

3x3 Convolver with Run-Time Reconfigurable Vector Multiplier in Atmel AT6000 FPGAs

Introduction

Convolution is one of the basic and most common operations in both analog and digital domain signal processing. Often times, it is desirable to modulate a given signal in order to enhance or extract important information contained in it. This signal modulation is generally known as filtering. In two-dimensional digital signal processing (2-D DSP), a 3x3 convolver is commonly used in applications such as object detection and feature extraction of 2-D images. However, due to the compute-intensive nature (multiply-add operations) of 2-D convolution, the size of the required digital logic circuitry increases accordingly. In this application note, we present an efficient single-chip FPGA implementation of a bit-parallel 3x3 symmetric convolver that features run-time software-configurable convolver coefficients (taps). Our reference design takes full advantage of time-division multiplexing (TDM), pipelining, CacheLogic®, and vector multiplication. We will also discuss an alternative to using CacheLogic, which features run-time hardware self-configurable LUT using static random access memory (SRAM).

Convolution Basics

Unlike the “flip-and-drag” procedure performed in the analog-domain convolution, the 2-D digital-domain convolution is much easier to perform from a graphical standpoint. Given two 2-D infinite-impulse arrays $a(m,n)$ and $h(m,n)$, where m and n are integer indices, the convolution sum $p(m,n)$ is defined as:

$$p(m,n) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} a(i,j) \cdot h(m-i, n-j) \quad [1]$$

Graphically, when we convolve $a(m,n)$ with $h(m,n)$, we multiply each tap in $a(m,n)$ with the corresponding tap in $h(m,n)$. The convolution output is then the sum of all multiplies. A 3x3 finite-impulse symmetric convolver example is shown below. $H(m,n)$ denotes the constant convolver mask. The underlined tap denotes the origin of an array.

Given:

$$a(m,n) = \begin{bmatrix} 1 & 2 & 4 \\ 0 & \underline{2} & 1 \\ 1 & 0 & 3 \end{bmatrix} \quad h(m,n) = \begin{bmatrix} 1 & 0 & 1 \\ 0 & \underline{2} & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

Then,

$$p(0,0) = \underline{13}$$

In the above example, our output origin has a value of $13 = (1 \times 1 + 2 \times 0 + 4 \times 1 + 0 \times 0 + 2 \times 2 + 1 \times 0 + 1 \times 1 + 0 \times 0 + 3 \times 1)$. Other entries of the output array will be computed as $a(m,n)$ slides across $h(m,n)$ (i.e., when the origin shifts in $a(m,n)$.)

An important property of a symmetrical 3x3 mask is that it allows for “pre-addition” of certain input values before any multiplication takes place. As in the above example, notice that we have only three constants in $h(m,n)$. Instead of performing nine multiplications, we can reduce the number to three by pre-adding the input values that get multiplied to each of the three mask constants. We get the same result of $13 = ((1 + 4 + 3 + 1) \times 1 + (2 + 1 + 0 + 0) \times 0 + 2 \times 2)$.

(continued)

AT6000 FPGAs

Application Note

Although the number of multiplication is greatly reduced, a single-chip FPGA ($\approx 10K$ usable gates) implementation is impossible with fully pipelined multipliers due to their fairly large size. In view of this problem, vector multipliers have been proposed to provide an alternative to regular multipliers.

Vector Multiplier - Theory of Operation

Suppose we need to perform the following:

$$P = \sum_{i=1}^p a(i)h(i) \quad [2]$$

where the $a(i)$ (input mask taps) and the $h(i)$ (constant convolver taps) are 2-bit positive integers. Let:

$$\begin{aligned} a(1) &= 01 & h(1) &= 11 \\ a(2) &= 11 & h(2) &= 00 \\ a(3) &= 10 & h(3) &= 10 \end{aligned}$$

By convention, the multiply-add operation takes place in a "top-down" fashion. First, we compute all stages of partial products of multiplying two numbers. Then we sum all partial products with appropriate arithmetic shifts to obtain each product. We then repeat the same procedure for all other multiplies. Finally we sum all products to get the output (see figure 1 below).

