This paper presents an in-place computing design for the deblocking filter used in H.264/AVC video coding standard. The proposed in-place computing flow reuses intermediate data as soon as data is available. Thus, the intermediate data storage is reduced to only the four 4x4 blocks instead of whole 16x16 macroblock. The resulting design can achieve 100MHz with only 13.41K gate count and support real-time deblocking operation of 2Kx1K@30Hz video application when clocked at 73.73MHz by using 0.25µm CMOS technology.

Index Terms—VLSI architecture design, deblocking filter, H.264/AVC

1. INTRODUCTION

The block-based video coding, due to its simple and regular block structure, has been widely used in various video coding standards, such as MPEG-1, MPEG-2, MPEG-4 and H.26x. However, the block structure also introduces the blocking effect [1][2], which becomes worse especially in the low bit rate or highly compressed video environment. To reduce the artifact, the deblocking filter is adopted to improve both objective and subjective video quality, either inside or outside the coding loop. The in-loop approach adopted by the H.264/AVC standard [3], as shown in Fig. 1, improves the quality of reference frame, thereby improving the perceived quality of the decoded video.

However, the deblocking filter in H.264/AVC is both computational and memory intensive due to its highly adaptive mode decision and small 4x4 block size [4]. The adaptive mode decision is required for each edge to distinguish real edges from block artifacts. The small 4x4 block size used in H.264/AVC requires almost every pixel in a frame loaded from and written to frame memory for deblocking operations. It is reported that even with highly optimized filtering algorithm, the deblocking operation still occupies one third of the computational complexity of a decoder [4]. Thus, VLSI implementation is necessary for high performance or low power applications such as HDTV or mobile phone.

For an efficient VLSI design of the deblocking filter, memory bandwidth and intermediate buffer are the major design issues. In the H.264/AVC standard, the adaptive deblocking filter is applied on edges of each 4x4 block in a macroblock (MB). For details, readers may refer to [3]. The processing order of the deblocking filter in a macroblock, shown at Fig. 2, first processes on the four vertical edges and transposes the intermediate data, and then processes on the four horizontal edges. The major drawback of this direct approach is that intermediate data storage is as large as a whole 16x16 macroblock size. Besides, this data flow accesses the 4x4 block four times for original and intermediate data, which results in high bandwidth. Previous approaches directly adopted this coding flow as in the reference software [5], thus required a large buffer size and high bandwidth [6].
Instead of the direct approach, this paper proposes an in-place computing architecture that reuses intermediate data as soon as data is available. With such full data reuse, both memory bandwidth and intermediate storage can be reduced. The resulting design can easily meet the real-time high resolution requirement (2048x1024@30Hz) with low cost hardware, such as live broadcasting or HDTV applications.

The organization of this paper is as follows. In Section II, we describe the full data reuse computational flow of deblocking filter in H.264/AVC. In Section III, we present the hardware architecture. Then, the architecture performance will be compared with other approaches in Section IV. Finally, concluding remarks will be made in Section V.

II. THE FULL DATA REUSE COMPUTATIONAL FLOW

Fig. 3 shows the 4x4 block index for the full data reuse computational flow. Before deblocking processing, the blocks above (blocks 1-4) and left (blocks 5, 10, 15, and 20) from the current macroblock are first retrieved from external memory for current macroblock deblocking. In Fig. 3, data from blocks 1-4, 5, 10, 15, and 20 have been vertically and horizontally filtered as part of the filtering process for the above and left macroblocks respectively. For simplicity, we will denote the block number as blk\textsubscript{i}, where i is from 1-24. These block data will be combined with current macroblock data to complete the deblocking operations.

![Fig. 3. 4x4 block index of the full data reuse flow.](image)

Fig. 4 shows the full data reuse computational flow that maintains the same result as specified by the H.264/AVC standard. The processing order is as denoted by the number in Fig. 4. As shown in Fig. 4 (a), we process edges of each 4x4 block from the left-top-most block (blk6), and proceed to the right-bottom-most block (blk24). Start from the left-top-most 4x4 block (blk6). We first do the horizontal filtering over its two vertical edges, (edge 0 and edge 1). Then, since all data is available for horizontal edge 2 (intermediate data from blk1 and blk6), we can do the vertical filtering over the top horizontal edge (edge 2). This horizontal-vertical interleaved approach is repeated for each 4x4 block in a raster scan order, as the edge number shown in Fig. 4 (a) and (b).

