hardware

Richard G. Shoup

Interval Research
1801 Page Mill Road
Palo Alto, CA 94304 USA
email: shoup@interval.com

Abstract This paper discusses the use of restructurable hardware, specifically Field Programmable Gate Arrays, in real-time image processing and manipulation tasks such as convolution filtering, scaling and rotation, composition, color space transformation, etc. Each of these functions can be implemented using a customized pipeline design to obtain a high degree of parallelism and thus high performance. In this work, we show how a simple arrangement of FPGAs and memory can be used to synthesize a wide variety of image processing pipelines having different topologies and functionality.

1. INTRODUCTION

Most imaging processing operations such as convolution filtering, compositing and blending, compression, color change and color transformation, morphology, etc. [1] are good candidates for implementation in custom hardware. This is primarily because:

1. These functions generally operate identically, homogeneously, and independently on a large number of pixels, which thus can in principle be performed in parallel.
2. The function being performed on each pixel is often simple. This means that in a software implementation the loop and other overhead can be significant, while in a hardware implementation only a small amount of circuitry is required.
3. High performance is frequently required in imaging applications such as real-time video processing, teleconferencing, computer vision, and image databases.

Because of factors 1 and 2, hardware accelerators for a *specific* image processing function can often outperform conventional microprocessor implementations by several orders of magnitude at modest cost. This speedup is achieved through a combination of

1. parallelism in space (i.e., replication of hardware with execution of several identical functions at once), and
2. parallelism in time (i.e., pipelining so that several

different stages of the computation are executing at once), and

3. reduction of overhead such as memory accesses, loop count increment and test, etc.

While this is advantageous for a given function or small class of functions, it is much more difficult to achieve these benefits over a variety of imaging functions. In general each function requires a different parallelization, a different pipeline structure, a different memory topology, and often different types and lengths of arithmetic. Simply put, greater parallelism (simultaneity in space or time) implies greater structure and thus less generality in the problem domain. The great generality and longevity of the VonNeumann architecture is in its simplicity and serial nature.

It is also worth noting that existing commercial Digital Signal Processors (DSPs) [2] are becoming increasingly similar to conventional microprocessors, in part for the above reasons. DSPs are perhaps better characterized as specialized microprocessors than as signal processing circuits with control.

One solution which can provide nearly the same dramatic performance increase over a very wide range of functions is *restructurable hardware*, often called *programmable logic*. Image processing functions are particularly well-suited to this technique, as can be seen in the examples below.

2. RESTRUCTURABLE HARDWARE

Programmable logic is hardware with added gates and configuration logic which allows its function to be changed under software control at any time. This concept originated quite a few years ago [3,4], but has only recently (ca. 1985, see for example [5]) become commercially practical due to advances in integrated circuit technology.

A. FPGAs

Programmable logic is usually constructed in an array of identical cells called a Field Programmable Gate Array or FPGA. Fig. 1 shows a typical fine-grained FPGA [6] consisting of three thousand cells, each containing a few

gates and one flip-flop. Each cell can implement frequently-needed functions such as a one-bit add, a multiplexer with storage, a 3-term AND, etc. The cell can communicate directly with its four immediate neighbors, as well as more globally via an arrangement of buses passing by the cell on each side. Flexible clocking of the flip-flops is made possible by clock drivers associated with each column of the array.

Cells may be grouped into macros or larger functions, such as a one-bit serial-parallel multiplier, which requires 6 cells and may be tiled to any desired length which fits within the array. Typical clock rates for a fully programmed array are in the 10-50 MHz range.

Associated with each cell are 16 bits of configuration state (not shown) and a number of logic gates to control the core logic in the cell and the connections to buses and

adjacent cells. The entire array configuration (or any part of it) can be changed in a few milliseconds by simply reloading this state from the host computer.

While an array of this kind can be flexibly configured to implement a wide variety of hardware functions, the price paid for this flexibility is high -- as much as a factor of 5 to 10 in area over a fully-custom circuit designed to perform any single function that the FPGA can perform. However, in many applications this is a desirable tradeoff, since the FPGA can be reconfigured at run time to accelerate many different operations.

Since FPGA arrays are homogeneous and more like memory arrays than microprocessors, it is hoped that they will, in time, become ubiquitous and produced in large volumes with cost declining faster than microprocessors.

