Circuit Implementations for High-Efficiency Video Coding Tools

by

Mehul Tikekar

B.Tech., Indian Institute of Technology Bombay (2010)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2012

Author . . . .
Department of Electrical Engineering and Computer Science
May 22, 2012

Certified by . . . .
Anantha P. Chandrakasan
Joseph F. and Nancy P. Keithley Professor of Electrical Engineering
Thesis Supervisor

Accepted by . . . .
Leslie A. Kolodziejski
Chairman, Department Committee on Graduate Theses

Circuit Implementations for High-Efficiency Video Coding Tools

by

Mehul Tikekar

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 2012, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

High-Efficiency Video Coding (HEVC) is planned to be the successor video standard to the popular Advanced Video Coding (H.264/AVC) with a targeted 2x improvement in compression at the same quality. This improvement comes at the cost of increased complexity through the addition of new coding tools and increased computation in existing tools. The ever-increasing demand for higher resolution video further adds to the computation cost. In this work, digital circuits for two HEVC tools - inverse transform and deblocking filter are implemented to support Quad-Full HD (4K x 2K) video decoding at 30fps. Techniques to reduce power and area cost are investigated and synthesis results in 40nm CMOS technology and Virtex-6 FPGA platform are presented.

Thesis Supervisor: Anantha P. Chandrakasan
Title: Joseph F. and Nancy P. Keithley Professor of Electrical Engineering

Acknowledgments

I would like to thank

- Prof. Anantha Chandrakasan, my research advisor
- Vivienne Sze, a great project mentor
- Chao-Tsung Huang and Chiraag Juvekar, my colleagues on the project
- Masood Qazi, who “taught me everything I know”
- Sai Kulkarni, for her advice when everything seemed just another brick in the wall
- The authors of LaTeX, GNU/Linux, Eclipse, Python and countless other programs I have freely used and taken for granted

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The ever-increasing demand for richer video content and explosion in use of internet video have motivated work on algorithms that achieve higher video compression without sacrificing visual quality. Experts from the Moving Picture Experts Group (MPEG) and Video Coding Experts Group (VCEG) formed the Joint Collaborative Team on Video Coding (JCT-VC) that first met in April 2010 to evaluate proposals that achieve twice the video compression of the popular H.264/AVC standard at the same quality. Following two years of extensive research involving 9 meetings and over 4000 contributions, High-Efficiency Video Coding (HEVC) has developed as a successor video compression standard to H.264/AVC [1]. HEVC Test Model HM-4.0 [2], the reference software version used in this work achieves 1.5× to 2× bitrate improvement over AVC High Profile [3].

In video conferencing and video streaming over ethernet, this would result in lower latencies and shorter buffering times. However, this coding efficiency is achieved at the cost of increased computational complexity. Even a decoder targeting the Low Complexity mode is 61% more complexity compared to a baseline H.264 decoder. [4]. In mobile applications, the energy cost for downlink over wireless networks like 3G, WiFi, LTE is $\approx$ 100nJ/bit [5]. Compared to that, an H.264 hardware decoder with external DDR2 memory requires $\approx$ 1.5 nJ/decoded pixel[6] or 10 nJ/compressed bit for a typical compression ratio of 50. In these applications, reducing power consumption clearly requires higher compression even at the cost of increased decoder complexity. These factors motivate more complicated algorithms to achieve lower bitrates and work on hardware implementations to mitigate the processing cost and enable real-time decoding.

This work focuses on the hardware design of two coding tools in HEVC - inverse transform and deblocking filter. A short explanation of HEVC and its improvements over H.264 are presented in this chapter followed by a description of the HEVC hardware decoder and the key contributions of this work. The next two chapters focus on the design of the two coding tools and synthesis results. The results are summarized in the concluding chapter and future research directions are proposed.

1.1 High-Efficiency Video Coding

HEVC targets a wide range of picture resolutions from 320×480 up to 7680×4320. A decoder may choose to support a subset of these resolutions specified as a ”level” in the standard. Similarly, the decoder may support a subset of all the coding tools defined as a profile. The supported pixel format is 8-bit YUV420. HEVC works by breaking the picture into blocks called Largest Coding Units (LCU). These LCU's are processed in a horizontal raster scan order. The allowed LCU sizes are 64×64 (LCU_{64}), 32×32 (LCU_{32}) and 16×16 (LCU_{16}) though the LCU size is fixed for a video sequence and manually specified as an encoding parameter.

Each LCU can be recursively split into four equal squares as shown in Figure 1-1. The leaf nodes in this quad-tree are called Coding Units (CU). HEVC supports CU_{64}, CU_{32}, CU_{16} and CU_{8}. By allowing a wide range of Coding Unit sizes, the standard enables an encoder to effectively adapt to video content. Uniform regions of the picture, typically in the background, can be encoded as larger CU's while regions with more detail and more motion can use smaller CU's. It should be noted that the standard dictates only the format of the compressed video and the decoding process for it. The encoder is free to make optimizations and trade-offs as long as it generates a compliant bitstream. The HM-4.0 reference encoder is computationally very complex as it uses all LCU and CU configurations. Practical encoders can be

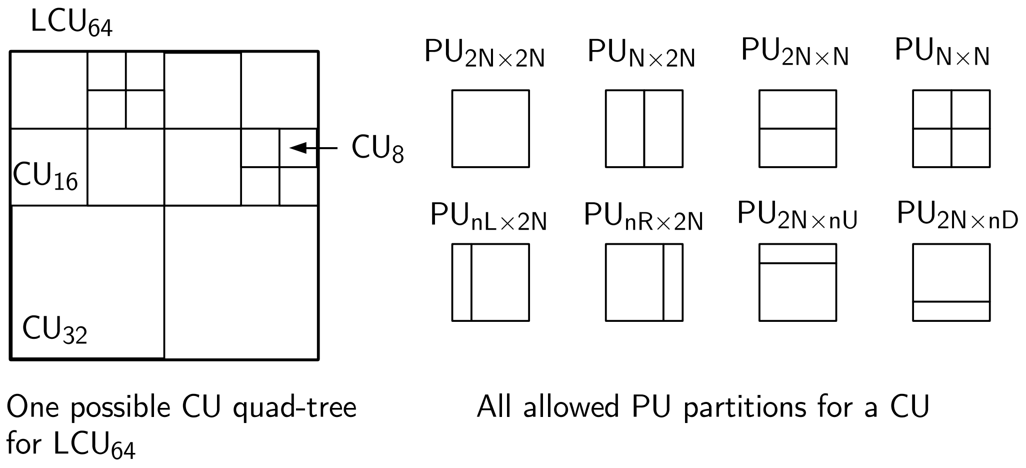


Figure 1-1: CU and PU partitions in HEVC

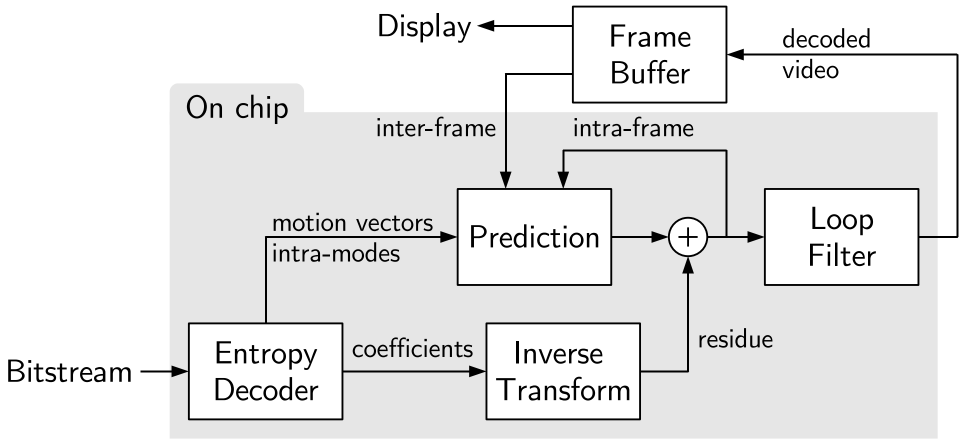


Figure 1-2: Typical video decoding scheme

expected to use only a subset of configurations.

Each CU is partitioned into 1, 2, or 4 prediction units (PU) shown in Figure 1-1. The CU can use inter-frame or intra-frame prediction. Inter-frame prediction can use uni-prediction or bi-prediction while 36 intra-prediction modes are available. For the purpose of residue coding, each CU is further divided into a residue quad-tree. The leaf-nodes of the residue quad-tree are called Transform Units (TU). Unlike the CU quad-tree, the TU quad-tree can use non-square partitions. HEVC uses 4 square and 4 non-square TU sizes. TU size depends on the PU partition and CU size as listed in Table 2.1.

Figure 1-2 shows the block diagram of a typical HEVC decoder. The entropy

Feature	HEVC	H.264/AVC
Processing block	$LCU_{64}, LCU_{32}, LCU_{16}$	MB 16×16
Entropy decoder	CABAC, CAVLC*	CABAC, CAVLC
Prediction unit	8 types for each CU	8 types
Transform unit	8 sizes: 4×4 to 32×32	2 sizes: 4×4, 8×8
Intra-prediction	36 modes	10 modes
Loop filter	Deblocking, ALF, SAO	Deblocking

Table 1.1: Comparison of HEVC HM-4.0 and H.264 features. *CAVLC removed from later version of HEVC.

decoder parses the compressed bitstream for information such as CU, PU, and TU sizes, motion vectors, intra-modes and residue coefficients. The prediction block generates a prediction for each PU based on its motion vector (inter-prediction) or intra-mode (intra-prediction). The residue coefficients for each TU are scaled by a quantization parameter and transformed, typically using a 2-dimensional inverse DCT, to generate the residue. Residue is added to prediction to reconstruct the video. The reconstructed pixels are filtered before writing back to the decoded frame buffer. Typical video sequences have large spatial correlation within the frame and temporal correlation across frames. So, the decoded video can be used to generate predictions for the new pixels.

The loop filter acts on edges of the CU. As a result, it requires pixels from CU's after the current CU. To avoid a chicken-and-egg problem, intra-prediction uses reference pixels prior to the loop filtering. HEVC uses three concatenated loop filters - deblocking filter, adaptive loop filter (ALF) and sample adaptive offset (SAO). Deblocking is used to smoothen blocking artifacts added to the video by the lossy decoding process. The others are adaptive filters which allow the encoder to compare the deblocked pixels with the raw input pixels and adapt the filter parameters to achieve lowest error.

The differences between HEVC and H.264 are summarized in Table 1.1.

LCU	LPU size	No. of LCU's	No. of pixels
LCU_{64}	64×64	1	6144
LCU_{32}	64×32	2	3072
LCU_{16}	64×16	4	1536

Table 1.2: Pipeline buffer size for all LCU's

1.2 HEVC Hardware Decoder

The inverse transform and deblocking filter designed in this work are used in an decoder capable of real-time Quad-Full HD (3840×2160) decoding at 30 fps. The HEVC standard was in flux throughout this project and decoder profiles have not been fixed by the JCT-VC. So the choice of coding features supported by this decoder was based on what may be most interesting for a first implementation. We chose to implement features that would have most impact on the architecture as compared to a H.264 video decoder. As a result, all LCU and CU configurations are supported. Similarly, the prediction supports all PU configurations and all intra and inter-prediction modes. The tranform also supports all the square and non-square TU sizes. Among the loop filters, only the deblocking filter could be implemented due to time limitations.