Figure 1. Conventional multiply-add operation

Constants $h(n)$	11	00	10
Inputs $a(n)$	<u>x 01</u>	<u>x 11</u>	<u>x 10</u>
	11	00	00
	<u>00</u>	<u>00</u>	<u>10</u>
	011	000	100

$P = 011 + 000 + 100 = 111$

Another approach, which leads to the theory of vector multiplication using LUT (Look-Up Tables), is to consider summing partial products of the same bit-level (see figure 2). The output is then the sum of all partial product stages $P1, P2, \dots, Pn$ with the appropriate arithmetic shifts.

Figure 2. Alternative multiply-add operation

constants $h(n)$	11	00	10
inputs $a(n)$	<u>x 01</u>	<u>x 11</u>	<u>x 10</u>
partial prod. P1	= 11 +	00 +	00 = 11
partial prod. P2	= <u>00</u> +	<u>00</u> +	<u>10</u> = 10

$P = P1 + P2 = 11 + 100 = 111$

Notice that when forming the partial products, we sum copies of the constant multiplicand $h(n)$. For example, when forming partial product P1, we bring down a copy of $h(n)$ when the least-significant bit (LSB) of the corresponding $a(n)$ is 1. If the LSB is 0, we simply bring down a zero. As a result, for the LSB combination of 110 (shown in figure 2 as bold numbers), we perform:

$$P1 = h(1) + h(2) + 0 = 11$$

The same procedure applies when forming P2. We look at the second LSBs of the $a(n)$. For the bit combination 011, we form:

