Designing a halftoning coprocessor

Anders KUGLER, Roger D. HERSCH

Laboratoire de Systèmes Périphériques,
Swiss Federal Institute of Technology, Lausanne (EPFL), Switzerland.

May 1993

Abstract

Halftoning is a fairly slow process when executed by software on conventional processors. To speed up halftoning, a halftoning algorithm has been developed and integrated into a dedicated hardware architecture. This paper describes the implementation of the architecture with a XILINX Field Programmable Gate Array (FPGA) and compares its performances with results obtained by a software implementation. A discussion on how to improve the present architecture concludes the paper.

Introduction

Generating a binary image from a gray scale image in such a way that the resulting image gives the impression of having gray scale tones is called halftoning. In the past several halftoning techniques have been developed. For producing regular periodic screen elements, dithering algorithms allow generating both dispersed and clustered screen cells [Hou83]. The halftoning process which is developed in this paper is based on a regular grid of dither cells. Each cell contains N ordered binary pixels and offers N+1 levels of gray.

Transforming a gray scale image into a binary image is slow due to the very large amount of pixels involved. For each binary pixel in the destination image, the gray level of the corresponding pixel in the source image has to be compared with the threshold level of the corresponding pixel in the dither cell [Foley90]. It is possible to precompute the screen cells for each gray level. Then, to each binary pixel in the destination image corresponds one pixel of the screen cell which represents the gray level of the current pixel in the source image. This method is faster than comparing for each binary pixel a prestored dither threshold value with a gray level, because several binary pixels of the destination image are generated at once.

Two halftoning algorithms have been developed previously: *forward mapping* and *backward mapping*. In the forward mapping algorithm, each gray pixel in the source image is taken and the corresponding binary pixels of the destination image are filled. The second method considers each binary pixel in the destination image, from left to right and from top to bottom and fills it according to its corresponding gray pixel in the source image. *Backward mapping* can be executed word by word in one pass through the destination image. Both of these methods are thoroughly discussed in [Morgan93]. It turns out that *backward mapping* is faster than *forward mapping*.

In order to decrease the time of halftoning we propose a hardware architecture including a FPGA halftoning coprocessor.

Proposed hardware architecture

The proposed hardware architecture (Figure 1) is composed of a microprocessor, dynamic memory, a fast static memory and a FPGA integrated circuit (IC). The memory contains the source image and will receive the destination image bitmap. The memory is shared between a transputer (T800) and an application specific FPGA circuit. A fast static memory (256K 16 bit-wide words, 25 ns) was added to reduce access time for retrieving the screen cells.

The application specific integrated circuit (ASIC) is a XILINX 4010 Field Programmable Gate Array and contains the application which was designed to increase the speed of the halftoning algorithm.

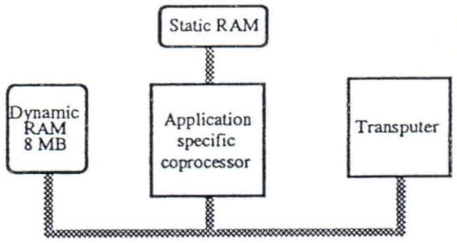

Fig. 1 - General architecture

The advantage of using a XILINX FPGA is that it is reconfigurable for any other application which requires the same general architecture. Currently, the 4010 is the biggest FPGA available from XILINX. It offers 400 configurable logic blocs (CLB).

Most operations (reading the source image, generating the addresses of the screen cells, retrieving the corresponding screen element, shifting and masking the output word) are executed in parallel. The operations are controlled by a state machine and the boundaries of the gray pixels in the destination image are computed on the fly. Such operations are implemented using incremental methods.

At each memory access cycle, four gray scale pixels (4 bytes) from the source image are read and stored

in a buffer. The address of the screen cell in the screen cell table is computed and the screen element (16 bits) is retrieved from the static RAM. The screen element is shifted and the valid bits are masked. A bitwise OR between the shifted screen element and the previous result is performed, before storing the final destination image word in memory.