The chip interfaces with a Xilinx FPGA which interfaces with a 256 MB DDR3 SDRAM and the display. The DRAM access latency was identified as a key issue affecting the architecture of the decoder and several design decisions were based on simulation with real DRAM models to determine which option achieves the best throughput. A read-only cache is used for inter-prediction and separate pipeline stages is used to write back decoded pixels to the DRAM. The block of pixels processed in one pipeline stage is called largest pipeline unit (LPU). LPU size depends on LCU size as shown in Table 1.2.

Intra-prediction within an LPU requires reconstruction (prediction + residue) of previous pixels in the same LPU. This creates a dependency between prediction and inverse transform because of which, the inverse transform runs one pipeline stage before prediction. The entropy decoder generates motion vectors and sends requests to the cache/DRAM while the response is read by the prediction block. The variable

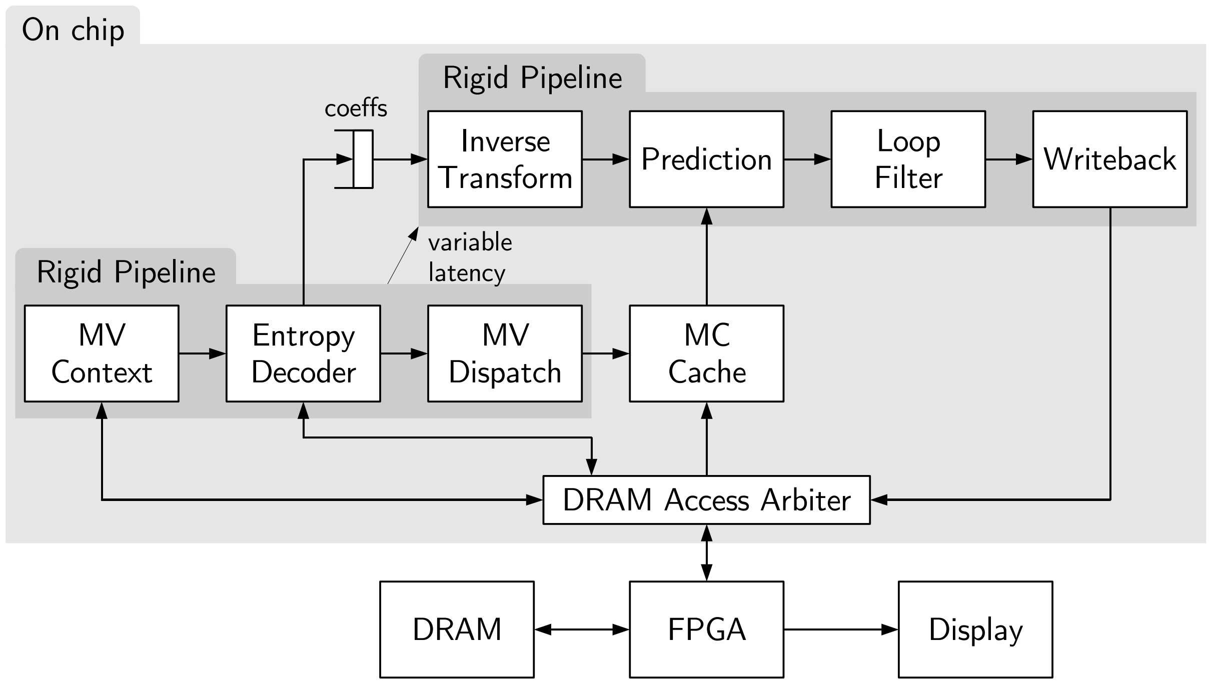


Figure 1-3: Simplified architecture of HEVC decoder with two rigid pipelines

DRAM latency is thus manifest in the interface between entropy decoder and prediction. To account for this variable latency, the decoder architecture is divided into two rigid pipeline groups that connect to each other through elastic links. This is shown in Figure 1-3. The latency between the two pipelines is dictated by the cache hit-rate and DRAM latency.

1.3 Contributions of this work

The key contributions in this work are in identifying challenges in hardware implementation for the HEVC inverse transform and deblocking filter and in providing solutions for them. For the inverse tranform, the major challenges are the high computation cost of large transforms, large transpose memory and varying processing latencies for different transform unit sizes. These are effectively addressed in this work through architecture and circuit techniques. In deblocking filter, complex dependencies make it very difficult to achieve the required throughout and an architecture is described to handle that. Bluespec SystemVerilog (BSV) [7] is used as the hardware description language and the BSV methodology of guarded atomic actions is found to be very useful in generating a concise, easy-to-debug hardware description.

We now describe the architecture of the inverse transform and deblocking filter blocks.

Chapter 2

Inverse Transform

The inverse transform block in the decoder computes the prediction error (residue) from its quantized transform values. In the encoder, the difference between the input pixels and their predicted values computed through inter-frame or intra-frame prediction is the residue. A two-dimensional transform of the residue is computed, then the transform values (coefficients) are quantized and then entropy encoded. The decoder performs the reverse operation. The quantized coefficients are parsed from the bitstream by the entropy decoder and then scaled back (dequantized). The residue is then computed from the inverse transform of the dequantized coefficients. This process is shown in Figure 2-1.

The two-dimensional transform is usually a DCT operation as it concentrates most of the signal energy in the low index DC coefficients. After the quantization, most of the small AC coefficients become zero, resulting in a sparse matrix of quantized coefficients. The 2-D matrix is reordered into a 1-D vector with a zig-zag mapping so that the vector has all the non-zero coefficients at the beginning followed by a long tail of zero coefficients. The entropy encoder can then efficiently compress this information. Figure 2-2 shows a histogram of the fraction of non-zero coefficients present in a typical video sequence encoded by HEVC. We see that most of the transforms have close to zero coefficients.

It should be noted that although entropy coding is a lossless compression, quantization is inherently lossy. So the coefficients, and consequently the residues, at the

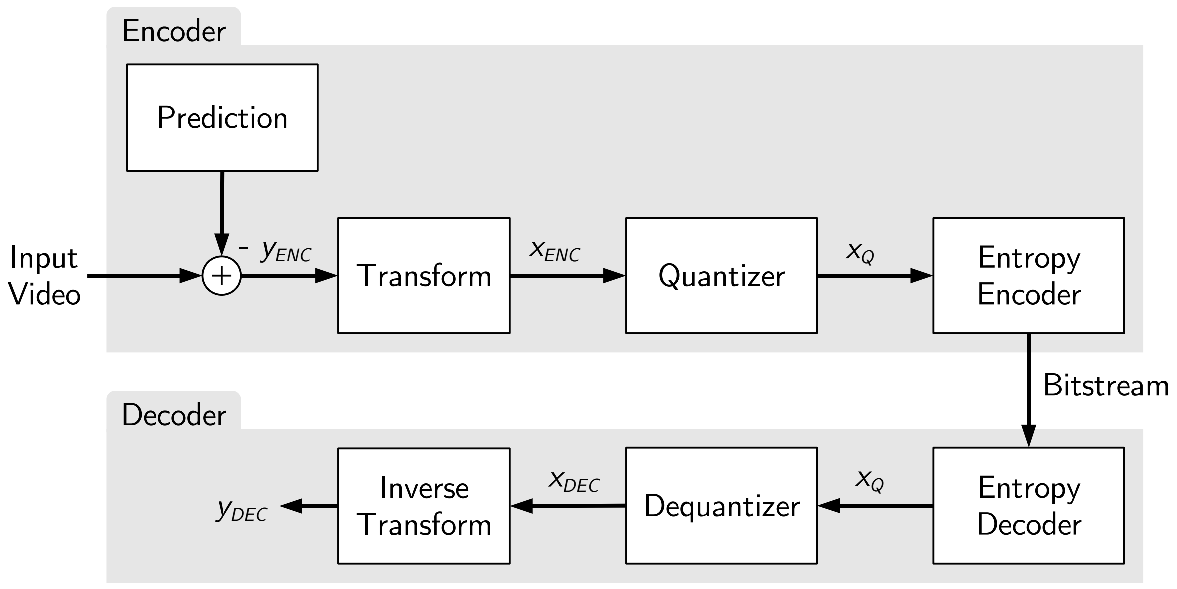


Figure 2-1: Residue coding and decoding (y = residues, x = coefficients)

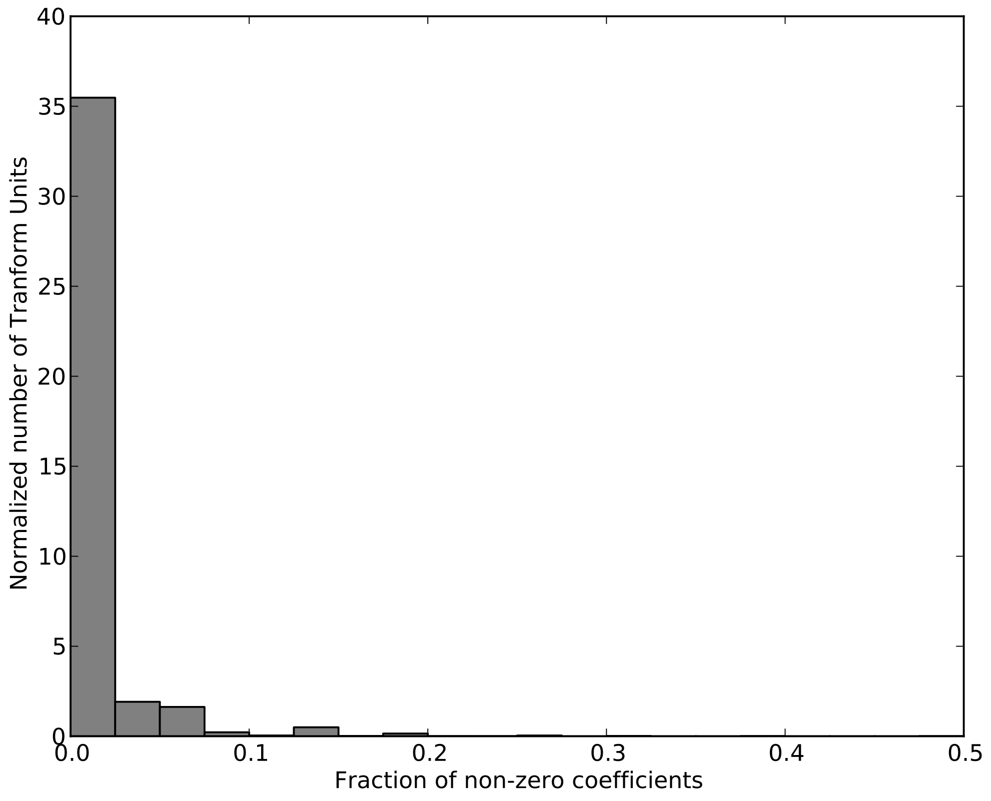


Figure 2-2: Normalized histogram of fraction of non-zero coefficients in HEVC transform units

encoder and decoder side are not equal, i.e. $x_{ENC} \neq x_{DEC} \Rightarrow y_{ENC} \neq y_{DEC}$.