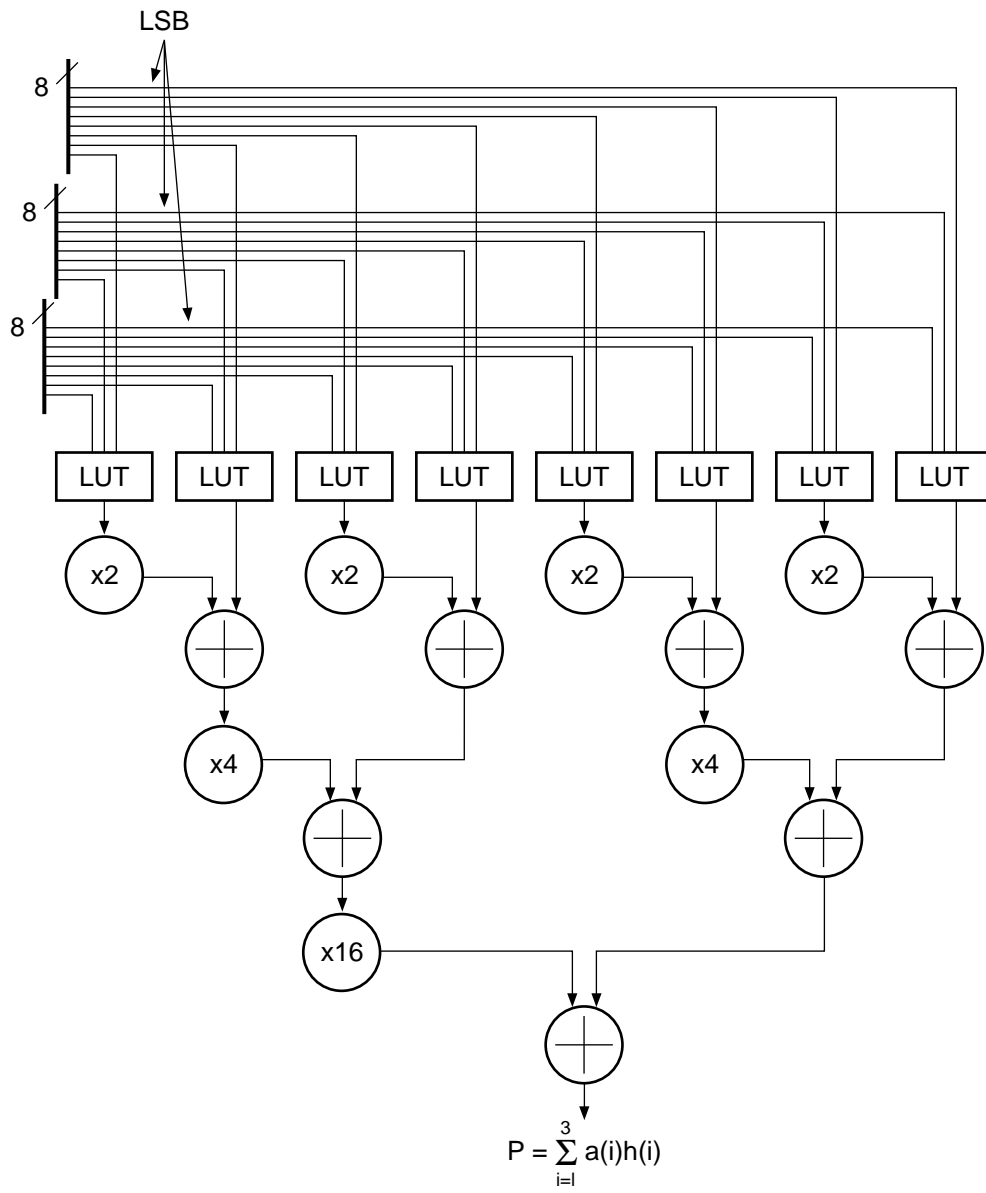
$$P2 = 0 + h(2) + h(3) = 10$$

Finally, we arithmetic left-shift P2 by one bit and add to P1 to get $P=111$. Taking advantage of this combinatorial property of the input bits, we can pre-compute all possible combinations (see table 1) and store them into a LUT. When performing vector multiplication, we then simply call the LUT for the specific input bit combination, and accumulate the LUT output with the appropriate arithmetic bit-shift(s). For reference, a three 8-bit input vector multiplier is shown in figure 3.

Table 1. Look-up table

LSB Combination	Operation	Partial Product P
000	0+0+0	0
001	0+0+h(3)	10
010	0+h(2)+0	0
011	0+h(2)+h(3)	10
100	h(1)+0+0	11
101	h(1)+0+h(3)	101
110	h(1)+h(2)+0	11
111	h(1)+h(2)+h(3)	101