A sequencer coordinates the flow of the main operations with the rest of the computations (updating pointers to the screen cell boundaries, etc...) and controls the write and read operations to the shared memory. A certain number of registers (8-bit and 12-bit wide) contain local values or values initialized directly by the microprocessor. An 8-bit wide control register is directly accessible and interfaces the application specific circuit with the program running on the host processor. Before starting the algorithm, the microprocessor writes initial values in the registers of the ASIC. Figure 2 shows the proposed hardware architecture.

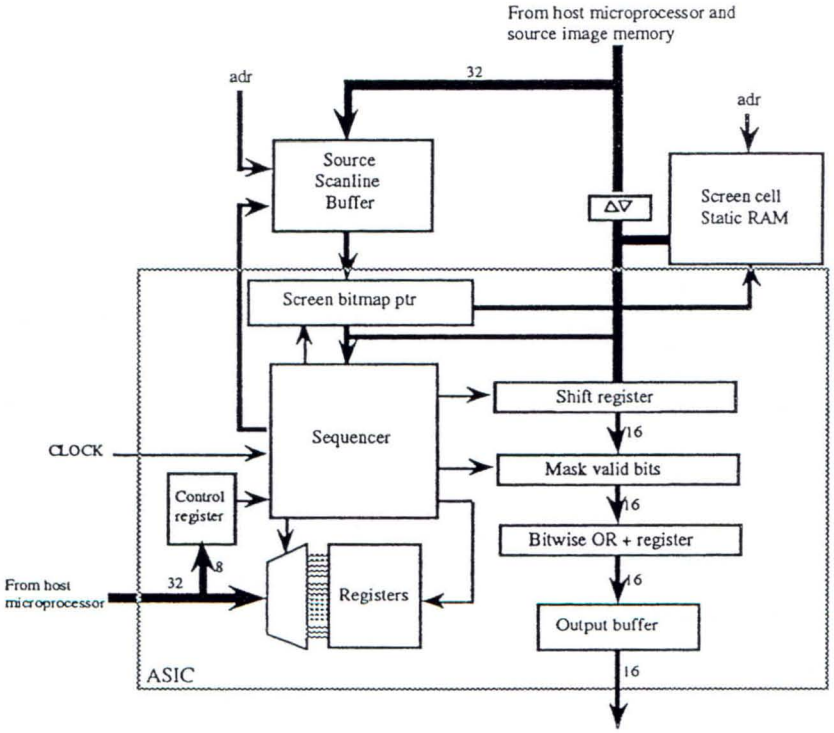

Fig. 2 - The proposed hardware architecture

The backward mapping algorithm and its hardware implementation

The backward mapping algorithm contains three loops. The outer loop executed by the halftoning coprocessor scans the source image lines. The inner loop reads each gray pixel of the current line in the source image and computes the corresponding bitmapped screen element. Only the two inner loops are executed by the IC. The outermost loop, which scans the source image pixels, is accomplished by the microprocessor because it requires computations which are difficult to implement by hardware. When it comes to continue with the next line in the source image, the microprogram in the IC interrupts the microprocessor and waits for a signal to restart with a new cycle.

The algorithm, as executed in the circuit, was taken from the original backward mapping algorithm and split into several steps. Each step is carried out during a cycle time, which is chosen as short as possible. The sequencer controls the flow of these steps. The cycle time of the steps is limited by the worst-case speed of the circuit. A cycle needs to be long enough to allow the new values to propagate from one register to the next one through the logic between the registers (adders, comparators, multiplexers...).

The backward mapping algorithm is segmented into parts which are integrated in the ASIC. How the algorithm was split into different stages is explained below. Some of the values in registers are initialized directly by the microprocessor, before the start of the application. The static RAM is filled with the computed screen cells once, and can then be used for halftoning the full image.

One scan line of the binary destination image after another is generated. This algorithm can be summarized with the following pseudo-code. The numbers refer to the stages of the sequencer. On the left hand side are the number of cycles, necessary to carry out the computations at each step.