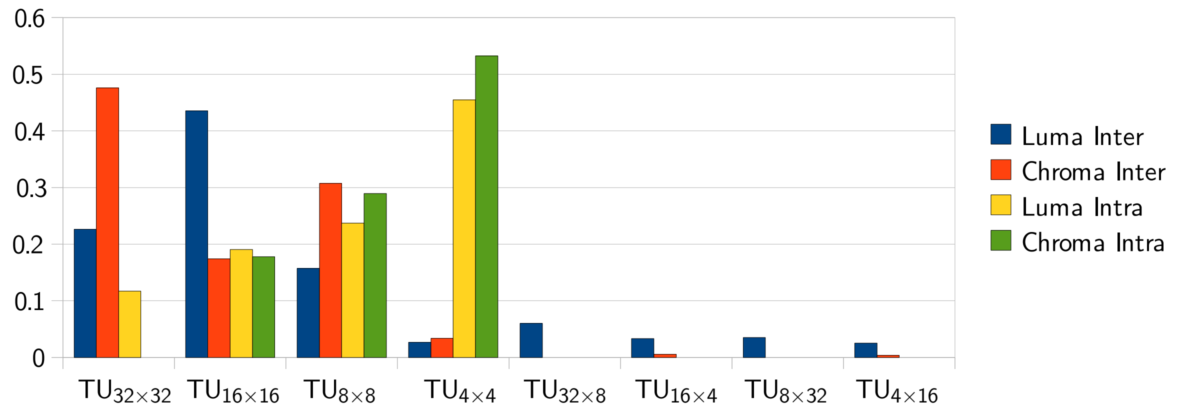
Section 2.1 describes the HEVC Inverse Transform specification in detail. The main considerations for circuit implementation of the transform are also described. Section 2.2 elaborates on the system requirements for the inverse transform block and the architecture used to meet them.

2.1 HEVC Inverse Transform

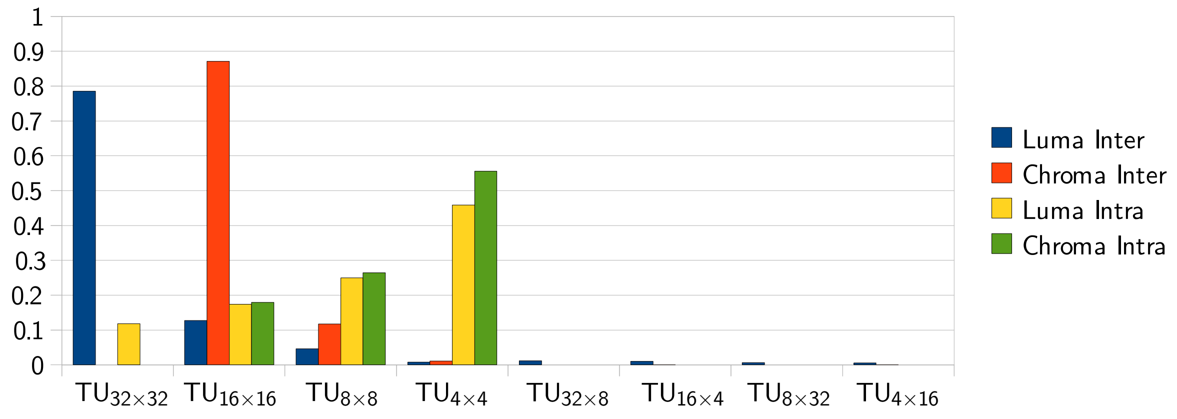
Starting from the Coding Unit (CU), the residual quad-tree is signalled using split flags similar to the CU quad-tree. However, unlike the CU quad-tree, a node in the residual quad-tree can have non-square child nodes. For example, a $2N \times 2N$ node may be split into four square $N \times N$ child nodes or four $2N \times 0.5N$ nodes or four $0.5N \times 2N$ nodes depending upon the prediction unit shape. The non-square nodes may also be split into square or non-square nodes.

HEVC HM-4.0 uses 8 Transform Unit (TU) sizes - $TU_{32\times32}$, $TU_{16\times16}$, $TU_{8\times8}$, $TU_{4\times4}$, $TU_{32\times8}$, $TU_{8\times32}$, $TU_{16\times4}$, and $TU_{4\times16}$. The TU size depends on TUDepth, CUSize and PUSize as shown in Table 2.1. The corresponding chroma TU sizes can be obtained from the same tables assuming half the CU size, except when the luma TU is $TU_{4\times4}$ in which case chroma is also $TU_{4\times4}$ equivalent to merging 4 luma TU's into an 8×8 block. For CU_8, $TU_{4\times4}$ is the only valid chroma TU size. The distribution of TU sizes observed in a test video sequence for all three LCU configurations is shown in Figure 2-3. Also, for LCU_{64}, the mean-square energy in all coefficient positions for TU's is shown in Figure 2-4. The energy is normalized to the the maximum energy in the TU. We can see that most of the energy is concentrated around the DC coefficient.

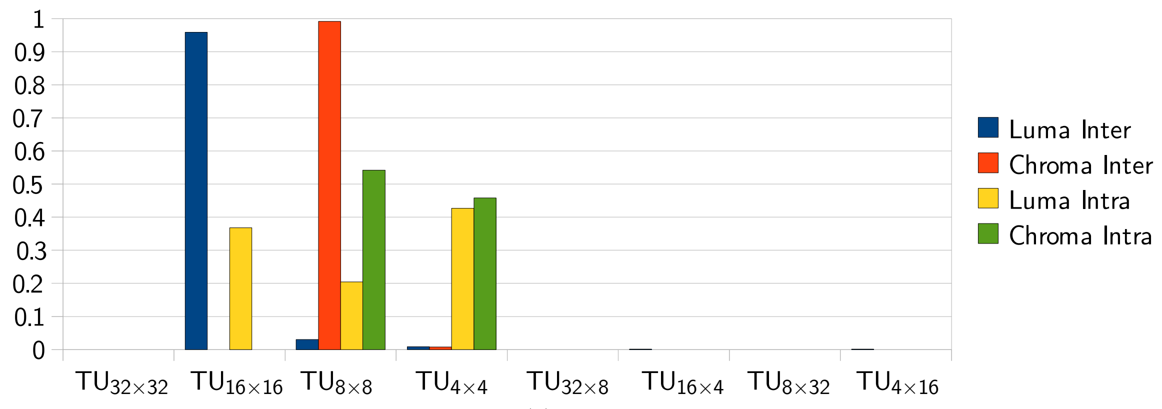
HEVC uses the orthogonal type-2 Discrete Cosine Transform with signed 8-bit



(a) LCU_{64}



(b) LCU_{32}



(c) LCU_{16}

Figure 2-3: Normalized distribution of TU sizes for OldTownCross (3840×2160) for all LCU configurations. Intra CU's do not use non-square TU's.

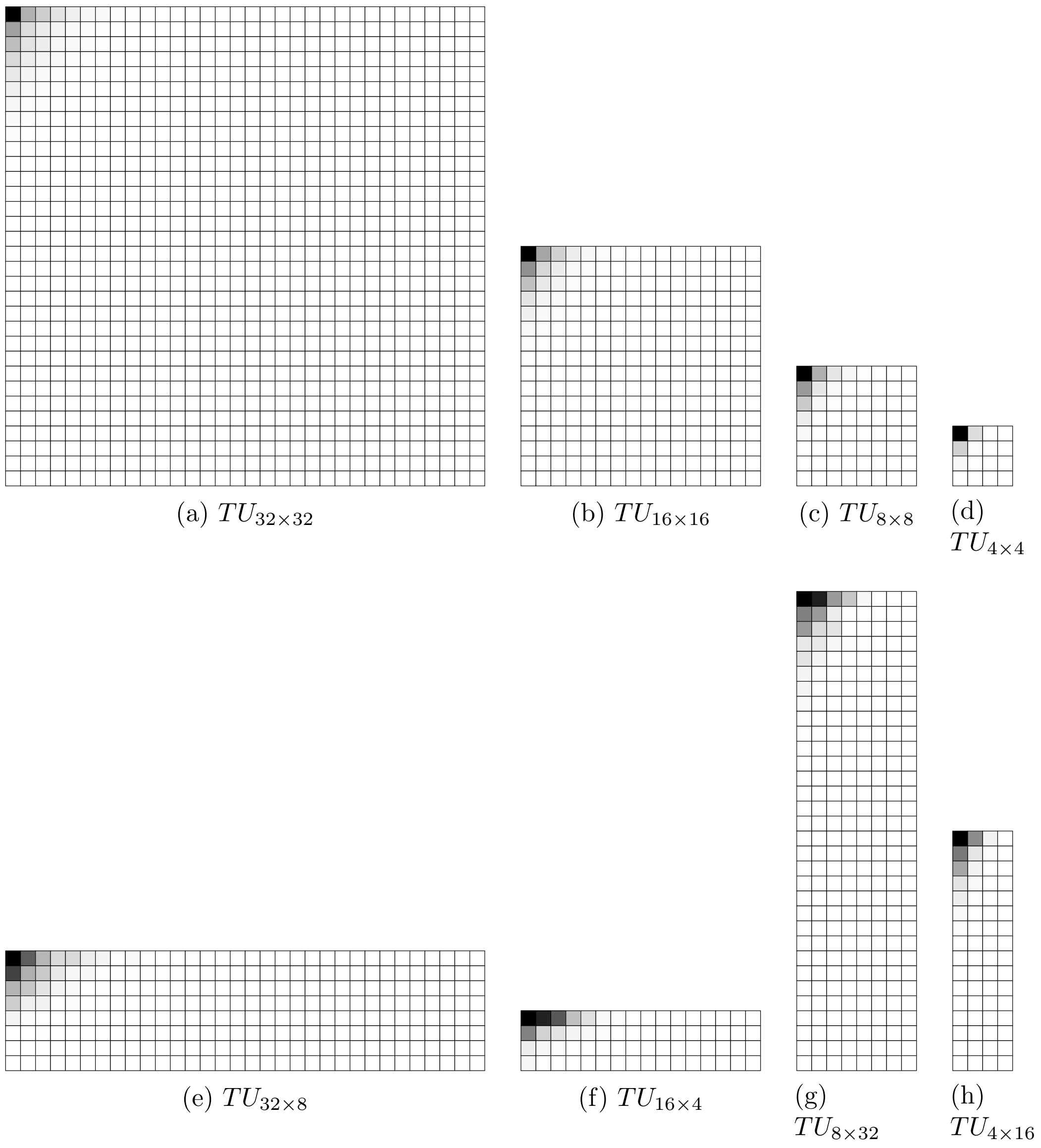


Figure 2-4: Normalized mean-square energy for coefficients in all TU's for OldTown-Cross (3840×2160) encoded with LCU_{64}. Darker pixels denote more signal energy. The energy is concentrated around the origin and most of the energy is in the DC coefficient.

