Hardware Implementation of Floyd-Steinberg Halftoning Algorithm

Michael Dushkoff
Department of Computer Engineering
Rochester Institute of Technology
Email: mad1841@rit.edu

Vineeta Singh
Department of Electrical and Microelectric Engineering
Rochester Institute of Technology
Email: vxs9946@rit.edu

***Abstract*—Halftoning or dithering is the technique wherein images are converted to single on/off values for each color present using hardware or software in order to quantize a large amount of information into the same number of bits as colors in an image. Several algorithms such as dot diffusion, blue noise masks, green noise halftoning, direct binary search (DBS) etc have evolved over the years for halftoning. One popular method for halftoning is referred to as error diffusion which has been used extensively in various applications in order to correct the errors representing each pixel value using a single bit.**

The quality of image offered by error diffusion methods is regarded superior to majority of other known methods. Traditional error diffusion methods compares the signal at the current pixel value with a certain threshold, and depending on whether it is less or more than that threshold value, it concludes whether it is a bright or dark dot. The difference between the actual signal and the decision is termed as quantization error which is diffused to the neighboring pixels.

The error diffusion algorithm developed by Robert W. Floyd and Louis Steinberg is currently one of the most popular methods used for halftoning.

I. Introduction

Halftoning helps in rendering a continuous-tone colored or grayscale images on a computer, printer or other media on which two levels can be displayed. Earlier in late nineteenth century, it was difficult to print images on paper using the printing machines. In order to make that possible, the sizes of the dots were varied according to the intensity required by the local image. If the image was darker, then the halftone dots become larger and can even overlap at times. This process can be defined as analog halftoning [1]. Some other earlier methods involved using woodblock printing where different blocks were used for producing patterns, or using hand-coloring to add colors to printed items on paper. Halftoning is used in a lot of devices such as early video adapters, modern LCDs, mobile phones, inkjet printers. It is also widely used for compressing images and video clips. Some other applications for halftoning can be found in digital audio, digital video, digital photography, seismology, weather forecasting system, in RADAR, and different fields where digital processing and analysis can be done.

Digital halftoning is a method designed which uses black and white pixels to produce a gray-scale image [2]. It creates an illusion of continuous-tone pictures on bi-level displays taking advantage of the fact that human eye blends the halftone image into a smooth one thus preserving a significant portion of the original content. Digital halftoning can also be done for colored images using a limited color palette. Different methods which are quite popularly used for halftoning are ordered dithering, iteration-based dithering, and error diffusion [3]. The image obtained after performing error diffusion is of good visual quality and is one of the most preferred methods as it has low computational complexity. For color halftoning, the color is handled by using four superimposed halftones, like cyan, yellow, magenta and black, however it is one per color.

In order to demonstrate that this halftoning algorithm can be efficiently implemented in hardware, an FPGA implementation was created in VHDL. This was then uploaded to a Xilinx Nexys 3 board which utilizes a Spartan-6 FPGA. Images were sent to the device through its built-in UART transmit and receive port, processed on-chip, and sent back to a computer where the images were displayed in MATLAB. In order to verify this process, we show the MATLAB implemented algorithm directly compared to the hardware algorithm and the times are compared in order to show the advantage that a hardware implementation can have over a software solution.

II. Background

The error diffusion algorithm was first proposed by Robert W. Floyd and Louis Steinberg in 1976 [4]. It helps in combining few operations and gives a decent visual performance. The pixels are processed in a linear top-to-bottom, left-to-right order and the grayscale value is compared with the threshold value (127) at every step. If the value is greater than the threshold value, the pixel is considered as white and the output value is set to 1, otherwise the pixel is considered to be black, and the output value is set to 0. The quantization error is calculated as the difference between the pixel value and the threshold value. For achieving the continuous-tone effect, the quantization error is distributed to the four neighboring pixels at a time which are not processed yet [5]. The coefficients for distribution are 7/16, 3/16, 5/16, and 1/16. Fig. 1 shows the high-level implementation of the error-diffusion kernel for this process.