T_i : microprogram instruction cycle time, equal to the clock cycle time

T_S : time for the shift operation

T_{rw} : time for a read or write operation

Halftoning by backward mapping

		FOR EACH scan line in the source image DO *this outermost loop is done by the microprocessor*
$1\ T_i$	**1**	FOR EACH corresponding row in the destination image DO • calculate the pointer to the corresponding grayscale pixel location (XAdr, YAdr) mapped into the output bitmap
$\frac{1}{4}\ T_{rw}$	**2**	FOR EACH gray level pixel in the current row of the source image DO • if necessary, retrieve the gray level (Gray) of the next four pixels from memory
$1\ T_i$	**3**	• calculate the address (DataPixAdr) of the corresponding screen element • calculate the position (LeftCol) of the next gray pixel mapped into the output bitmap
$1\ T_s + 1\ T_i$	**4**	• read the word from the screen cell table • shift the word to align the screen cell with the binary output word (DataPixShift)
$1\ T_i$	**5**	• mask the valid bits in the word to avoid overwriting the binary pixels associated with the previous gray pixel • OR the result with the previous result • update the pointer (NextLeftCol) to the destination bitmap for the next word REPEAT
$1\ T_{rw}$	**6**	• store the output image word to the destination bitmap memory • update the address (DataPixAdr) of the current screen element
$1\ T_s + 1\ T_i$	**7**	• read the next word from the screen cell table • shift the word to align the screen cell with the binary output word • update the pointer (NextLeftCol) to the destination bitmap for the next word UNTIL the position of the next grayscale pixel mapped into the destination bitmap is reached
$1\ T_i$	**8**	• increment the position (LeftCol) and continue with the next grayscale pixel in the current row of the source image END FOR
$1\ T_{rw}$	**9**	• mask the last word of the destination image scan line if necessary • write that last word to memory • increment the destination image row counter and continue with the current scan line of the source image
	10	END FOR • send an interruption to the microprocessor to continue with the next scan line of the image
		END FOR

The sequencer generates the different stages of the algorithm, and each stage takes a defined cycle time. A stage starts on the falling clock edge and ends with the next falling clock edge. On the rising clock edge, between these two limits, precomputed values are latched in registers. Some of the operations of the algorithm require more cycles and the corresponding stages take more time than just one cycle.

The main registers, necessary to calculate the address of the bitmapped screen cells, are shown in figure 3. Figure 4 details how the address of the screen element is computed arithmetically from the screen cell boundaries (XAdr, YAdr), the gray level (Gray), the screen cell tiling size (RectH, RectW, DispX) and the position (LeftCol, NextLeftCol) of the source image pixels mapped into the output bitmap [Morgan 92].