Figure 3. Three 8-bit input vector multiplier



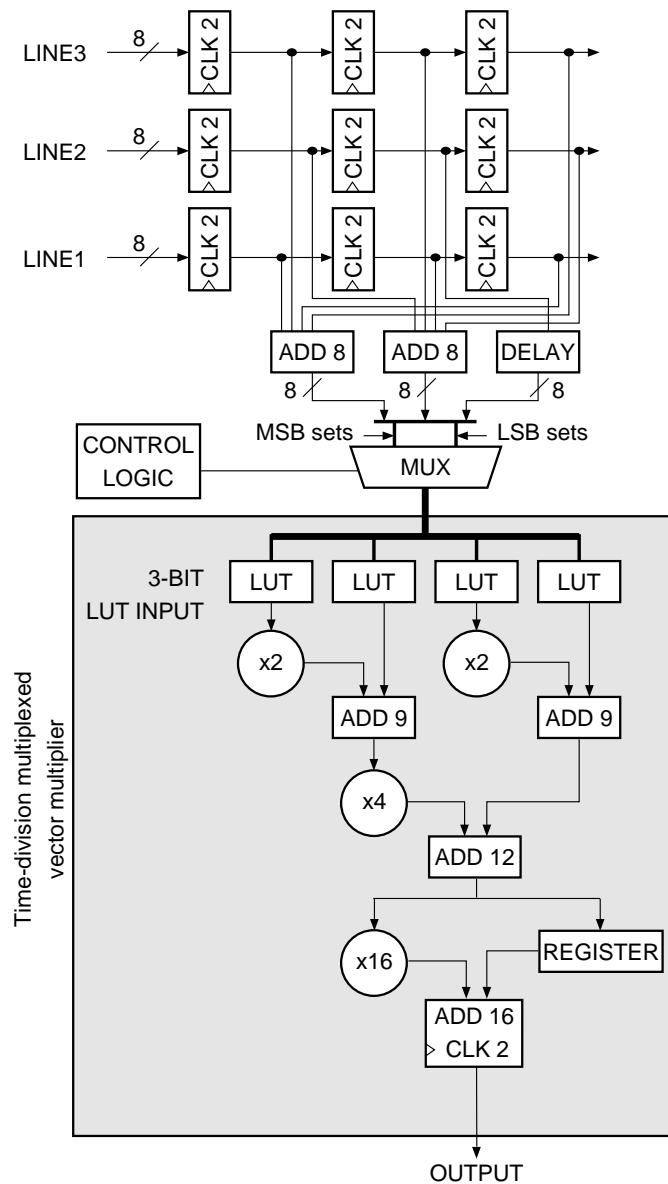
Implementation

Our reference design assumes operation on 256-level grayscale images whose intensity values are unsigned 8-bit integers. Same assumption holds for the convolver mask taps for simplicity. The precision and the choice of number system are application specific, and are customizable with additional logic circuitry. An overall block diagram of our 3x3 symmetric convolver design is shown in figure 4.

Input registers and Pre-addition

Three 8-bit parallel datastreams of the digitized image/video frame are clocked into the 3x3 registers matrix every two global clock cycles (CLK_2). This clocking is essential for the TDM vector multiplier. Fully pipelined adders are used to pre-add inputs, as described above. The adders can either run on the global clock or the CLK_2 signal. All logic blocks are generated by Atmel IDS Component Generator.

Figure 4. Block diagram for 3x3 symmetric convolver using TDM vector multiplier



LUT

Depending on the system performance requirement, three choices are available for the LUT implementation: ROM with CacheLogic, mux-constant with CacheLogic, and on-chip SRAM.

ROM implementation

The CacheLogic software allows the user to change convolver mask values at run-time. Partial-Product ROM generation software computes the necessary combinations to form the new LUTs. The CacheLogic software then partially reconfigures the Atmel FPGA and downloads the LUT bitstreams without any effect on existing circuitry and register values. Figure 5 shows the architecture of our reference design using ROMs as LUTs.

Mux-constant implementation

A faster system can be realized if mux-constant combinations are used to emulate the ROMs. Shorter reconfiguration time is achieved at the expense of slightly more complex routing. Figure 6 shows the architecture of our TDM convolver with mux-constant ROM emulation.

SRAM implementation

As seen in Table 1, the LUT entries are simply the sums of different combinations of the constant mask coefficients. We can thereby configure our hardware to self-generate the LUT entries by using a counter to generate the combinations. Since we have three constant coefficients, we have a maximum of eight combinations of additions.