The pseudocode for implementing error-diffusing using the Floyd-Steinberg algorithm is as follows:

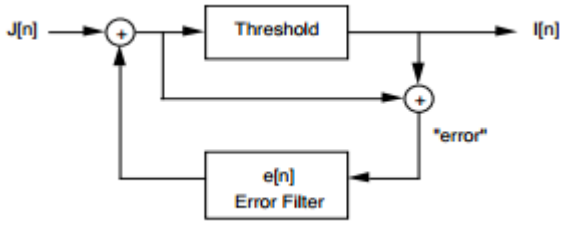

Fig. 1: Floyd-Steinberg Dithering [1]

```
for row in 1 to max_row loop
    for col in 1 to max_col loop
        old_pixel:=pixel [col][row];
        new_pixel:=old_pixel<127 ?0:255;
        quant_error:=old_pixel-new_pixel;

        pixel[col+1][row]:=
            pixel[col+1][row]
            +quant_error*7/16;
        pixel[col-1][row+1]:=
            pixel[col-1][row+1]
            +quant_error*3/16;
        pixel[col][row+1]:=
            pixel[col][row+1]
            +quant_error*5/16;
        pixel[col+1][row+1]:=
            pixel[col+1][row+1]
            +quant_error*1/16;
    end loop;
end loop;
```

Fig. 2: Floyd-Steinberg Algorithm [4]

III. Design Methodology

From the algorithm described, a hardware architecture was developed in order to implement the Floyd-Steinberg algorithm in hardware using a Xilinx Nexys 3 development board. The architecture was based off of a previously developed coprocessor design which could already send and receive images through the UART interface of the board which made it simpler to implement the image processing algorithm itself integrated into it. The architecture was made up of a large number of buffers that store intermediate calculation results before storing them back into memory. This avoids memory access and write latencies due to the fact that the buffers are implemented such that there is only a single clock cycle between reads and writes of the buffer values. These buffers were broken up into storing the quantization error, storing the multiplication of this quantization error, storing the addition and shift of the multiplication values, and finally a buffer to store the calculated final pixel value. This architecture can be seen in Fig. 3. The row buffer initially loads an entire two rows of the image from memory and then proceeds to calculate the diffusion error from left to right. This calculation occurs in a

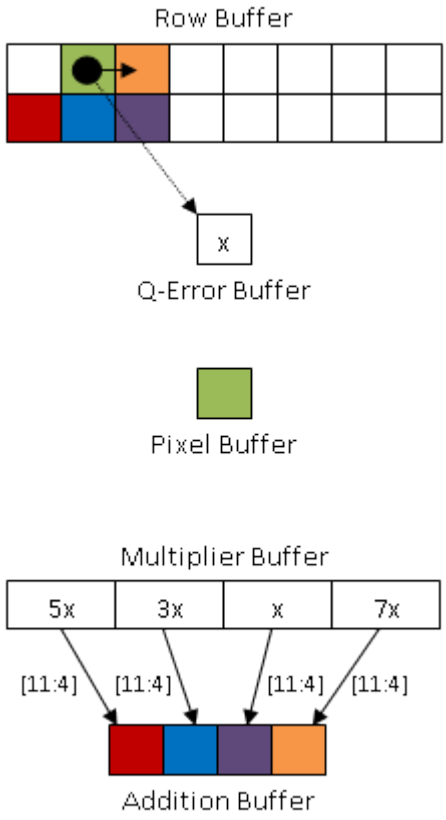

Fig. 3: Intermediate Calculation Buffers

pipelined fashion until it reaches the end of the row wherein the previous row of values are pushed upward and the next row from memory is loaded into the bottom of the row buffer.

In order to use less hardware, instead of implementing division by 16, a shifting operation was used which takes the multiplied quantization error values from their fourth bit up to their eleventh bit. This is faster than implementing division and yields the same result. After the addition operation is performed the values are then stored back into their corresponding positions in the row buffer as shown in Fig. 3 (Colors represent where calculations will be stored when complete).