TUDepth	CU_8	CU_{16}	CU_{32}	CU_{64}
0	$TU_{8\times8}$	$TU_{16\times16}$	$TU_{32\times32}$	-
1	$TU_{4\times4}$	$TU_{8\times8}$	$TU_{16\times16}$	$TU_{32\times32}$
2	-	$TU_{4\times4}$	$TU_{8\times8}$	$TU_{16\times16}$

(a) $PU_{N\times N}, PU_{2N\times 2N}$

TUDepth	CU_8	CU_{16}	CU_{32}	CU_{64}
0	$TU_{8\times8}$	$TU_{16\times16}$	$TU_{32\times32}$	-
1	$TU_{4\times4}$	$TU_{4\times16}$	$TU_{8\times32}$	$TU_{32\times32}$
2	-	$TU_{4\times4}$	$TU_{4\times16}$	$TU_{8\times32}$

(b) $PU_{N\times 2N}, PU_{nL\times 2N}, PU_{nR\times 2N}$

TUDepth	CU_8	CU_{16}	CU_{32}	CU_{64}
0	$TU_{8\times8}$	$TU_{16\times16}$	$TU_{32\times32}$	-
1	$TU_{4\times4}$	$TU_{16\times4}$	$TU_{32\times8}$	$TU_{32\times32}$
2	-	$TU_{4\times4}$	$TU_{16\times4}$	$TU_{32\times8}$

(c) $PU_{2N\times N}, PU_{2N\times nU}, PU_{2N\times nD}$

Table 2.1: Luma TU sizes for different PU Sizes

coefficients. The 32-pt inverse DCT (IDCT) operation is given by [8]

$$y_{DEC}[m] = \sum_n x_{DEC}[n] \cdot g\_aiT32[m][n] \tag{2.1}$$

$$g\_aiT32[m][n] = \text{round}\left(128\ k_m \cos\left(\frac{m\left(n+\frac{1}{2}\right)\pi}{32}\right)\right) \tag{2.2}$$

$$m, n = 0, 1, \ldots, 31$$

$$k_m = \begin{cases} 1/2 & \text{if } m = 0 \\ 1/\sqrt{2} & \text{if } m \neq 0 \end{cases}$$

The round() function is chosen so as to keep the matrix orthogonal. This ensures that the DCT and IDCT matrices are just transposes of each other enabling an encoder, which must perform DCT in the forward path and IDCT in the reconstruction path, to reuse the same hardware for both. The complete 32-pt DCT matrix is at [9]. The 16-pt, 8-pt and 4-pt DCT matrices are subsampled from the 32-pt matrix.

$$g\_aiT16[m][n] = g\_aiT32[m][2n] \qquad m,n = 0,1,\ldots,15 \tag{2.3}$$

$$g\_aiT8[m][n] = g\_aiT32[m][4n] \qquad m,n = 0,1,\ldots,7 \tag{2.4}$$

$$g\_aiT4[m][n] = g\_aiT32[m][8n] \qquad m,n = 0,1,\ldots,3 \tag{2.5}$$

For certain intra-prediction modes, HEVC has adopted Discrete Sine Transform (DST) for compressing the residue. Intra-prediction essentially extrapolates the previously decoded pixels on the edge of the current Prediction Unit (PU) to generate a prediction. This results in a prediction error that increase from the edge of the PU which is better modeled as a sine function rather than co-sine [10]. So, DST is used for column transforms with vertical intra modes and for row transforms with horizontal intra modes. A simplified 4-pt DST matrix has been adopted by the standard and so, only 4×4 TU's in intra-predicted CU's may use the DST.

As compared to H.264/AVC, the HEVC inverse transform is highly compute intensive. This is the result of two factors:

1. Large transform sizes: The largest transform in H.264 is an 8-pt DCT whereas HEVC can have up to a 32-pt DCT. A brute force matrix multiplication for an 8-pt DCT requires 8 multiplies per input or output pixel as compared to 32 multiples for a 32-pt DCT - a $4\times$ complexity increase.

2. Higher precision transform: This is best seen in (2.6) and (2.7) which compare the 8-pt DCT matrices for H.264 and HEVC. The H.264 matrix uses 5-bit precision as compared to 8-bit precision for HEVC. The 5-bit constant multiplies can be implemented with 2 shift and adds, while the HEVC coefficients would require 4 of them with a canonical signed digit multiplication.

Together, both these factors result in $8\times$ computational complexity of the HEVC transforms compared to H.264.

$$DCT_{8,\mathrm{H.264}} = \begin{bmatrix} 8 & 8 & 8 & 8 & 8 & 8 & 8 & 8 \\ 12 & 10 & 6 & 3 & -3 & -6 & -10 & -12 \\ 8 & 4 & -4 & -8 & -8 & -4 & 4 & 8 \\ 10 & -3 & -12 & -6 & 6 & 12 & 3 & -10 \\ 8 & -8 & -8 & 8 & 8 & -8 & -8 & 8 \\ 6 & -12 & 3 & 10 & -10 & -3 & 12 & -6 \\ 4 & -8 & 8 & -4 & -4 & 8 & -8 & 4 \\ 3 & -6 & 10 & -12 & 12 & -10 & 6 & -3 \end{bmatrix} \tag{2.6}$$

$$DCT_{8,\mathrm{HEVC}} = g\_aiT8 = \begin{bmatrix} 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 \\ 89 & 75 & 50 & 18 & -18 & -50 & -75 & -89 \\ 83 & 36 & -36 & -83 & -83 & -36 & 36 & 83 \\ 75 & -18 & -89 & -50 & 50 & 89 & 18 & -75 \\ 64 & -64 & -64 & 64 & 64 & -64 & -64 & 64 \\ 50 & -89 & 18 & 75 & -75 & -18 & 89 & -50 \\ 36 & -83 & 83 & -36 & -36 & 83 & -83 & 36 \\ 18 & -50 & 75 & -89 & 89 & -75 & 50 & -18 \end{bmatrix} \tag{2.7}$$

The complete inverse transform operation ("Scaling, transformation and array construction process prior to deblocking filter process" [2]) for each TU is as follows:

1. Inverse scanning: Reorder the 1-D transform coefficients into 2-D matrix
2. Scaling: Dequantize the coefficients depending on the quantization parameter QP to get a scaled TU matrix
3. Inverse Transform:
    - Column transform: Perform 1-D inverse transform along all columns of the scaled TU matrix to get an intermediate TU matrix
    - Row transform: Perform 1-D inverse transform along all rows of the intermediate TU matrix to get the residue

As specific profiles and levels have not been decided for the HEVC standard, we chose the following requirements for our implementation:

1. All transform sizes and types: We chose to support all the square and non-square transform sizes and both DCT and DST.
2. Quad-Full HD at 200 MHz: This translates to a throughput requirement of 2 pixels/cycle from (2.10) with $fW = 3840, fH = 2160, fR = 30Hz, F_{CLK} = 200MHz$.

Also, the inverse scanning process is performed by the entropy decoder block, so the transform block only performs the scaling and the actual transform.

$$\begin{aligned}
\text{Pixels per picture} &= \text{Y pixels} + \text{Cb pixels} + \text{Cr pixels} \\
&= fW \cdot fH + fW/2 \cdot fH/2 + fW/2 \cdot fH/2 \\
&= 1.5fW \cdot fH && (2.8) \\
\text{Pixels per second} &= \text{Pixels per picture} \cdot \text{Frame rate} \\
&= 1.5fW \cdot fH \cdot fR && (2.9) \\
\text{Pixels per cycle} &= \text{Pixels per second/Clock frequency} \\
&= 1.5fW \cdot fH \cdot fR/F_{CLK} && (2.10)
\end{aligned}$$

2.2 Inverse Transform Architecture

The main challenges in designing a HEVC inverse transform block are listed below.

1. Increased computational complexity as detailed in Section 2.1 result in large circuit area. A brute-force single cycle 32-pt IDCT was implemented with all multiplications as Booth encoded shift-and-add and found to require 145 kgates on synthesis in the target technology. Hence, aggressive optimizations that exploit various properties of the transform matrix are a must to achieve a reasonable area. A solution is presented that performs partial matrix multiplication

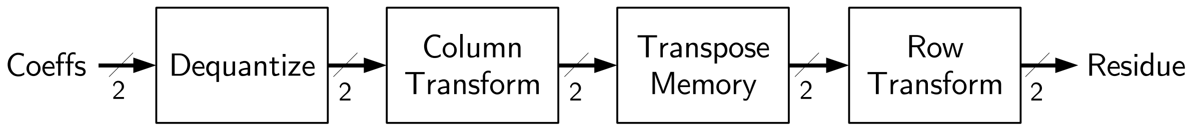
to compute the DCT over multiple cycles. Also, large generic mutipliers are avoided by using some interesting properties of the matrix.

2. A 16 kbit transpose memory is needed for the 32×32 transform. In H.264 decoders, transpose memory for inverse transform are usually implemented as a register array as they are small. For HEVC, a 16 kbit register array with column-write and row-read requires about 125 kgates on synthesis. To reduce area, SRAM's which are denser than registers are used in this work. However, they are slower than registers and not as flexible in read-write patterns and so, a small register-based cache is used to get the maximum throughput.

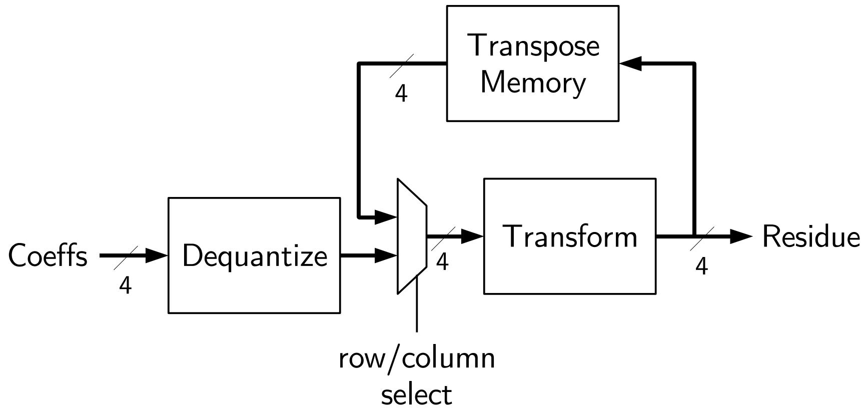
3. TU's of different sizes take different number of cycles to finish processing. A pipelined implementation that manages full concurrency with multiple TU's in the pipeline having varying latencies is very challenging to design. The transpose operation is not amenable to an elastic FIFO-based pipeline which would normally be ideal for varying latencies. Designing for the worst case latency of the 32×32 transform would result in a lot of wasted cycles or wasted buffer size.

The presented solution meets all the above challenges and can be extended to larger transforms, different non-square TU sizes and higher throughputs.

In general, two high-level architectures are possible for a 2 pixel/cycle inverse transform [11]. The first one, shown in Figure 2-5a uses separate blocks for row and column transforms. Each one has a throughput of 2 pixel/cycle and operates concurrently. The dependency between the row and column transforms (all columns of the TU must be processed before the row transform) means that the two must process different TU's at the same time. The transpose memory must have one read and one write port and hold two TU's - in the worst case, two $TU_{32\times32}$'s. Also, the two TU's would take different number of cycles to finish processing. For example, if a $TU_{8\times8}$ follows $TU_{16\times16}$, the column transform must remain idle after processing the smaller TU as it waits for the row transform to finish the larger one. It can begin processing the next TU but managing several TU's in the pipeline at the same time



(a) Separate row and column transforms



(b) Shared transform block

Figure 2-5: Possible high-level architectures for inverse transform. Bus-widths are in pixels.

will require careful management to avoid overwriting and ensure full concurrency (no stalls in the pipeline).

With these considerations, the second architecture, shown in Figure 2-5b is preferred. This uses a single transform block capable of 4 pixels per cycle for both row and column transform. The block works on a single TU at a time, processing all the columns first and then the rows. Hence, the transpose memory needs to hold only one TU and can be implemented with a single port SRAM since row and column transforms do not occur concurrently.

The next three sections describe the micro-architecture of each of the blocks - dequantizer, transpose memory and transform - in detail.

2.2.1 Dequantizer

HEVC HM-4.0 adopts a flat quantization matrix i.e. all coefficients in the TU are scaled equally. The scaling operation for each quantized parameter x_Q to get the scaled coefficient x depends on the quantization parameter QP and TU size $TU_W \cdot TU_H$ as follows:

$$\begin{aligned}
nS &= \sqrt{TU_W \cdot TU_H} \\
g\_auIQ[6] &= [40, 45, 51, 57, 64, 72] \\
QP_{SCALE} &= g\_auIQ[\text{QP mod } 6] \\
QP_{SHIFT} &= \text{QP}/6 \\
x &= \left(x_Q \cdot QP_{SCALE} \cdot 2^{QP_{SHIFT}} + (nS/2)\right)/nS
\end{aligned}$$

For chroma pixels, a higher value of QP is used as higher compression is possible with less perceived distortion. The scaling with QP_{SCALE} is implemented using a generic multiplier while scaling with $2^{QP_{SHIFT}}$ and nS are simple shift left and shift right respectively.