Figure 5. Architecture of TDM convolver with ROMs as LUTs

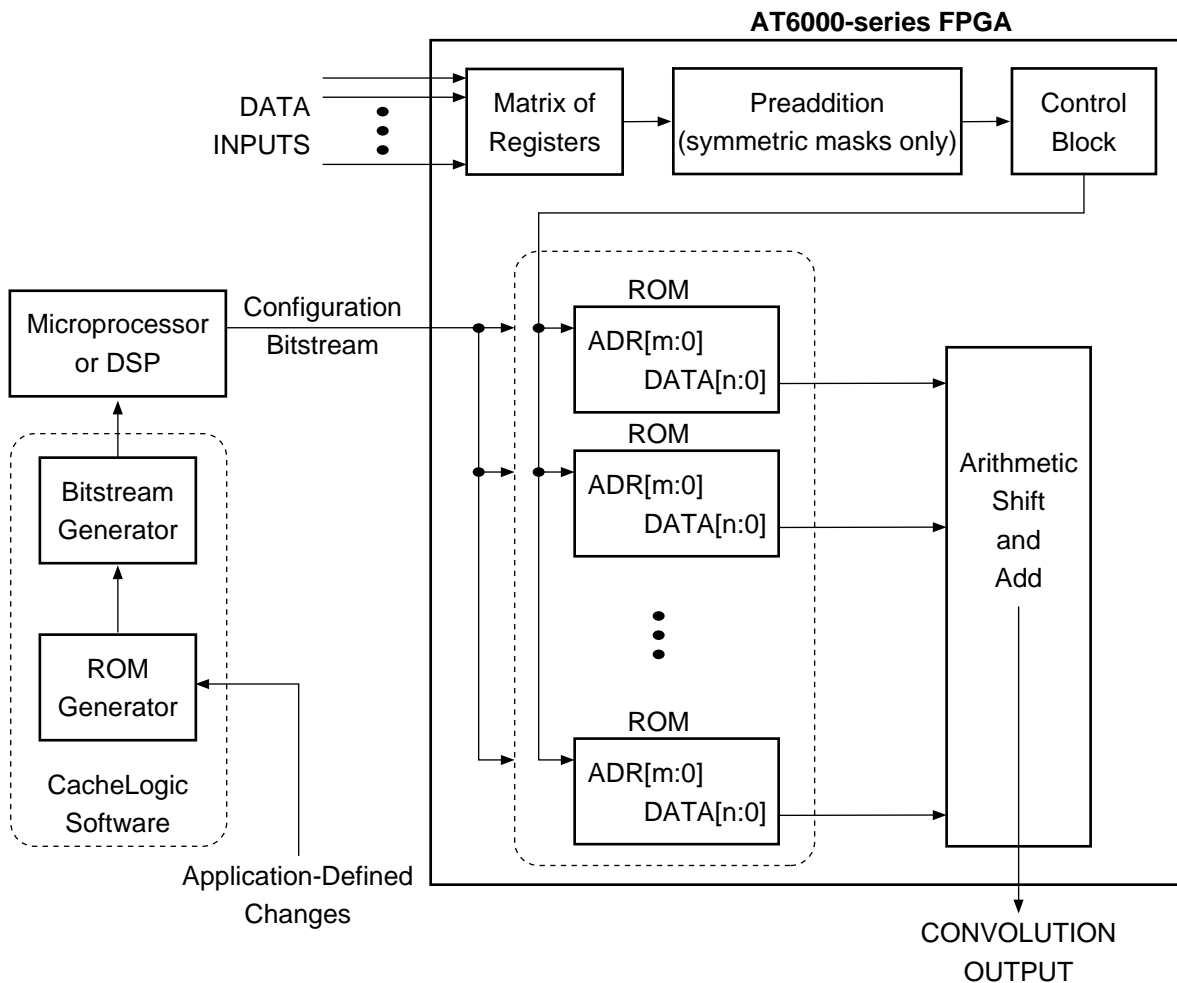


Figure 6. Architecture of