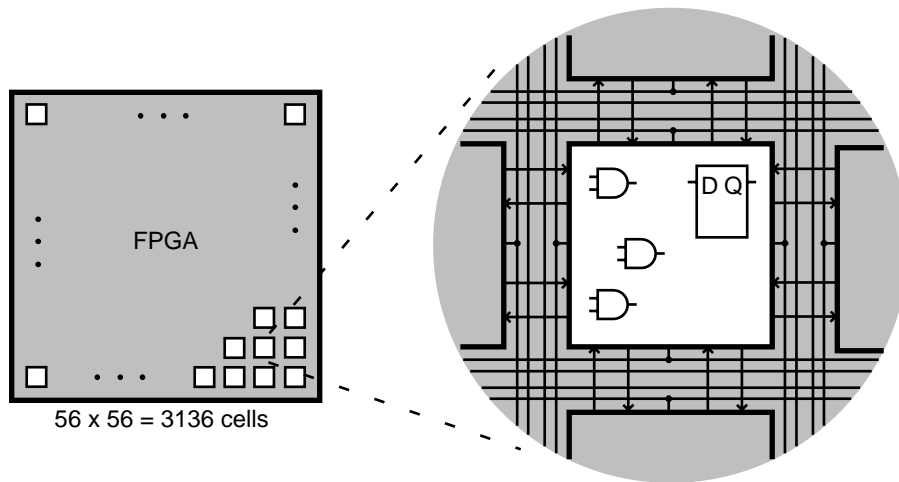


Fig. 1. FPGA integrated circuit

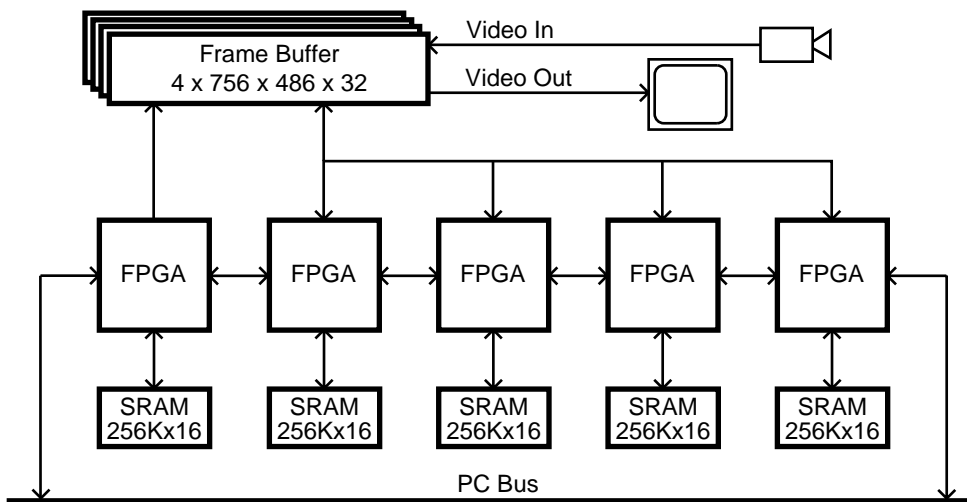


Fig. 2. System block diagram

B. System design

Fig. 2 shows a system configuration designed for real-time imaging applications. The main circuit board includes five interconnected FPGA chips, each with fast local SRAM, and a high-speed direct interface to frame buffer memory on an adjacent board. Video input and output are implemented on the frame buffer board [7].

With restructurable hardware, the designer can trade circuit complexity (number of cells used) for time, just as with software. For example, a multiply-accumulate function typically used in filtering can be implemented in bit-serial fashion [8] or in parallel in an FPGA array. In the array employed here, each multiplier bit uses 6 cells in a 2 x 3 block. Each ripple-carry adder requires 1 cell per bit of accumulation. Thus, if 8-bit multiplies are desired with 16-bit accumulation, about 36 serial-parallel multiply-accumulates will fit within each FPGA array (with a few cells left over for control, etc.) and each will produce a new value every 16 bit times. Alternatively, 4 fully-parallel multiply accumulates could be implemented, each producing a new value every clock cycle.

In highly-parallel arrays such as FPGAs, serial arithmetic is often advantageous, since arithmetic functions are quite compact and more instances of a function can operate in parallel.

An important issue in system design using FPGAs is the placement and connection of memory and I/O capability. In fact, this is clearly the least flexible part of the present system. *Field Programmable Interconnect Circuits* (FPICs) [9,10] can be used to somewhat alleviate this problem, but the dichotomy between large arrays of logic and large arrays of memory remains a fundamental barrier to truly reconfigurable designs.

C. Applications

In operation, typically one or more images are stored in the frame buffers via real-time video input or disk access. These images are accessed by the FPGA array using addresses generated in the first array. Data flows in parallel to arrays 2 through 5, which process four color components (Red, Green, Blue, Alpha) simultaneously using local SRAM for storage of intermediate results (e.g. line buffers, etc.), additional images, or tables, etc. Result images are returned to the frame buffers for subsequent display.