2.2.2 Transpose Memory

The transform block uses a 16-bit precision input for both row and column transforms. The transpose memory must be sized for $TU_{32\times32}$ which means a total size of $16 \times 32 \times 32 = 16.4$ kbits. In comparison, H.264 decoder designs require a much smaller transpose memory - $16 \times 8 \times 8 = 1$ kbit. A 16.4 kbit memory with the necessary read circuit for the transpose operation is prohibitively large (125 kgates) when implemented with registers and muxes. Hence, an SRAM implementation is needed.

The main disadvantage of the SRAM is that it is less flexible than registers. A register array allows reading and writing to arbitrary number of bits at arbitrary locations, although very complicated read(write) patterns would lead to a large output(input) mux size. The SRAM read or write operation is limited by the bit-width of its port. A single-port SRAM allows only one operation, read or write, every cycle. Adding extra ports is possible at the expense of significant area increase in advanced technology nodes such as 45nm.

The proposed solution uses a 4 pixel wide SRAM implemented as 4 single-port banks of 4096 bits each with a port-width of 1 pixel. The pixels in $TU_{32\times32}$ are

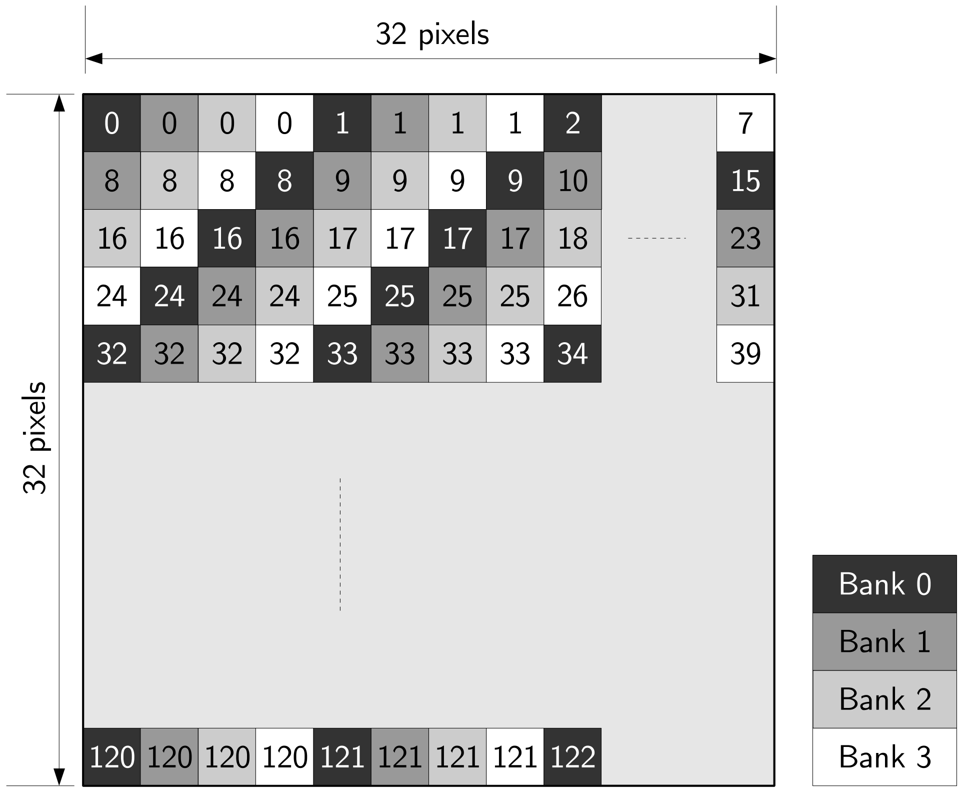
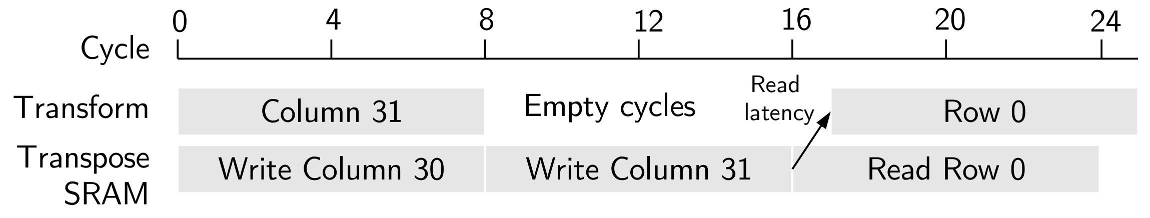


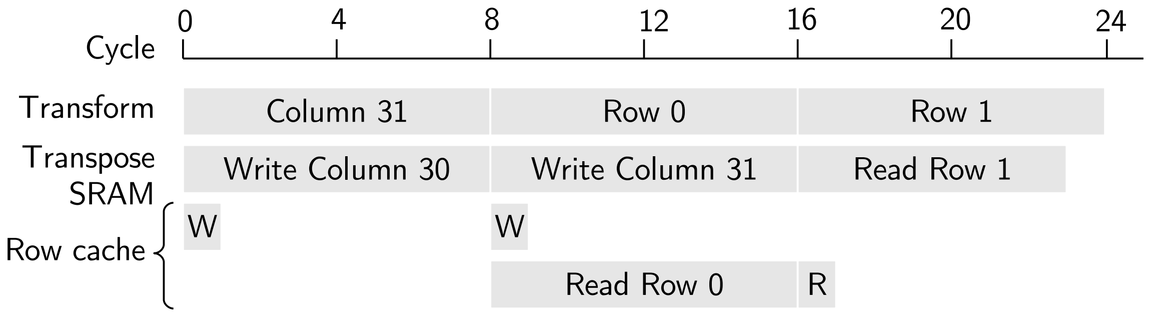
Figure 2-6: Mapping a $TU_{32\times32}$ to 4 SRAM banks for transpose operation. The color of each pixel denotes the bank and the number denotes the bank address.

mapped to locations in the 4 banks as shown in Figure 2-6. By ensuring that 4 adjacent pixels in any row or column sit in different SRAM banks, it is possible to write along columns and read along rows by supplying different addresses to the 4 banks.

After a 32-pt column transform is computed, it takes 8 cycles for the result to be written to the transpose SRAM, during which time the transform block processes the next column. This is shown in cycles $0-7$ in Figure 2-7a where result of column 30 is written to the SRAM while the transpose block works on column 31. However, when the last column is processed, the transform block must wait for it to be written to the SRAM before it can begin processing the row. This results in a delay of 9 cycles for $TU_{32\times32}$. In general, for a $TU_W \times TU_H$ TU, this delay is equal to $TU_W/4+1$ cycles. This delay is avoided through the use of a row cache that stores the first $TU_W + 4$ pixels in registers. This enables full concurrency as shown in Figure 2-7b. The first pixel in each column is saved to the row cache so that the first row can be read from



(a) Pipeline stall due to transpose SRAM delay for $TU_{32\times32}$



(b) Row caching to avoid stall

Figure 2-7: Enabling full concurrency with register + SRAM transpose memory

the cache while the last column is being stored in the SRAM. The pixel locations that need to be cached for different TU's is shown in Figure 2-8.

2.2.3 Transform Block

As mentioned in Section 2.1, the HEVC DCT is 8× more complex than the H.264 DCT. Two interesting properties of the DCT matrix are used to derive an area-efficient implementation.

1. As seen in (2.7) even and odd rows of the matrix are symmetric and anti-symmetric. For the 8-pt DCT

$$g\_aiT8[m][2n] = g\_aiT8[7-m][2n]$$
$$g\_aiT8[m][2n+1] = -g\_aiT8[7-m][2n+1] \quad m, n = 0 \ldots 3$$

For a general N-pt DCT

$$g\_aiTN[m][n] = \begin{cases} g\_aiTN[N-1-m][n] & \text{if } n \text{ even} \\ -g\_aiTN[N-1-m][n] & \text{if } n \text{ odd} \end{cases} \tag{2.11}$$

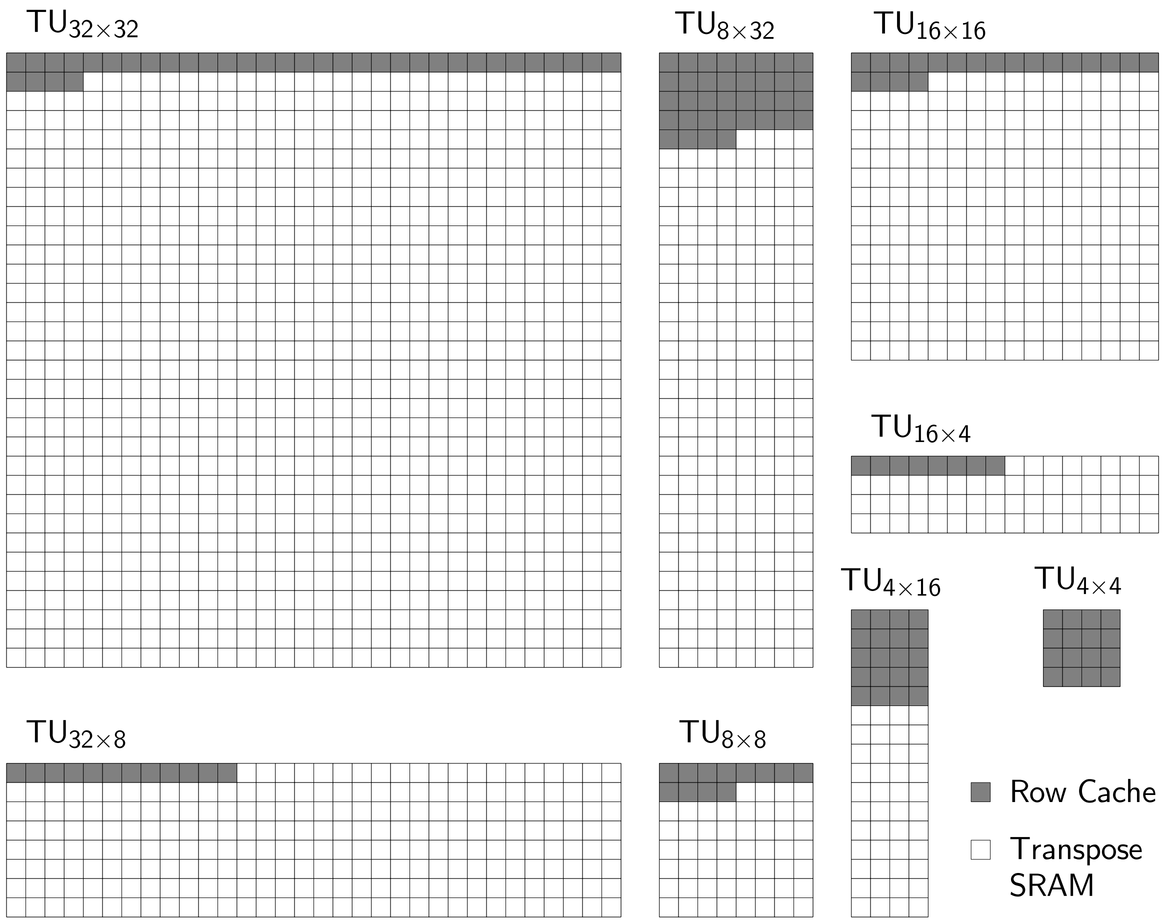


Figure 2-8: Location of pixels cached in registers for all TU cases. For $TU_{4\times4}$, the row cache is itself used as a transpose memory.