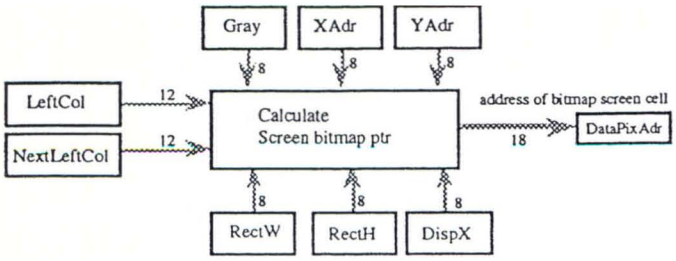

Fig. 3 - Calculating the address of the screen cell

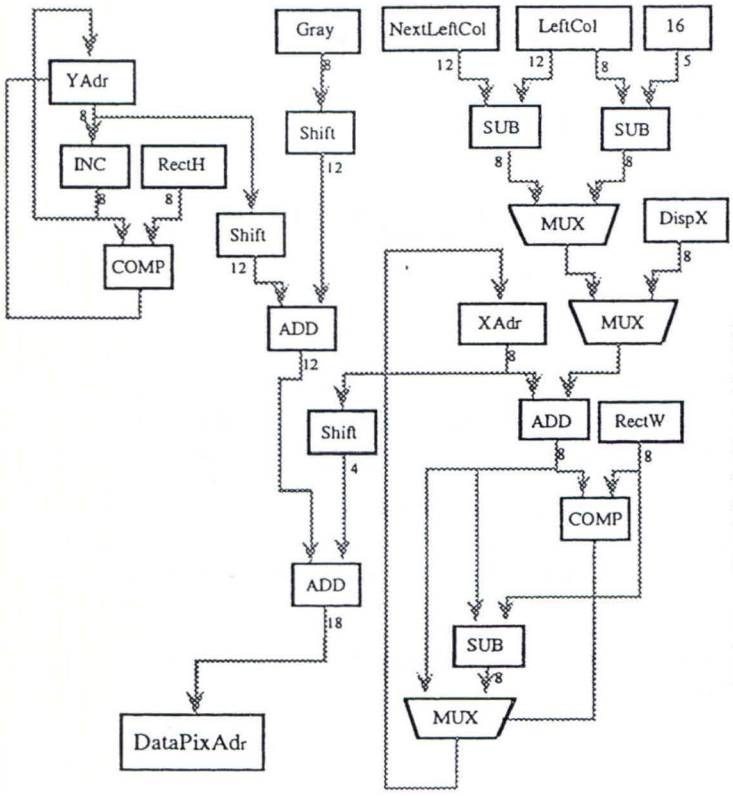

Fig. 4 - Computing the screen cell address

As the memory used for the screen cells is a fast static RAM, the operation which fetches the screen element can be achieved within a single instruction cycle. On the other hand, reading or writing to the dynamic memory requires more than one instruction cycle.

The length of the instruction cycle is function of the clock cycle time. The amount of logic through which values propagate limits the clock frequency. Typically, the stage of the algorithm during which the shifted pixels are masked needs to be long enough so that the 16-bit wide value at the entrance of the output buffer becomes valid (Figure 5).

When routing the layout of the IC, the design is mapped onto the available FPGA logic. Combinatorial logic is transformed into function generators and the registers are divided into latches, which are spread out over the whole area of the IC.

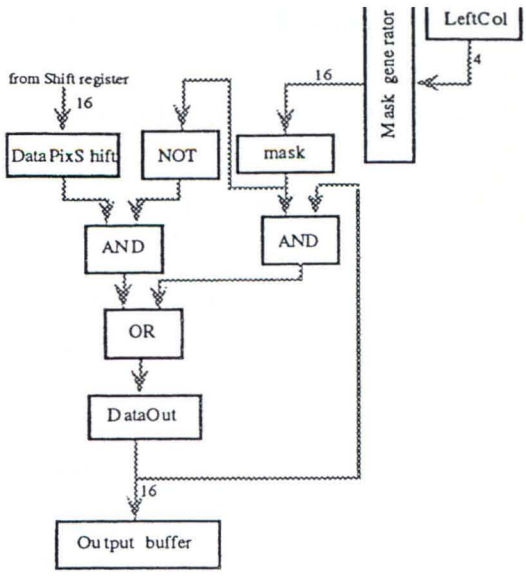

Fig. 5 - The final operation on the shifted screen cell

Shifting a 16-bit wide word to the left or to the right by a certain amount is usually implemented with a barrel shifter. This method has the advantage of being fast but requires a lot of space and is out of range of the size of the FPGA which is used here. The solution which was finally adopted consists of a register which shifts a word one bit at the time and a counter (figure 6). The counter counts how many times the screen cell ought to be shifted. Less space is needed for that solution than for a barrel shifter, but this solution is much slower.

Therefore the cycle time for the shift operation is longer than a single clock cycle and depends on the shift value. For convenience, the counter is pulsed with the same clock as the rest of the application.

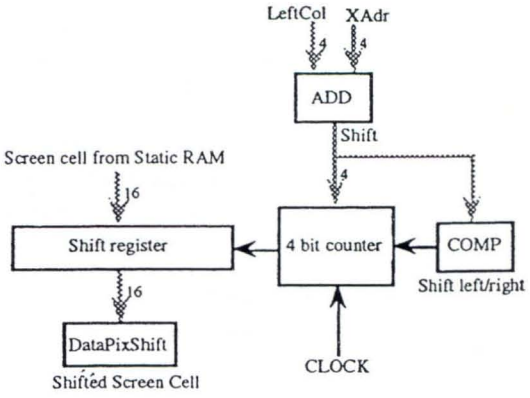

Fig. 6 - Implementing the shift operation

After having described the general architecture, we will now concentrate on the results and the performance obtained with the proposed hardware solution. Undoubtedly two operations in the algorithm are time-consuming. Accessing the memory to read or to write takes more time than one instruction cycle. Shifting too, is not fast and can even be slower than a read or write operation.

Only a larger FPGA than the XILINX 4010 can speed up the application. The shift operation should be realized with a barrel shifter. Since a barrel shifter is a combinatorial logic part, the instruction cycle necessary for shifting the screen cell would be eliminated.