The host computer is a 486 PC running MS Windows and programmed in C. A typical use would be as an accelerator for custom filter plug-ins in programs such as Adobe Photoshop or Altamira Composer.

3. EXAMPLES

We show here two simple examples, each a common imaging function often needed in real-time systems. Each is a combination of simple arithmetic operations such as multiplication and addition, however the two functions have very different topologies, as well as different arithmetic lengths, and different memory and I/O requirements.

A. Alpha Blend

Alpha Blend is a well-known image compositing technique [11] which uses a transparency or opacity value alpha at each pixel to calculate a proportional mixture of foreground and background images. This computation is widely used in image compositing, overlaying, painting and drawing, text display, and various special effects such as fade, dissolve, and matting or chroma key.

Examples of the use of Alpha Blend are shown in Fig. 3, while the function is illustrated in Fig. 4. The computation is a simple proportional mix with alpha as the proportionality factor. Note that Alpha Blending preserves anti-aliasing of foreground images under composition.

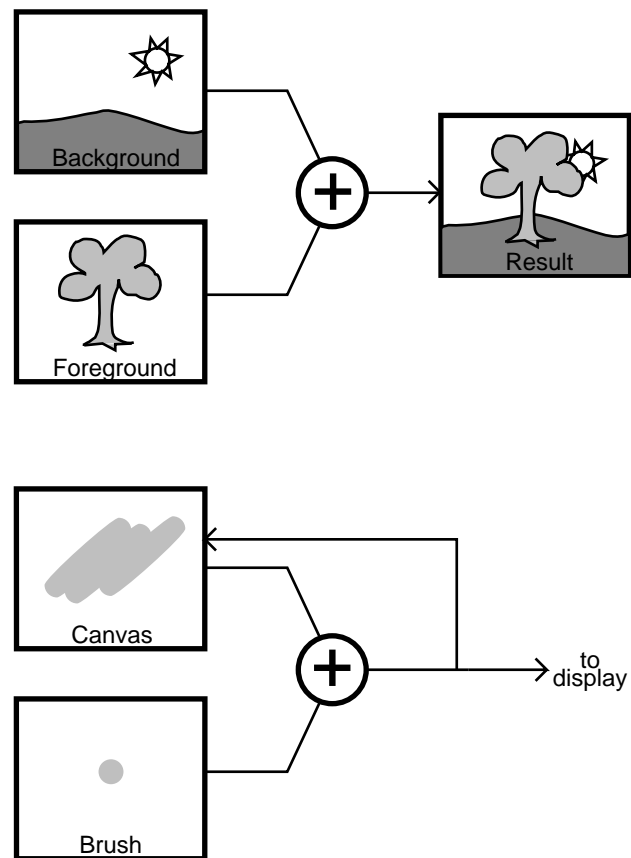


Fig. 3. Image composition and painting using the Alpha Blend function.

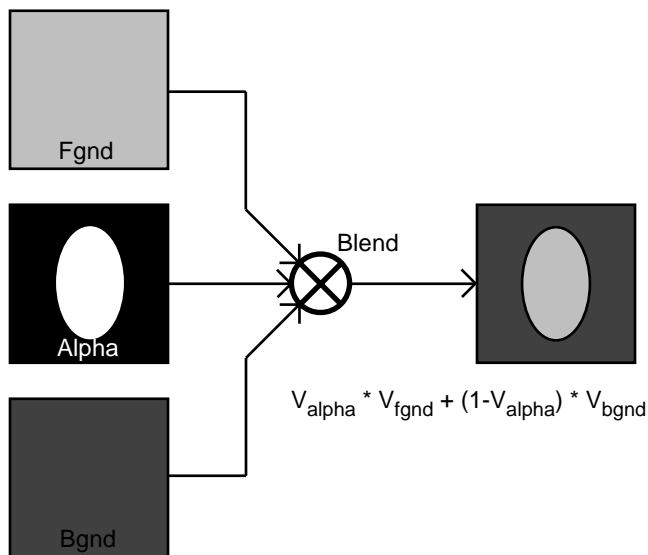


Figure 4 Alpha Blend function.

An implementation of this function is shown in Fig. 5 for simultaneous computation of four color components Red, Green, Blue, and Alpha. At every pixel, each color component is blended separately using the single foreground Alpha as the proportionality factor. The quantity 1-Alpha is

created by simply complementing Alpha. For the computation of Alpha itself, a constant value of "1" (fully opaque) is substituted. Obviously, the multiplier in this case is superfluous, but is included here for consistency.