TDM convolver with mux-constant ROM emulation

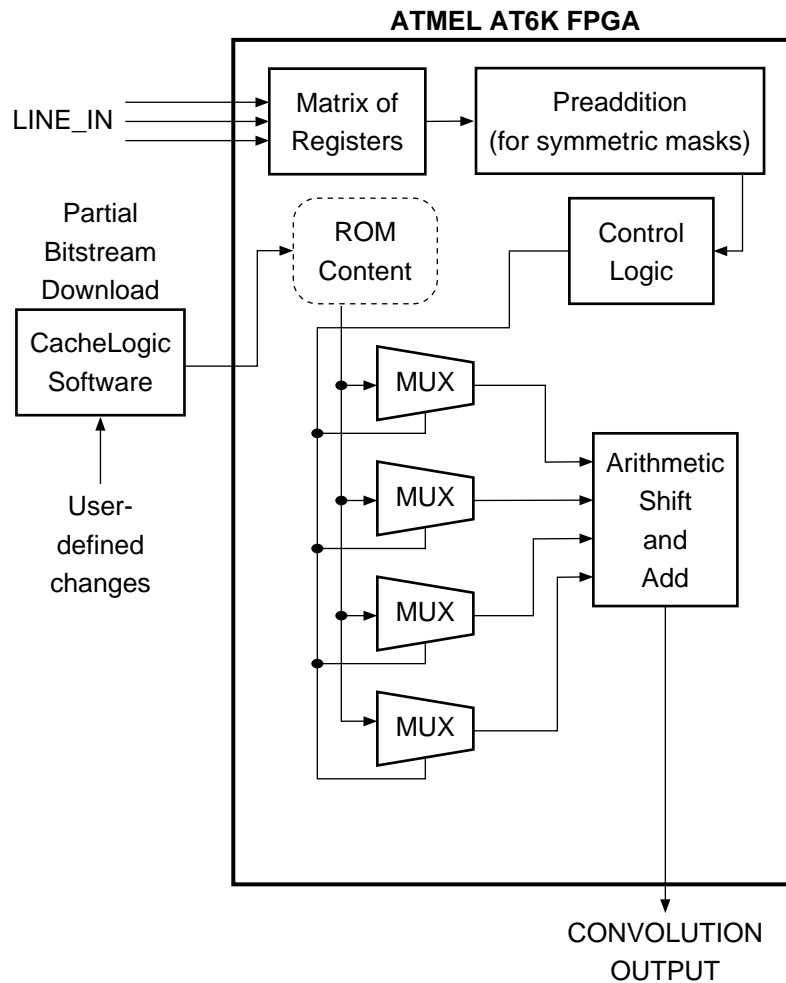
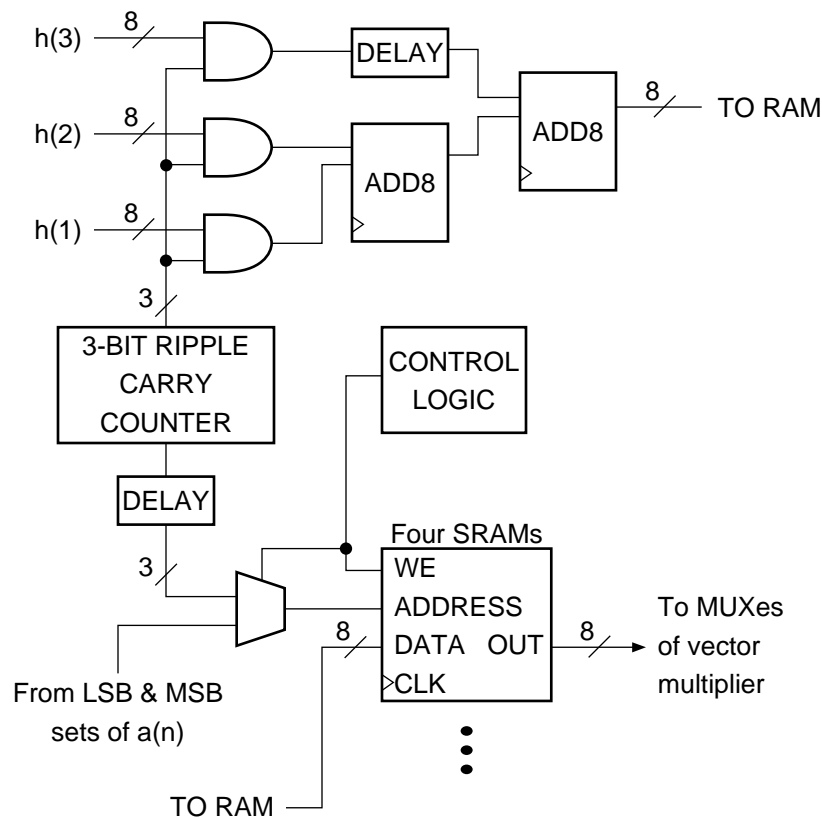