With the interleaved approach, the intermediate data will be used immediately. Thus we can save the memory access and buffer required to process the left, top, and right edge in a 4x4 block. The only buffer and memory access remaining are the intermediate data for the bottom edge in a 4x4 block that will be used in the below MB. Therefore, we need only four 4x4 blocks (4x16x8bits) above the current filtering block row (e.g. store blk6-blk9 when processing blk11-blk14), rather than a whole macroblock (24x16x8bits) as in the conventional data flow. Because of this high data reuse, internal memory access number and size are both greatly reduced. Note that the deblocking algorithm, as described in the paper, in general, requires all macroblock data to be received prior to application of the deblocking filter process for FMO and ASO cases in H.264. The in-place algorithm is compatible to MBAFF feature of H.264/AVC since our algorithm only changes the computing order inside the macroblock and keeps the macroblock processing order unchanged.

The in-place deblocking algorithm is also applicable to software implementations due to low memory requirement and in-place computation. For example, a software implementation might, depending on the computing platform, be able to load all data necessary to filter a 4x4 block into on-chip registers, thereby providing extremely fast computation of the filter process. The described technique might be particularly powerful if the deblocking filter process is carefully pipelined (e.g., prefetch/DMA picture data from memory to L1 cache followed by a process designed to move data from L1 cache to on-chip registers).
III. IN-PLACE COMPUTING ARCHITECTURE

Overall Architecture and Computation Flow

Fig. 5 shows the proposed architecture, where the solid arrows denote 32-bit dataflow. First, we assume that all data I/O is row major order with 32-bits width (4 pixels). For simplicity, we will only explain the operation of luma block. The chroma block is processed using the same strategy.

In the first 16 cycles, we transpose data of block indexes 1 to 4 from row major order to column major order with the transpose buffer, and store them in the local SRAM buffer to wait for vertical filtering. Then we start horizontal filtering over the vertical edge 0 as shown in Fig. 4 (a) with data of blocks 5 and 6. The data of block 5 is from external memory via input port and shifted into 4x4 shift register after four cycles. After that, the data of block 6 from input port, and data of block 5 from 4x4 shift register are loaded to perform the horizontal filtering as shown in Fig. 6(a). The filtered data of block 5 is sent to output port, and the data of block 6 after first time filtering is sent to 4x4 shift register. So the data in 4x4 shift register can be used to perform the next horizontal filtering.

Next we start horizontal filtering over vertical edge 1 with data of block 6 and 7. In this phase, filter input is from 4x4 shift register (block 6), and from external memory (block 7) via input port as shown in Fig. 6 (b). The filtered data of block 6 is then transposed to column major order by transpose buffer after the data has been filtered two times, and the data of block 7 after first time filtering is sent to 4x4 shift register.

The data of block 1 and block 6 are both ready to perform vertical filtering over horizontal edge 2 in column major order now. In this phase, filter input is from local SRAM buffer (block 1), and from 4x4 transpose register (block 6) as shown in Fig. 6 (c). The data of block 6 after filtering is sent to local SRAM buffer for use when filtering horizontal edge 10 and the data of block 1 is transposed again and output in row major order in the next four cycles. The remaining edges are processed using the same data flow mentioned above.

The 16x32bits SRAM buffers four 4x4 block pixels to be processed, as described in the previous Section. The register array is for transposing operation during the 2-D deblocking filtering.

A. Filter Architecture

To speedup the processing, this design uses parallel-in parallel-out style (32-bit, processing 4-pixels concurrently). The deblocking filter part implements the required function as specified by the H.264/AVC standard.

Fig. 7 shows the 8 pixels parallel-in parallel-out filter architecture. Input pixels pass by normal filter and strong filter, and output 8 pixels are selected by Bs. Fig. 8 shows the name of a line data to be filtered by the parallel-in parallel-out filter. Strong filter and normal filter are shown in Fig. 9 and Fig. 10. Only right parts are shown in the figure, since the left parts are symmetric to right parts. In the datapath of strong filter and normal filter, multiple input adders are implemented by carry save adder to speedup the addition operation.

As shown in Fig. 7, the filtering does not take place for edges with Bs equal to zero. For edges with nonzero Bs values, two quantization-dependent parameters, referred to as $\alpha$ and $\beta$, are used to determines whether the input data is filtered. Filtering on a line of samples only takes place if the three conditions all hold true.

$$|R0 - L0| < \alpha, |R0 - R1| < \beta, \text{ and } |L0 - L1| < \beta$$

If one of the three equation not holds true, and the control logic will select the output of Bs=0.
Proposed Architecture two port two port dual port x1
Memory size (bits) 16x32 160x32 32x64x1 32x96x2 Frame size 96x32 80x32
Memory Architecture two port two port dual port x1 two port single port single port