Performance and results with the XILINX FPGA technology available in 1992

The application which was implemented fills 80% of the FPGA and is equivalent in size to 9492 Gate Array gates. The determined upper speed limit of the application is 4 MHz and the resulting instruction cycle time 250 ns. Speed can be increased with a larger circuit, in which the logic would be less packed. People used to work with XILINX FPGAs report that the limit below which they can be used efficiently is around 50 % CLB utilization.

The size of the registers is limiting the maximum width of the resulting destination bitmapped image to 4096 pixels. Wider output images can be generated by cutting the input and destination images into smaller parts.

Since the shifting operation is slow and due to the lack of pipeline, the speed of the circuit is twice slower than its software equivalent, running on a Sparc-2 workstation. The size and complexity of the design combined with the current FPGA technology make it impossible to go faster.

The scale factors were chosen to generate images covering a whole A4 page at output resolutions of 2540 and 5080 dpi. These are typical resolutions used for high quality printing. The tables below show the processing times (in minutes and seconds) to halftone an image scanned at 300, 600 and 800 dpi.

Backward mapping running on a Sparc-2 workstation:

Source image resolution	300 dpi	600 dpi	800 dpi
Output resolution: 2540 dpi	2:42	4:49	6:06
Output resolution: 5080 dpi	7:03	11:14	13:32

Backward mapping on the proposed hardware architecture:

Source image resolution	300 dpi	600 dpi	800 dpi
Output resolution: 2540 dpi	5:34	9:50	11:27
Output resolution: 5080 dpi	15:13	22:15	26:58

Model-based performance evaluation

It is possible to estimate the time taken by the ASIC to halftone a given image. To calculate the expected time, let us consider the following variables:

T_i : instruction cycle time

$T_{\mu P}$: microprocessor clock period

T_S : average cycle time for the shift operation

T_{rw} : average cycle time for a read or write operation

In our version, $T_i = 250$ ns, $T_{\mu P} = 50$ ns, $T_S = \underline{sh} * T_i$, $T_{rw} = 2 * T_i + 3 * T_{\mu P}$. $\underline{sh}$ stands for the average shift value of which is 8.

The time is calculated by counting the number of different instruction cycles in each loop and summing the values. Be aware that the last step, which terminates the algorithm and sends an interruption to the microprocessor, is not counted because that step tells the processor to continue with the next line of the source image. The transition time between the interruption and the start of the next step in the ASIC is small and can be omitted.

Since the memory which stores the images is shared between the microprocessor and the IC, the memory cycles are based on the microprocessor clock period.

The time (in nanoseconds) taken to halftone a source image with a size of ImSrcW * ImSrcH, given a size of ImDstW * ImDstH for the resulting destination bitmap image, can be calculated with the following formula

$$t(\mathrm{ImSrcW}, \mathrm{ImDstW}, \mathrm{ImDstH}) =$$

$$\mathrm{ImSrcH} \cdot \frac{\mathrm{ImDstH}}{\mathrm{ImSrcH}} \cdot (T_i + T_{rw} + \mathrm{ImSrcW} \cdot (\frac{1}{4} T_{rw} + T_s + 4T_i + \frac{\mathrm{ImDstW}}{16 \cdot \mathrm{ImSrcW}} \cdot (T_i + T_{rw} + T_s))) =$$

$$\mathrm{ImDstH} \cdot (T_i + T_{rw} + \mathrm{ImSrcW} \cdot (\frac{1}{4} T_{rw} + T_s + 4T_i + \frac{\mathrm{ImDstW}}{16 \cdot \mathrm{ImSrcW}} \cdot (T_i + T_{rw} + T_s)))$$

Backward mapping on the proposed architecture according to the estimation model, using $T_i = 250$ ns:

Source image resolution	300 dpi	600 dpi	800 dpi
Output resolution: 2540 dpi	5:46	9:39	12:14
Output resolution: 5080 dpi	15:18	23:04	28:15

We observe that the results approximated by the model closely match real execution times.

Possible improvements with future FPGA technologies

Using a larger FPGA would enable us to go at least 3 times faster. A new version of the IC could be driven with a higher frequency and should integrate a barrel shifter ($T_S = 0$) instead of the "step by step" shifter used here. The tables below indicate the performance which could be reached with a larger and faster circuit.