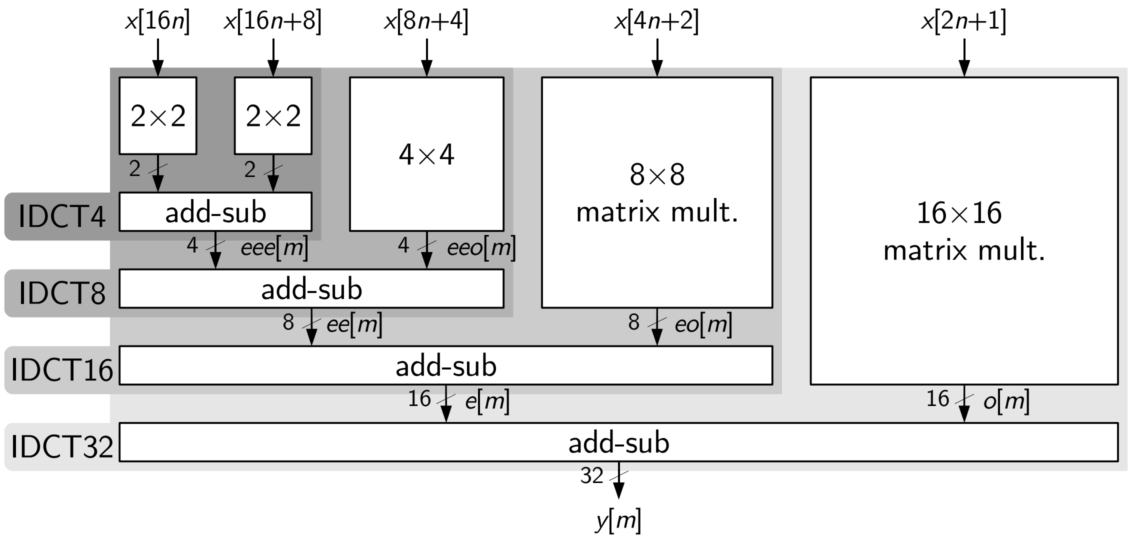


Figure 2-9: Recursive decomposition of 32-pt IDCT using smaller IDCT's (A picturization of partial-butterfly transform in the reference software)

2. The N-pt DCT matrix has only less than N unique coefficients differing only in sign. For example, the 8-pt matrix has only 7 unique coefficients - 18, 36, 50, 64, 75, 83 and 89.

The first property is used to decompose the matrix into even and odd parts to create the partial butterfly structure shown in Figure 2-9 used in HM-4.0 reference software. The decomposition is derived for the 32-pt transform in Appendix A.

The second property can now be used to further optimize the odd matrix multiplications in Figure 2-9. The 16×16 matrix multiplication takes the odd numbered inputs $x[2n + 1]$. In a 4-pixel per cycle case, only 2 of these inputs are available per cycle. So, it is enough to perform a partial 2×16 matrix multiplication every cycle and accumulate the outputs over 8 cycles. In general, this would require 32 multipliers each with one input from one of $x[2n + 1]$ and the other input from an 8-entry look-up table (one entry per cycle). The second property allows us to use 32 constant multipliers and simply multiplex their outputs. This is demonstrated for the 4×4 even-even-odd(eeo) matrix multiplication with 1 input pixel per cycle throughput.

Matrix multiplication	Area for generic implementation (kgates)	Area exploiting unique operations (kgates)	Area savings
4×4 for *eeo*	10.7	7.3	32%
8×8 for *eo*	23.2	13.5	42%
16×16 for *o*	46.7	34.4	26%

Table 2.2: Area reduction by exploiting unique operations

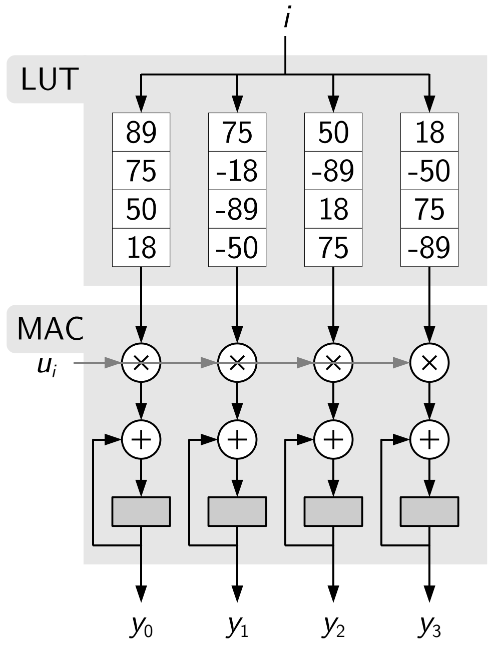
The matrix multiplication with input $u = x[8n+4]$ and output $y = eeo[m]$ in Figure 2-9 to be implemented is:

$$\begin{bmatrix} y_0 & y_1 & y_2 & y_3 \end{bmatrix} = \begin{bmatrix} u_0 & u_1 & u_2 & u_3 \end{bmatrix} \begin{bmatrix} 89 & 75 & 50 & 18 \\ 75 & -18 & -89 & -50 \\ 50 & -89 & 18 & 75 \\ 18 & -50 & 75 & -89 \end{bmatrix} \tag{2.12}$$

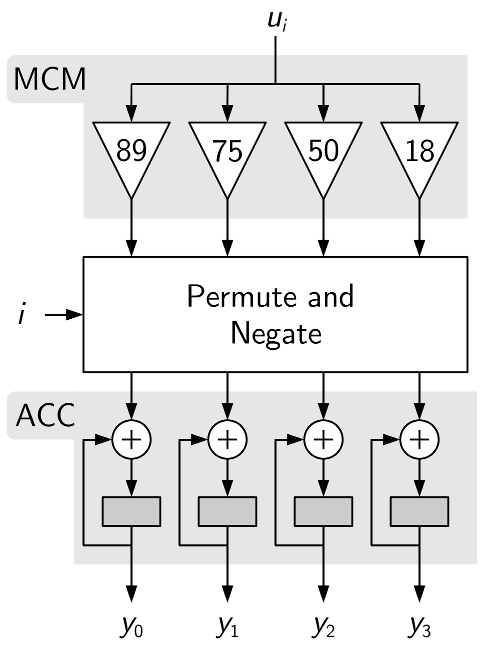
For a generic matrix, each column would be stored in a 4-entry look-up table. The ith entries of the 4 column LUT's are read every cycle and multiplied with u_i and the products accumulated as shown in Figure 2-10a. However, observing that the matrix has only 4 unique coefficients - 89, 75, 50 and 18, these multipliers can be implemented as shift-and-adds and the outputs permuted as per the row index. This is shown in Figure 2-10b. The 4 multipliers can be further optimized using Multiple Constant Multiplication, a problem studied extensively in [12], [13], [14]. For this implementation, an online MCM tool [15] was used to generate optimized multipliers. The 16×16 odd (o) matrix multiplication requires only 13 adders in all and the 8×8 even-odd (eo) matrix requires only 8 adders as shown in Appendix A. This enables more than 25% area reduction as listed in Table 2.2.

2.3 Results

The complete architecture of the inverse transform block is shown in Figure 2-11. The partial transform block includes the 4-pixel per cycle IDCT and IDST blocks. The transform coefficients along with TU information such as size, quantization parameter



(a) Generic implementation



(b) Exploiting unique operations

Figure 2-10: 4×4 matrix multiplication (2.12) with and without unique operations

and luma/chroma are input from the entropy decoder and the output is written to a residue SRAM to be read by the prediction module. Single-element FIFO's are used for pipelining. A $TU_{4\times4}$ immediately after a $TU_{32\times32}$ would result in a pipeline stall as it takes 8 cycles to write out the last row of TU_{32x32} to the residue SRAM while the first row of the $TU_{4\times4}$ would be ready in 4 cycles itself. A stall is avoided by using a separate 4-pixel 4-entry FIFO to writeback all $TU_{4\times4}$'s.

Breakdown of the synthesis area at 200MHz clock in TSMC 40nm technology is given in Table 2.3. Post-layout power estimation by the place-and-route tool using simulation waveforms for 3 LCU's of real video is 12.6 mW. This includes dynamic power (6.1 mW), clock network power (5 mW) and leakage (1.5mW). The author thanks Chao-Tsung for providing these numbers. On a Virtex-6 FPGA running at 25MHz, this design takes 12k LUT's, 3k registers and 4 BlockRAM's.

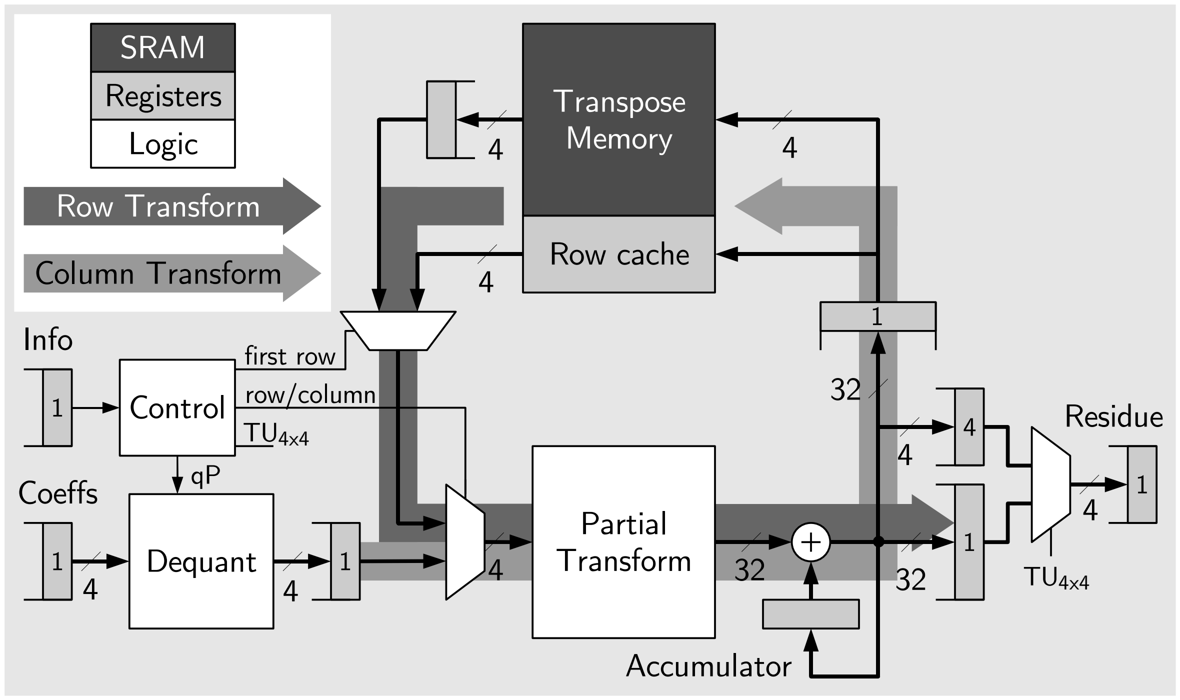


Figure 2-11: Architecture of inverse transform

Module	Logic area (kgates)	SRAM size (kbits)
Partial transform	71	0
Accumulator	5	0
Row cache	4	0
Transpose Memory	0	16.4
FIFO's	5	0
Dequant + Control	19	0
Total	104	16.4

Table 2.3: Area breakdown for inverse transform

Module	Logic area (kgates)
4-pt IDCT	3
Partial 8-pt IDCT	10
Partial 16-pt IDCT	24
Partial 32-pt IDCT	57
4-pt IDST + misc.	14

Table 2.4: Area for different transforms. Partial 32-pt IDCT contains all the smaller IDCT's

Chapter 3

Deblocking Filter

Quantization of transform coefficients in the encoder means that the decoder can perform only a lossy reconstruction. The quantization of the DC coefficient is particularly visible as blocking artifacts as seen in Figure 3-1. The block edges correspond to TU edges and this effect is worst in intra-frames which have a large prediction error. A deblocking filter was introduced in H.264 to overcome these problems. By smoothening the edges, the deblocking filter makes these artifacts less objectionable. Also, by providing a more accurate reconstruction of the original image, it improves coding efficiency [16].