Figure 7 shows the LUT generator. A 3-bit ripple-carry counter is used to generate the combination signal. The AND gates are to “gate in” the coefficients to the adder stages. All adders are fully pipelined. The delay element is to compensate for the latency due to pipelining. Once the additions are performed, they will be stored in the four dual-port SRAMs that emulate the LUTs. Various control logic blocks will determine the activation of certain logic stages with precise timing. Note the address lines of the SRAMs. During the LUT configuration stage, the 3-bit ripple-carry counter will drive the address lines of the SRAMs. When the actual convolution is performed, the LSB and MSB combinations of the $a(n)$ will read the SRAMs for pre-computed partial products.

TDM vector multiplier

The TDM vector multiplier shown in Figure 4 operates on the same principle as the one shown in Figure 3. The only difference is the number of LUTs used. In our design, we split the look-up operation into two parts. We first operate on the four LSB sets of $a(n)$, performing the necessary shift and add. Then we repeat the same procedure for the remaining four MSB sets of $a(n)$. Notice the register immediately before the final adder stage. This register holds the result of the LSB operation so that the final addition is synchronized with the MSB sets of the same inputs $a(n)$. Again, this stage of the design is fully-pipelined, and runs on the global clock unless otherwise specified on Figure 4.

Figure 7. LUT generator



Applications

Convolvers, cellular automata, and neural nets all share the feature that the value of the pixel, cell, or neuron is replaced by a value dependent on its current value and the values of its neighbors. In 2-D, both MxN sequential convolvers and MxN CA processors need identical dataflow capabilities that are best implemented with row buffers and pipeline registers. If these neighborhood operations seem simple and undemanding, consider that a 5x5 convolution requires at least 60 operations: 25 multiplies, 25 additions, four row-buffer writes, and four row buffer updates. At 13.5 MHz (D1 Video) this results in 810 million operations per second. DSPs simply do not have the required bandwidth to do this sort of imaging in real-time.