The finite state machine shown in Fig. 4 was developed based on the previous UART coprocessor architecture in order to allow halftoning to process throughout thirteen state transitions excluding the RAM writes and reads. The addition and multiplication were split up into four steps each in order to check the condition wherein the edges of an image were being processed since the kernel can not be applied to the leftmost part or rightmost part of any image. Once these steps are completed, the state machine loops around until the number of pixels in a single row are completely finalized. This then initiates a shift and read from RAM once again and that operation continues until the entire image is completely processed.

Three separate commands exist in order to accomplish this. The first command is initiated by MATLAB and immediately

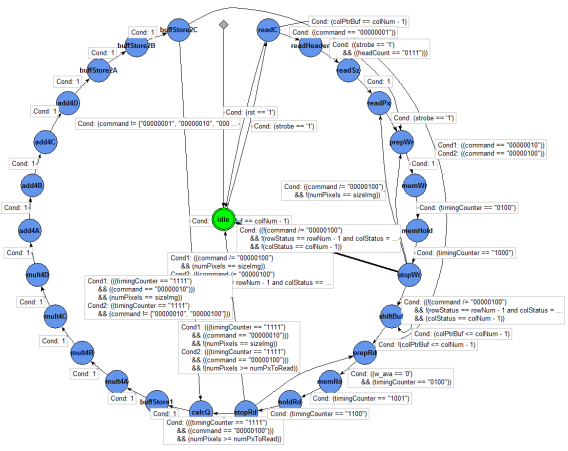

Fig. 4: State Machine Diagram of Hardware Architecture

expects an image to be sent through UART along with its size information. This image is read in pixel by pixel until the total size of the image has been completed. The second command is to send a processed picture back through UART to MATLAB which is also requested through software. This takes the memory address where the processed image is stored and sends it back to MATLAB one pixel at a time until the entire image has been sent. The final command is the process command which performs the dithering algorithm on the image that was sent through MATLAB to the device. By separating these functionalities into separate commands, the device can easily take input from another device and process data based on those commands.

IV. Experimental Results

First the algorithm was implemented in MATLAB in order to show the baseline results of applying the algorithm to grayscale images. These results can be seen in Fig. 5. This shows image before (image on the left) and after (image on the right) grayscale halftoning was applied. The MATLAB implementation can also be used on colored images by extracting the color as a separate picture and applying the algorithm to one color at a time. The image is then recombined with the final pixel values which will be either fully saturated, or completely off giving effectively N bits of precision where N is the number of colors in the color profile of the image. Popular color profiles include RGB, CMYK, and YUV, the first two of which are commonly found in printer hardware.

This software implementation was run and timed to take 0.111991 seconds in order to complete. This, however, is unoptimized code and could potentially run faster if the Floyd-Steinberg algorithm was vectorized in MATLAB in order to take advantage of the fast operations that it can do. This verison of the algorithm would be noticably slower on larger images which could be detrimental if the algorithm is to run on hardware that requires next to realtime processing (such as in printing).

Next the hardware implementation was tested through the use of the Nexys 3 board and MATLAB. The MATLAB

Fig. 5: MATLAB Implementation of Grayscale Floyd-Steinberg

Fig. 6: MATLAB Implementation of RGB Floyd-Steinberg

environment was only utilized in order to send the picture to the device, while all of the processing occurred on-chip. This processing time was very much improved from the MATLAB timing at 0.0033 seconds which was calculated by storing the number of clock cycles that the computation took into a buffer that could be read out on the LEDs of the device. This buffer had registered 329784 cycles of a 100MHz clock after the image was processed which explains the calculated time. From this it can be concluded that a hardware implementation of the Floyd-Steinberg algorithm is worth using due to the fact that the processing time is much faster and could possibly be utilized in an embedded environment.

The individual resources that were used in the device are shown in Fig. 7.

Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	2420	18224	13%
Number of Slice LUTs	6576	9112	72%
Number of fully used LUT-FF pairs	2398	6598	36%
Number of bonded IOBs	67	232	28%
Number of Block RAM/FIFO	1	32	3%
Number of BUFG/BUFGCTRLs	2	16	12%

Fig. 7: Synthesis Report

This shows that a majority of the Spartan 6's lookup tables were utilized (72%) which corresponds to the amount of overhead that goes into creating separate buffers for storing intermediate results of computation. The design originally planned to utilize block RAM in order to quickly read and store values, however it was difficult to force the Xilinx design

tools to utilize it since the row buffer itself was controlled by multiple different signals in order to address it and it reads and writes to and from more than two locations. If it only wrote and read from two or less locations, the dual-port block RAM would have been synthesized in hardware. Furthermore, the maximum frequency of the clock in the design was found in the post-synthesis report to be 100.452MHz which is slightly faster than the board clock, which means that there is little overhead for the data to process. This is encouraging to see, however the sheer amount of hardware resources that are already taken up show that there is not much more space to implement much more to this coprocessor architecture. This leads to a limitation in the current architecture design in that the row buffer's column size limits the column size of any image that is passed in. This is due to the fact that entire columns from memory are passed directly into the row buffer and processed row-wise. This limitation can be overcome in future designs by designating a specific chunk of memory to the row buffer that can process chunks of the image at a time, while the quantization errors are stored back to RAM after the pixels in the row buffer are fully computed. This would ensure that each row would calculate error diffusion correctly and that the finalized image could be larger than the size of the row buffer.

V. Conclusion

Dithering makes up an important part of digital information processing in the world today from applications such as printing and even digital displays. Such a concept leverages the human eye's inability to distinguish individual colors from a group of closely knit colors if the resolution is small enough. This allows displays to be developed with less hardware in groups of three bits for red, green, and blue pixel values. These three colors can only be on or off in pixels on a display which necessitates the use of dithering in order to trick the human eye into perceiving the correct color. To do this, the Floyd-Steinberg error diffusion algorithm was implemented in hardware in order to show the speed advantage that a hardware implementation would have over a software version. The software version in MATLAB completed processing a small 98x81 pixel grayscale image in 0.1120 seconds while the same image was processed on a Xilinx Spartan 6 FPGA in under 0.0033 seconds. This speedup is however, not with a cost sicne the time that it takes to transfer the image to and from the FPGA is much longer than the processing. For embedded applications that need to utilize additional functions as well as this filter, the current architecture would not quite be the best choice due to the fact that much of the FPGA's resources are taken up from the error diffusion algorithm, however this can be reduced by creating a smaller fixed-size block calculation of the design which would be slower, but take up far less resources. Speed improvements could also be done to the design if the design were to properly utilize block RAM instead of registers and lookup tables, however this design oversight did not affect the overall operation of the device. In conclusion, the algorithm operated as expected and took up a good deal of FPGA resources which could have been implemented with less if a system where it was implemented in demanded more resources and could withstand a slight performance hit.

VI. Acknowledgements

The authors would like to thank Dr. Marcin Lukowiak for suggesting this as one of the topics for authoring the research paper and encouraging us with the work as well as providing an excellent picture for us to use for the final implementation.

References

[1] M. Mese and P. Vaidyanathan, "Recent advances in digital halftoning and inverse halftoning methods," *Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on*, vol. 49, no. 6, pp. 790–805, 2002.

[2] M. Erkoç and A. Yurdakul, "Halftoning soft cores for low-cost digital displays," in *Computer and Information Sciences, 2009. ISCIS 2009. 24th International Symposium on*. IEEE, 2009, pp. 76–81.

[3] J.-M. Guo and J.-H. Chen, "High efficiency digital halftoning with two-element error kernel," in *Image Processing, 2006 IEEE International Conference on*. IEEE, 2006, pp. 1501–1504.

[4] R. W. Floyd, "An adaptive algorithm for spatial gray-scale," in *Proc. Soc. Inf. Disp.*, vol. 17, 1976, pp. 75–77.

[5] P. T. Metaxas, "Parallel digital halftoning by error-diffusion," in *Proceedings of the Paris C. Kanellakis memorial workshop on Principles of computing & knowledge: Paris C. Kanellakis memorial workshop on the occasion of his 50th birthday*. ACM, 2003, pp. 35–41.