The HEVC deblocking process works on a 8×8 pixel grid. First, all the vertical edges in the picture are processed followed by all the horizontal edges. Implementing this process in hardware would require a full-frame buffer which is too large for an on-chip memory ($12MB$ for Quad-Full HD resolution). In order to avoid excessive

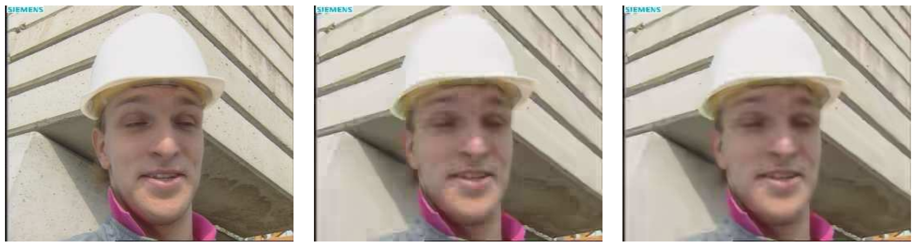

(a) Input picture (b) Blocking artifacts (c) After deblocking

Figure 3-1: Need for deblocking filter in video coding

latencies due to off-chip memory accesses and to fit deblocking in the system pipeline, the process is modified to operate in a horizontal raster scanned Largest Coding Unit (LCU) order. The dependencies across LCU boundaries are maintained through pixel buffers which enables an implementation that is bit-exact with the software reference.

3.1 HEVC Deblocking Filter

HEVC employs three types of deblocking filters - strong luma, weak luma and a chroma filter shown in Figure 3-2. The luma filters are 8-tap filters with the strong filter modifying 3 pixels on either side of the edge and the weak filter modifying 2 on either side. The chroma filter is a 4-tap filter that acts on 1 pixel on either side. For luma pixels, the choice between strong, weak and no filtering depends on two factors.

1. Coding decision - Intra-coded blocks and TU edges use strong filtering. For inter-coded blocks, the difference of motion vectors and reference frames on either side of the edge are used to choose the filter. The coding decision information is represented as a parameter called boundary strength from 0 to 4. A higher boundary strength results in stronger filtering.

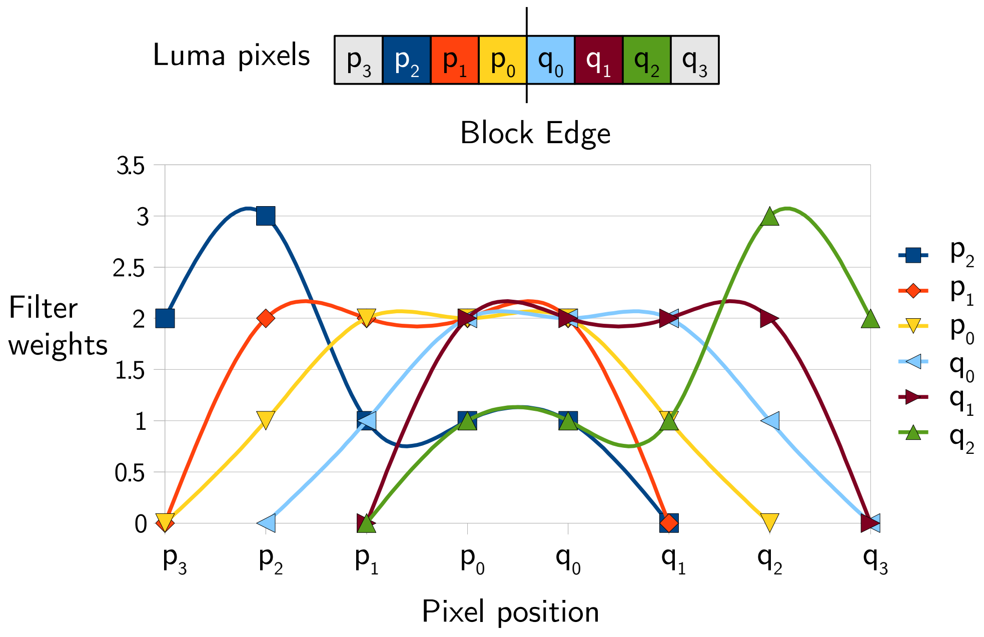
2. Blocking artifact - The pixel values around the edge and quantization parameter (QP) are used to estimate if the edge has blocking that is a coding artifact or a feature in the original image. If the second derivative of pixel values across the edge is greater than a threshold, it is deemed to be a feature that must be preserved by not filtering the edge. The threshold increases with QP i.e. a higher QP results in more edges being filtered.

The decision for chroma filtering is taken similarly. The filtering process can then be described as follows:

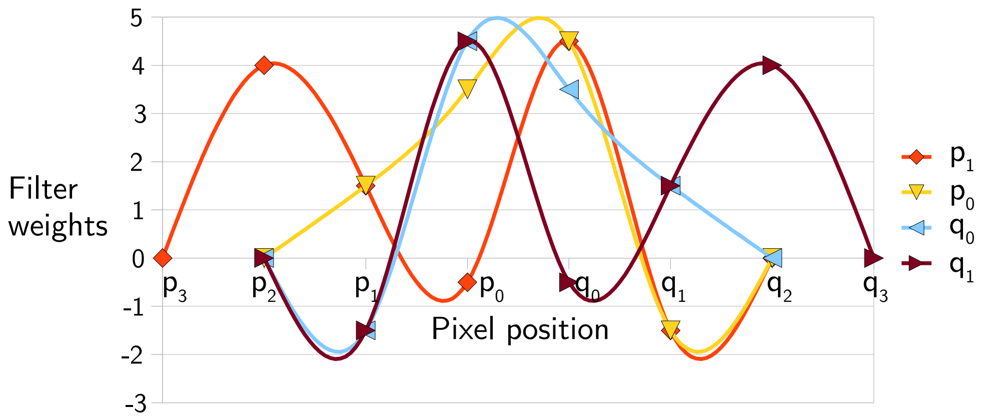
- For all 8-pixel vertical edges in the picture on an 8×8 grid,

    1. Check if edge is PU, TU or CU edge

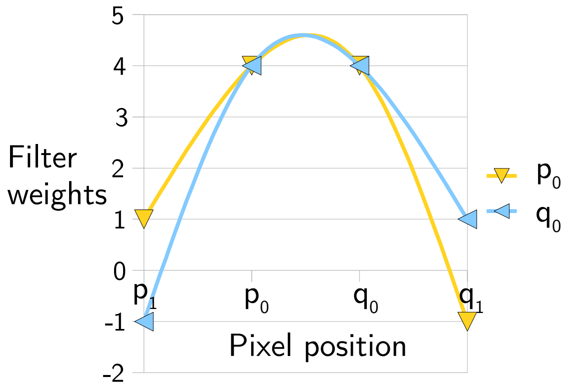
    2. If edge, compute boundary strength (bS). Else, exit loop.



(a) Strong Luma



(b) Weak Luma



(c) Chroma

Figure 3-2: Deblocking filters weights (scaled by 8) for all pixels around block edge

LPU count	Prediction		Deblocking		Writeback	
	LPU	SRAM	LPU	SRAM	LPU	SRAM
0	0	C				
1	1	B	0	C		
2	2	A	1	B	0	C
3	3	C	2	A	1	B
4	4	B	3	C	2	A
5	5	A	4	B	3	C

Table 3.1: Pipelining scheme for deblocking

3. If bS > 0, compute per-edge filter parameters. Else, exit loop.

4. Filter luma and chroma edges

- Repeat for all 8-pixel horizontal edges on 8×8 grid.

The luma and chroma filtering processes are independent of each other and can be parallelized.

3.2 Deblocking Filter Architecture

The deblocking filter block is part of a rigid pipeline consisting of prediction and DRAM writeback blocks. The size of pipeline buffers, called Largest Pipeline Unit (LPU) depends on LCU size as listed in Table 1.2. The pipeline is operated using three SRAM's connected to the processing block in a rotary fashion. For example, when prediction is processing LPU_2 in $SRAM_A$, deblocking is working on LPU_1 in $SRAM_B$ and LPU_0 is being transferred from $SRAM_C$ to the DRAM. Once all the blocks are done processing their respective LPU's, prediction moves to a new LPU_3 in $SRAM_C$, LPU_2 in $SRAM_A$ is deblocked and LPU_1 in $SRAM_B$ is written back. This process is shown in Table 3.1.

At any point of time, each processing block is connected to only one pipeline SRAM. All three blocks need to access pixels from the previous LPU and each block must maintain a buffer within itself for them. The buffer for pixels from LPU in the previous row is proportional to the picture width. Deblocking filter needs pixels from four rows above and so this top-row buffer needs to store 15K luma pixels and

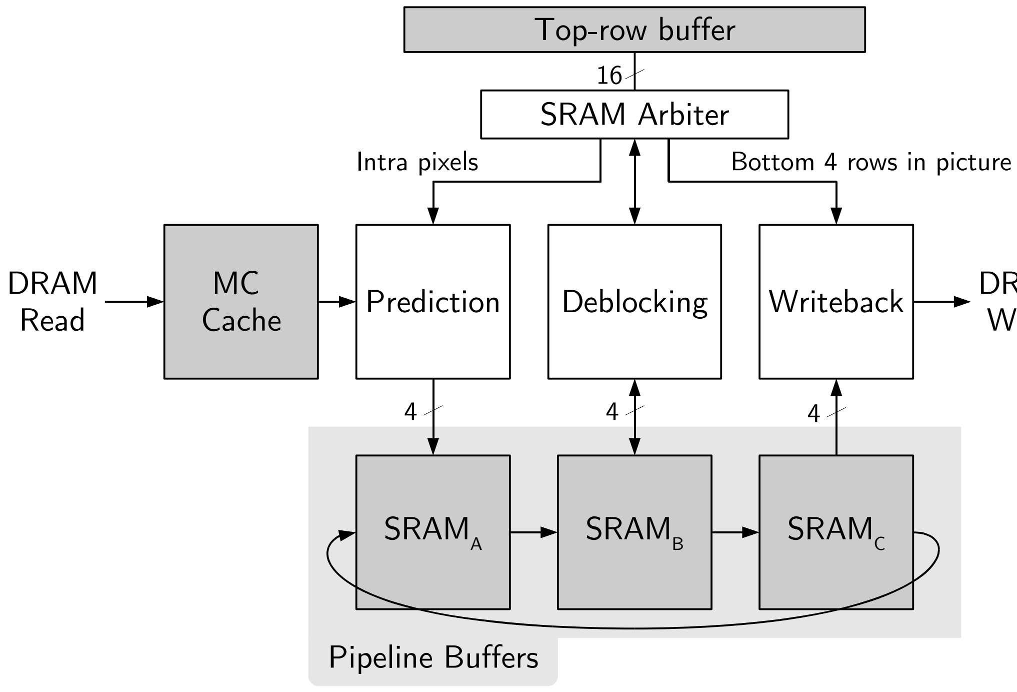


Figure 3-3: Pipelining scheme for prediction, deblocking and writeback with rotated pipeline buffers and shared top-row buffer

corresponding chroma pixels. This being a very large SRAM, the top-row buffer is shared between all three blocks as shown in Figure 3-3.