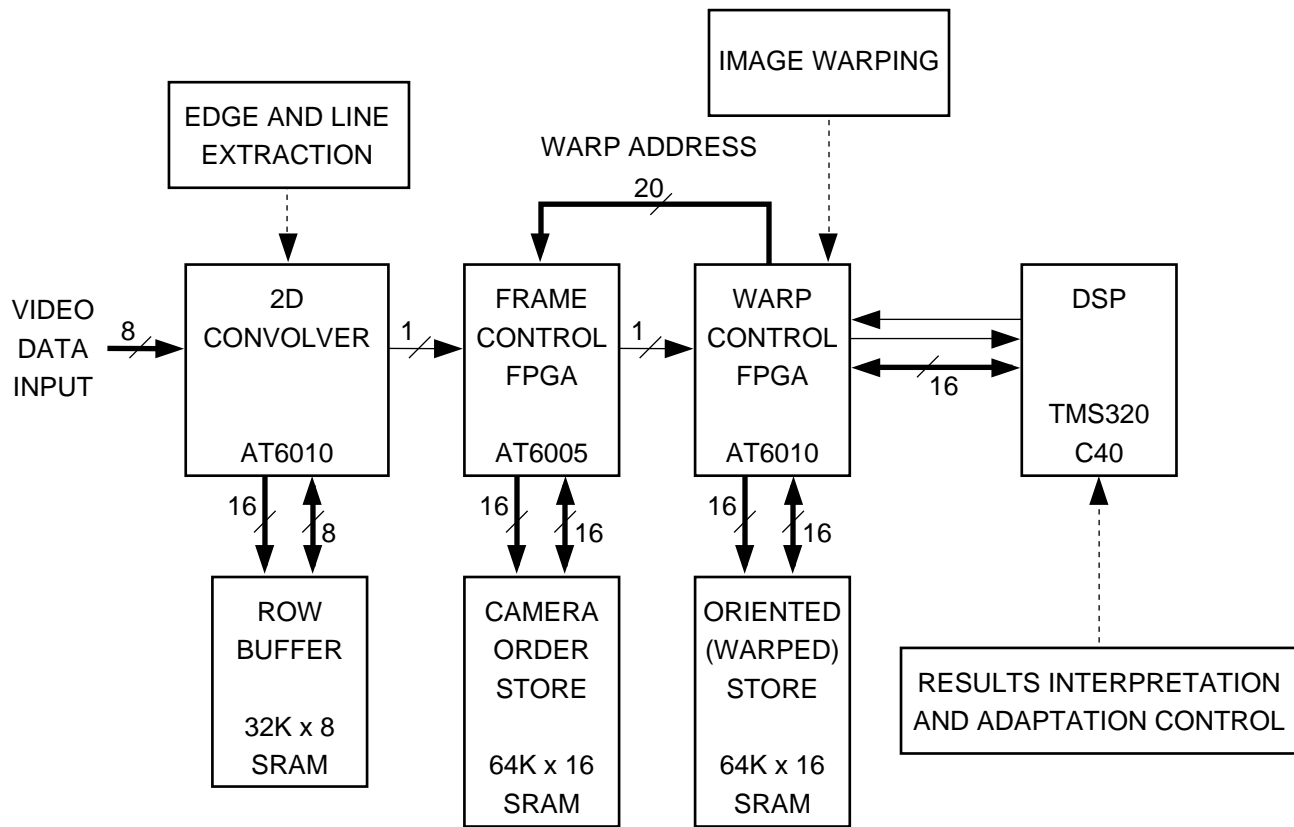
What the DSP chip can provide in a real-time imaging design is adaptive control. For example, in a system with an MxN convolver that performs low-pass filtering on the input image, if the objects of interest are out of focus then the low-pass filtering has to be adjusted or eliminated. If the low-pass filter is followed by a high-pass filter then the high-

pass filter's characteristics may also need to be adjusted. The DSP chip is used to interpret the results and make the adjustments, either by reloading coefficients or by reconfiguring the FPGAs.

In the real-time imaging system shown in figure 8, the DSP relies on the FPGAs to perform the actual imaging tasks. As a 2-D image analyzer, FPGAs do the feature extraction, and a DSP chip performs result implementation and computation of convolver coefficients.

Semi-real-time image processing is very similar to real-time image processing except for some avoidance of parallelism and duplication of resources. A semi-real-time system might include freeze-frame, followed by a warper, followed by a convolver. The image is processed with successive warps on the same frozen image with all operations in real-time. If the final warp is arrived at through successive refinement with no more than eight passes then the system is running at 1/8th real-time.

Figure 8. A Real-Time Image Processing System



Conclusion

A TDM 3x3 symmetric convolver using a vector multiplier fits into an Atmel AT6010 FPGA. Taking advantage of full-pipelining and CacheLogic, a very flexible design is

achieved. Vector multipliers are now run-time reconfigurable. Table 2 shows some statistics of our reference design.

Table 2. Reference design statistics

LUT Implementation	ROM	RAM	Mux-Constant
Datapath	TDM Pipelined	TDM Pipelined	TDM Pipelined
Input Precision	8-bit	8-bit	8-bit
Partial Reconfiguration	CacheLogic	CacheLogic/LUT Generator	CacheLogic
Partial Reconfiguration Time	0.2 μ s per cell	0.2 μ s per cell	0.2 μ s per cell
Throughput Frequency (approx.)	15 MHz	15 MHz	25 MHz

References

- [1] Anil K. Jain, "Fundamentals of Digital Image Processing," Prentice-Hall, Inc., 1989.
- [2] Lee Ferguson, "Image Processing Using Reconfigurable FPGAs," DSP and Multimedia Technology, May/June 1996, Golden Gate Enterprises, Inc.