It may also be noted that a simple rearrangement of terms gives

$$V_{\alpha} * (V_{fgnd} - V_{bgnd}) + V_{bgnd}$$

and thus, allowing for two's complement arithmetic, a simpler circuit with only one multiplier and two adders. However, the more straightforward version is illustrated here for simplicity.

It is important to emphasize how easily different configurations and algorithms can be tried using a reconfigurable hardware implementation. Just as with software, an algorithm may be implemented first in the most direct way, and later improved by optimization.

The pipeline shown in Fig. 5 occupies about one fourth of an FPGA using 8-bit serial-parallel multipliers with 16-bit intermediate results. If the array is clocked at 16 MHz., the pipeline computes all 8 multiplies and 4 adds once each microsecond. Real-time video rates (e.g. 13.5 MHz * 4 components * 3 ops = 162 MIPS) are readily achieved by operating several copies of the pipeline simultaneously. In order to provide appropriate memory access, these copies can be distributed across several FPGA chips.

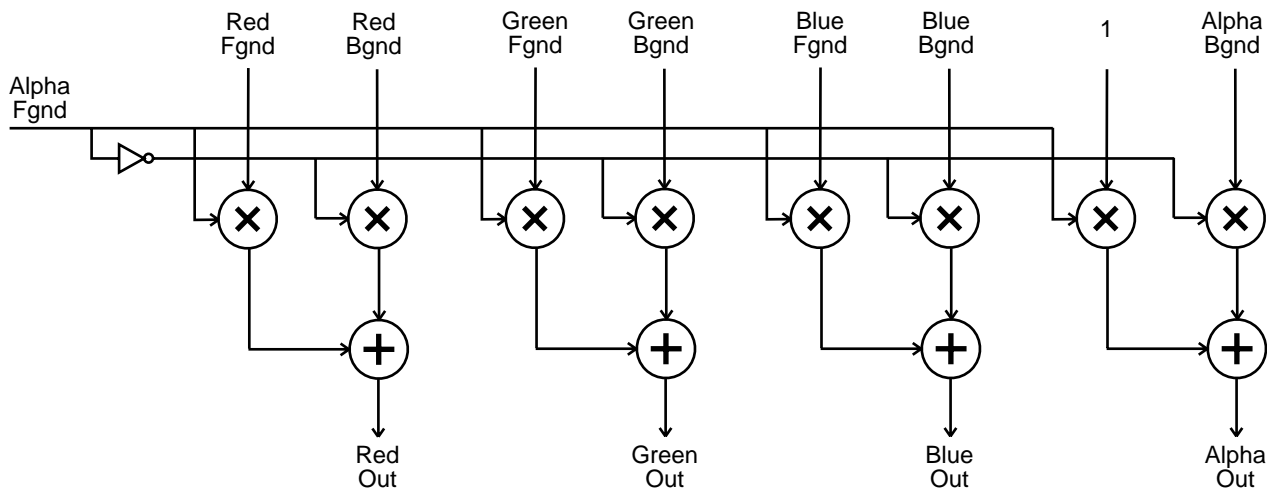


Figure 5. Alpha Blend pipeline.

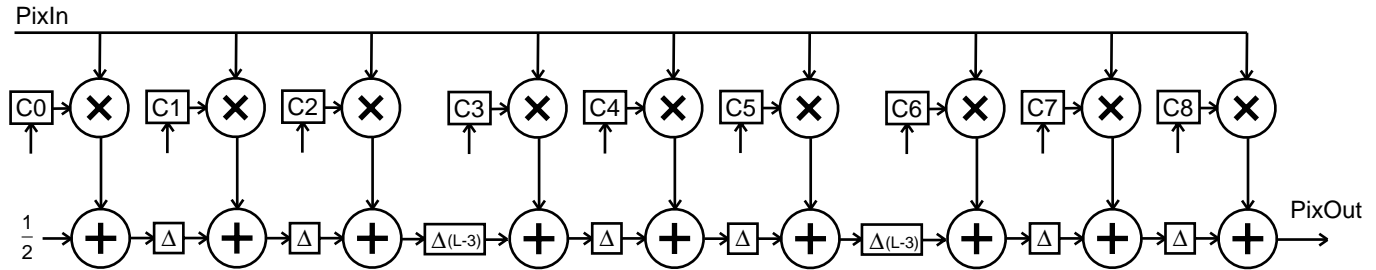


Fig. 6. Convolution pipeline.