Table 1 Comparisons with other design

<table>
<thead>
<tr>
<th></th>
<th>Proposed</th>
<th>[6]</th>
<th>[7]</th>
<th>[8]</th>
<th>[9]</th>
<th>[10]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Technology</td>
<td>0.25µm</td>
<td>0.25µm</td>
<td>0.25µm</td>
<td>n/a</td>
<td>0.18µm</td>
<td>0.18µm</td>
</tr>
<tr>
<td>Gate Count (100MHz) (without SRAM)</td>
<td>330</td>
<td>614</td>
<td>446</td>
<td>&gt;600</td>
<td>50(best)-342(worst)</td>
<td>336</td>
</tr>
<tr>
<td>Cycle/MB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory size (bits)</td>
<td>16x32</td>
<td>160x32</td>
<td>32x64x1</td>
<td>32x96x2</td>
<td>Frame size</td>
<td>96x32</td>
</tr>
<tr>
<td>Memory Architecture</td>
<td>two port</td>
<td>two port</td>
<td>dual port x1</td>
<td>two port x2</td>
<td>two port</td>
<td>single port</td>
</tr>
</tbody>
</table>

B. Memory Organization

In the proposed computation flow, the deblocking operation uses horizontal-vertical interleaved scheduling. However, to support such the interleaved operation, the corresponding architecture shall transpose the 4x4 block immediately when changing the filtering edges. The corresponding data path consists of a 4x4 shift buffer and a 4x4 transpose buffer. We assume that SRAM module is an ordinary one which has one 32bits read port and one 32bits write port. With this, we can transpose the 4x4 data blocks to support both horizontal filtering and vertical filtering on a parallel-in parallel-out deblocking filter seamlessly.

The on-chip buffer first stores blk1 to blk4. Fig. 7 shows its data organization, where m is an integer number from 0, the number in the block denotes the block index and each block stores one 4x4 block data. Then the same address location will be overwritten by new data resulting from blk6 to blk9, respectively. This will be repeated for each row of 4x4 blocks. Since data in-place is overwritten after reading out, no read-write conflict will occur.

C. Processing Schedule

Assume input data are four pixels (32bits) per clock cycle, it requires 16+(32+4+4)x4+16=192 cycles to process the luma block of a macroblock without overlapping the data flow. Among them, 16 cycles to input data of block 1 to 4 from external memory to on-chip memory before the start of data processing. 4x8=32 cycles to process edge 8m to 8m+6, 4 cycles to shift block 8m+9 from 4x4 shift buffer to 4x4 transpose buffer, 4 cycles to process edge 8m+7, and 16 cycles to output data of block 21 to 24 from on-chip memory to external memory, where m is an integer number from 0 to 3. For two chroma blocks, (8+(20+4)x2+8)x2=128 cycles are required. Thus, 320 cycles are required to process one
macroblock.

For a more efficient processing schedule, we can save four cycles by overlapping the data loading cycles of block 5\((m+1)\) from the external memory and data shifting cycles of block 5\(m+4\) from 4x4 shift buffer to 4x4 transpose buffer before processing horizontal edge 8\(m+7\). For example, we need four cycles to shift the data of block 9 to 4x4 transpose register. During the same four cycles, we can also move the data of block 10 from external memory to 4x4 shift register. With this scheduling, the total cycle count is reduced to 300 cycles for one macroblock.

IV. IMPLEMENTATION AND COMPARISON

To evaluate the accuracy and the efficiency of the proposed architecture, the architecture is designed in Verilog and implemented by TSMC 0.25\(\mu\)m CMOS technology. The result has been verified with that from the reference software and all data are matched. The resulting hardware can achieve real-time 2Kx1K (2048x1024) 30Hz video at 73.73 MHz. The gate count is only 13.41K when synthesized at 100MHz, excluding the memory cost. This design is for 8-bit data. However, it can be easily extended into wider word length to support other profiles.

Table 1 compares our design with the state-of-art approaches in [6]-[10]. For filtering a MB, our design requires least memory size because of using the in-place architecture. Specifically, we have the smallest cycle count at worst case when processing a MB. The processing order of [7], though similar to the proposed one, has quite different architectures and thus higher gate count and cycle numbers. In [9], they use adaptive MB transmission scheme to early skip the edge data which is unnecessary to be filtered. However, its result is dependent on Qp, video content, and frame types.

V. CONCLUSION

In this paper, we contribute a full data reuse deblocking processing flow and its corresponding VLSI architecture for deblocking filter in H.264/AVC. By rearranging the data flow we can achieve high data reusability, and therefore reduce the required memory size. The major idea is to filter a vertical edge immediately followed by the filtering of a horizontal edge for a 4x4 block instead of whole macroblock. With a 4x4 transpose buffer, the aforementioned interleaved vertical and horizontal deblocking filtering can be easily realized. Thus, the processing capability of the proposed architecture can operate very efficiently. The performance can meet high resolution, real-time processing applications and suitable to be integrated into H.264/AVC codec.

REFERENCES