Backward mapping on a larger and faster circuit:

Here, we assume driving the ASIC with a shorter clock cycle time ($T_i = 100$ ns). Using a barrel shifter to shift the bits enables us to go faster ($T_S = 0$). The formula for the halftoning time therefore becomes:

$$t(ImSrcW, ImDstW, ImDstH) =$$

$$ImDstH \cdot (T_i + T_{rw} + ImSrcW \cdot (\frac{1}{4} T_{rw} + 4T_i + \frac{ImDstW}{16 \cdot ImSrcW} \cdot (T_i + T_{rw})))$$

The speed up reaches a factor of 3 in speed, compared to the software implementation running on a Sparc-2 workstation.

Source image resolution	300 dpi	600 dpi	800 dpi
Output resolution: 2540 dpi	0:53	1:29	1:53
Output resolution: 5080 dpi	2:22	3:33	4:21

When replacing the slow dynamic memory with a very fast source scanline buffer (FIFO access time below 50 ns), it would be possible to gain a speedup factor of 4 with the proposed hardware solution, compared with our software implementation.

Backward mapping on a larger and faster architecture with a fast FIFO:

Clock cycle time: $T_i = T_{rw} = 100$ ns.

Source image resolution	300 dpi	600 dpi	800 dpi
Output resolution: 2540 dpi	0:39	1:10	1:31
Output resolution: 5080 dpi	1:33	2:36	3:18

Possible improvement with a faster ASIC technology

Experience shows that under normal circumstances it is difficult to drive a XILINX FPGA with frequencies above 10 MHz. This disadvantage compromises with the convenience of having a multi-purpose programmable IC.

However, it is interesting to estimate the performance of the proposed architecture using an ASIC technology which is faster than XILINX, such as ACTEL or even an application specific integrated circuit, using standard cells (VLSI). In that situation, the speed of the application would only be limited by the source scanline FIFO access time. The FIFO can be chosen to be as fast as 25 ns, which would limit the instruction cycle time to 60 ns approximately.

Backward mapping with a faster ASIC:

Clock cycle time: $T_i = 60$ ns, $T_{rw} = 50$ ns.

Source image resolution	300 dpi	600 dpi	800 dpi
Output resolution: 2540 dpi	0:23	0:41	0:54
Output resolution: 5080 dpi	0:54	1:31	1:56

This solution achieves an improvement in speed by a factor of 7, compared with the software implementation. The resulting hardware solution would only be limited by the memory access time. Since we try to make a complete memory access within a single cycle, the instruction cycle time T_i cannot be smaller than the memory access time.

Conclusion

The process of halftoning is time-consuming. Even on a powerful workstation, it takes between 2 and 5 minutes to halftone a complete A4 page at 2540 dpi. The hardware solution, based on the 1992 XILINX FPGA technology is two times slower than its software equivalent, running on a Sparc-2 workstation. Since the FPGA chip used for the implementation was too small to hold the fairly large amount of necessary logic, the design was adapted to fit the limited amount of available logic, inducing severe performance degradation.

An estimation model, based on the existing design, reveals that the use of a larger FPGA combined with a fast re-readable FIFO memory could highly improve halftoning performance. Future FPGA technologies would enable us to gain a factor of 4 and a standard cell based VLSI design a factor of 7 in speed over the standard Sparc-2 software implementation.

An ideal fully parallel and fully pipelined architecture [Morgan93] would bring a gain in speed by a factor of 20, but this could only be realized by a highly complex application VLSI design.

This paper shows that only high-performance VLSI solutions are able to compete effectively with modern and high-performance processor architectures.

References

[Foley90] James Foley, Andries van Dam, Steven Feiner, John Hughes, *Computer Graphics: Principles and Practice*, Addison-Wesley, 1990.

[Hou83] Hsieh S. Hou, *Digital Document Processing, Chapter 4: Digital Halftoning and Shading*, John Wiley & Sohns, 1983.

[Morgan93] Marc Morgan et al., *Acceleration of Halftoning*, SID Digest of Technical papers, Vol. XXIV, 151-154, 1993.