B. Convolution

Convolution is a standard image processing function often used for filtering (sharpening, blurring, noise removal) or resampling (compositing, scaling, rotation, warping), etc. [1]. The function at each pixel is simply a weighted sum of surrounding pixels given by

$$V(x,y) = \sum_{i,j} C(i,j) * V(x+i,y+j),$$

where C represents the convolution filter kernel, usually a small two-dimensional array of weights. If the filter size is $n \times n$, then each output pixel depends on n^2 adjacent pixels, and thus n^2 multiplies and adds are required. The topology and computational structure of this function is substantially different from the previous example.

Fig. 6 shows a pipeline implementing a 3×3 convolution filter. Each input pixel is simultaneously multiplied by all 9 filter coefficients. Partial sums are then accumulated by the adder chain, with a delay of one pixel between adjacent adds and a delay of the image line length minus 3 pixels between rows to allow proper alignment. The partial sum is accumulated at twice the precision of the input values to avoid loss. By initializing the sum to $1/2$, rounding of the result takes place automatically. Note that each input pixel is accessed only once. Output pixels may be rewritten directly into the image memory as no double buffering is necessary.

In our implementation of this pipeline, 8-bit pixel values, 8-bit two's complement coefficients, 16-bit sums, and serial-parallel multipliers are used. Faster operation could be easily achieved by using fully parallel multipliers, at the expense of more circuitry.

By parameterizing the pipeline size, different filter kernels can be generated at run time for variable scaling or for adaptive filtering applications. A similar pipeline structure may be used for other functions such as morphology, correlation, pattern matching and feature extraction.

C. Other functions

Many other functions with different characteristics are possible and, more importantly, several can be used during a single application program. For example, a particularly

dense real-time motion estimation algorithm is given by Furtek [12], using the same FPGA as above to implement a large number of simultaneous differences. Also see [13] for numerous examples of FPGAs in other custom computing applications.

4. CONCLUSIONS AND FUTURE DIRECTIONS

We have discussed the use of restructurable hardware in the implementation of reconfigurable image processing systems and shown how high performance can be achieved over a broad range of functions.

An important area for future research is in the integration of programmable logic processing and memory. Different applications often require different amounts of memory connected variously within the computation. While FPGAs allow very flexible reconfiguration of processing elements, a corresponding distribution of memory and other I/O is often much more difficult. This problem is at present only partially overcome with the use of programmable interconnections.

Creation of new designs is presently done using available CAD tools such as schematic editors and interactive place and route software. Textual languages such as VHDL can also be used. However, these tools are adequate and appropriate only for those users already somewhat familiar with digital engineering. Much higher-level tools will be necessary before the adoption of restructurable hardware becomes widespread.

ACKNOWLEDGMENTS

The author is indebted to National Semiconductor and Concurrent Logic Incorporated for their assistance in this project.

REFERENCES

- [1] Pratt, W.K., *Digital Image Processing*, 2nd ed., Wiley, 1991.
- [2] Lin, K., Frantz, G.A., and Simar, R., "The TMS320 Family of Digital Signal Processors", Proc. of the IEEE, Vol. 75, No. 9, Sept. 1987. Reprinted in *Digital Signal Processing Applications with the TMS320 Family*, Vol. 3, Texas Instruments, 1990.
- [3] Wahlstrom, S.E., "Programmable logic arrays - Cheaper by the

- million", *Electronics*, Dec. 11, 1967.
- [4] Shoup, R.G., "Programmable Cellular Logic Arrays", Ph.D. dissertation, Carnegie-Mellon University, 1970.
- [5] The Programmable Gate Array Data Book, Xilinx Inc., San Jose, California, 1993.
- [6] CLi6000 Series Field-Programmable Gate Arrays, Concurrent Logic, Inc., Sunnyvale, California, 1992.
- [7] Horizon 1280 Manual, Truevision, Indianapolis, Indiana, 1992.
- [8] Lyon, R.F., "Two's Complement Pipeline Multipliers", *IEEE Trans. on Communications*, April, 1976.
- [9] Aptix Data Book, Aptix Corp., San Jose, California, 1993.
- [10] FPIDPro User's Guide, I-Cube Design Systems, San Jose, California, 1992.
- [11] Foley, J.D., van Dam, A., Feiner, S.K., Hughes, J.F., *Computer Graphics, Principles and Practice*, 2nd ed., Addison-Wesley, 1990.
- [12] Furtek, F., "A Field-Programmable Gate Array for Systolic Computing", in *Proc. 1993 Symp. on Research on Integrated Systems*, MIT Press, 1993.
- [13] Buell, D.A., and Pocek, K.L., eds., *Proc. 1993 IEEE Wkshp on FPGAs for Custom Computing Machines*, IEEE Computer Society Press, April 1993.