The top-level architecture of the deblocking filter is shown in Figure 3-4. Each of the blocks is described in detail in the subsequent subsections.

3.2.1 Pipeline buffer

The pipeline buffer in Figure 3-4 corresponds to one SRAM block of the pipeline buffer in Figure 3-3. The largest LPU corresponds to LCU_{64} and contains $64 \times 64 + 32 \times 32 + 32 \times 32 = 6144$ pixels. The pipeline buffer stores this data as 1536 32-bit entries, each entry consisting of 4 pixels. The prediction block stores 4 pixels along the rows to each entry in the SRAM while the deblocking writes back 4 pixels along the columns.

In the worst case of filtering all edges in the LPU, each pixel needs to be read and written at least once. Hence, achieving a throughput of 2 pixel/cycle would require a throughput of 4 pixel/cycle from the pipeline SRAM, which means that they must

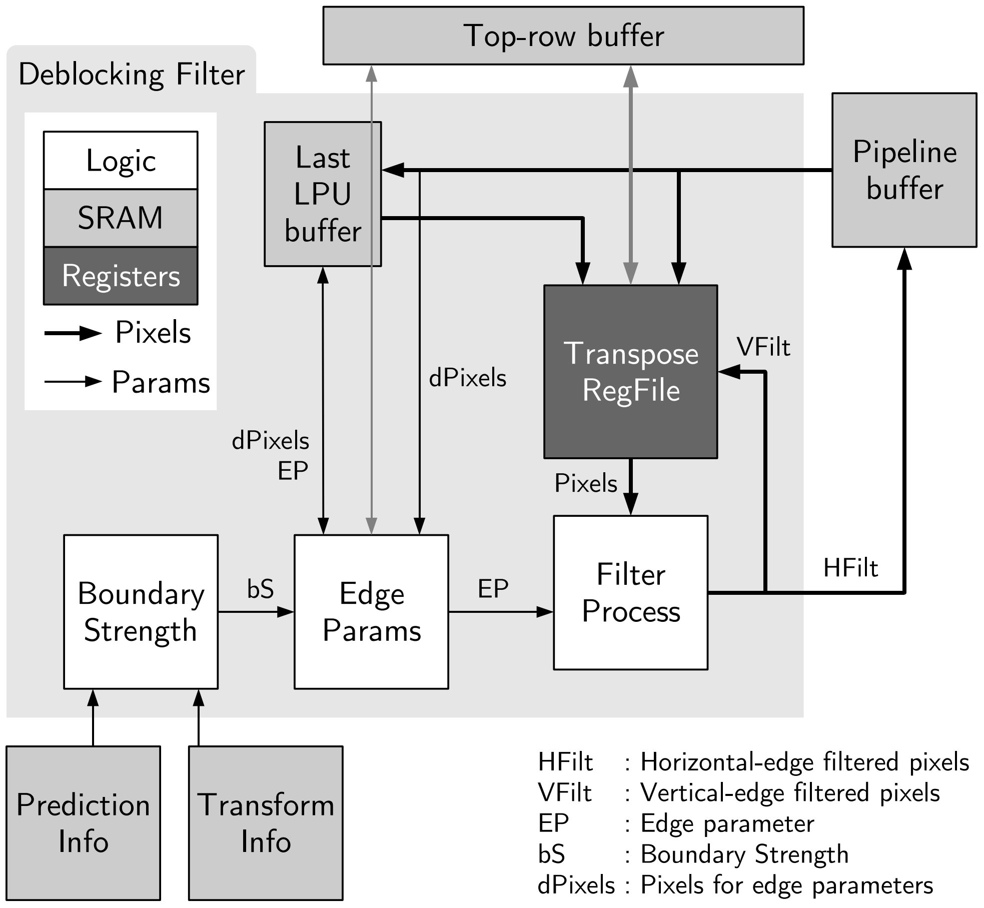


Figure 3-4: Top-level architecture of deblocking filter

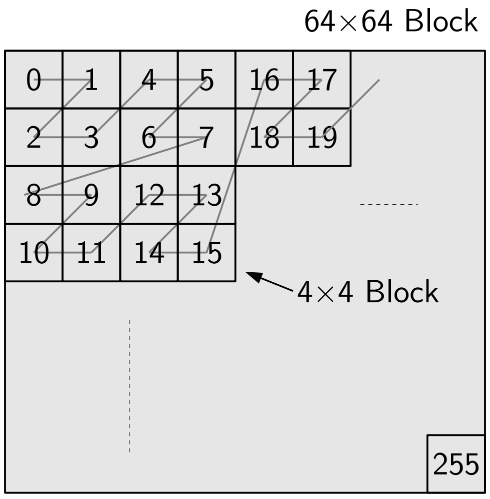


Figure 3-5: Double-Z order mapping for prediction info

be busy with a read or write every cycle. This being the most stringent constraint, it dictates most of the architecture decisions of the deblocking filter.

For the better code readability, the address to the SRAM is abstracted into position of 8×8 block, texture type (Y/Cb/Cr) and position of the 4-pixel lines within the block. The conversion to the physical address is a trivial bit-level manipulation that is accomplished inside a wrapper for the SRAM.

3.2.2 Prediction Info

This SRAM stores information such as prediction type (intra/inter), CU and PU size, intra-prediction direction, reference frame indices and motion vectors, etc. for the present LPU. Since the smallest PU is 4×4 pixels, this information is stored on a 4×4 basis in a double-Z order as shown in Figure 3-5. For the largest 64×64 LPU, the SRAM needs to store $16 \times 16 = 256$ entries. The double-Z index of each entry can be easily computed from its vertical and horizontal position using bit-level manipulations.

$$\text{Vertical position} = y[3:0]$$
$$\text{Horizontal posistion} = x[3:0]$$
$$\text{Double-Z order index} = \{y[3], x[3], y[2], x[2], y[1], x[1], y[0], x[0]\}$$

Prediction information for one row of 4×4 blocks above the current LPU is also required for boundary strength computation. This is also stored in this buffer making a total of 278 entries in the SRAM each entry containing 78 bits of information. This buffer is written to by the entropy decoder block and read by deblocking two stages later in the pipeline. So, this buffer is implemented using three rotated SRAM blocks similar to the pipeline buffer.

3.2.3 Transform Info

This SRAM stores transform information for the current LPU in a raster-scan order on a 4×4 basis. A total of 256 9-bit entries are stored in this buffer by the transform block. Each entry includes quantization parameter, coded block flag and two bits to indicate whether the left and top edges are TU edges. Since the transform block is two stages behind deblocking in the pipeline, the transform info buffer is implemented using three rotated SRAM blocks.

3.2.4 Top-row buffer

This buffer stores the last 4 rows of luma and chroma pixels from the LPU row above the current row. The size of this buffer is dictated by the maximum picture width supported by the design. For Quad Full HD video, this buffer must store $3840 \times 4 + 1920 \times 4 + 1920 \times 4 = 30720$ pixels. This is stored as 1920 128-bit entries with each entry containing 16 pixels. The SRAM also contains 120 entries for 480 edge parameters (EP) that are passed on to the next LPU row. The address map of the buffer is shown in Table 3.2. The 16-pixel entries store 4×4 pixel blocks.

Data type	Addresses
Y	[0, 959]
Cb	[960, 1439]
Cr	[1439, 1919]
EP	[1920, 2039]

Table 3.2: Address map for top-row buffer

3.2.5 Boundary Strength

The standard specifies the boundary strength (bS) of each 8-pixel edge as the maximum of the boundary strengths of each pixel on the edge. In the reference software, bS can take values from 0 to 4. However, it was found that some of these values are redundant. Values 3 and 4 lead to the same amount of filtering and so do values 1 and 2. So, only 3 bS values are needed which simplifies the boundary strength computation. The modified bS computation for an edge E with pixels P and Q on either side is described as follows:

- If E is neither CU nor PU edge, bS = 0.
- Else if either P or Q or both are intra-coded, bS = 2
- Else if E is TU edge and either P or Q or both contain non-zero transform coefficients, bS = 1
- Else, both P and Q are inter-coded. Read two motion vectors ($\mathbf{mvP0}, \mathbf{mvP1}$) and corresponding reference frames (refP0, refP1) for P if P uses bi-prediction. If P uses uni-prediction, set one motion vector to 0 and its reference frame as invalid. Similarly, read $\mathbf{mvQ0}, \mathbf{mvQ1}$, refQ0 and refQ1 for block Q. Define absolute difference of two motion vectors $\mathbf{mv1} = (mv1_x, mv1_y)$ and $\mathbf{mv2} = (mv2_x, mv2_y)$ to be greater than 4 as

$$|\mathbf{mv1} - \mathbf{mv2}| > 4 = (|mv1_x - mv2_x| > 4) \;\|\; (|mv1_y - mv2_y| > 4)$$

– If all four refP0, refP1, refQ0 and refQ1 are equal,

$$\begin{aligned} \text{bS} = (&|\mathbf{mvP0} - \mathbf{mvQ0}| > 4 \parallel |\mathbf{mvP1} - \mathbf{mvQ1}| > 4)\,\&\& \\ &(|\mathbf{mvP1} - \mathbf{mvQ0}| > 4 \parallel |\mathbf{mvP0} - \mathbf{mvQ1}| > 4) \end{aligned}$$

– Else if refP0 = refQ0 and refP1 = refQ1,

$$\text{bS} = |\mathbf{mvP0} - \mathbf{mvQ0}| > 4 \parallel |\mathbf{mvP1} - \mathbf{mvQ1}| > 4$$

– Else if refP1 = refQ0 and refP0 = refQ1,

$$\text{bS} = |\mathbf{mvP1} - \mathbf{mvQ0}| > 4 \parallel |\mathbf{mvP0} - \mathbf{mvQ1}| > 4$$

– Else, bS = 1

Since all the parameters required to compute bS of an edge change at most on a 4×4 pixel block basis, the reference software computes the bS of an 8-pixel edge as the maximum of the two 4-pixel boundary strengths. We now describe the design for the boundary strength computation block.

This processing block computes bS for the top and left edges for 8×8 pixel blocks, henceforth called block8. The TU and PU information is available on a 4×4 block (block4) basis. The left-most and top-most edges in an LPU need TU and PU info for blocks outside the LPU. For blocks to the left of the LPU, this info is stored in a 16-entry register file. For blocks to the top of the LPU, the PU info is in the PU info SRAM itself, while the TU info is taken from the top-row buffer.

Computing the boundary strength of any 4-pixel edge requires knowing if the edge is a PU and/or TU edge. The TU edge bits in TU info are written by the transform unit to signal a TU edge. For PU edges however, the PU info SRAM only signals the CU size and PU partition type. So, the location of the CU in the LPU is first determined by rounding-off the location of the current block4 in the LPU to the nearest lower multiple of CU size. For example, if the current block4 starts at (44,

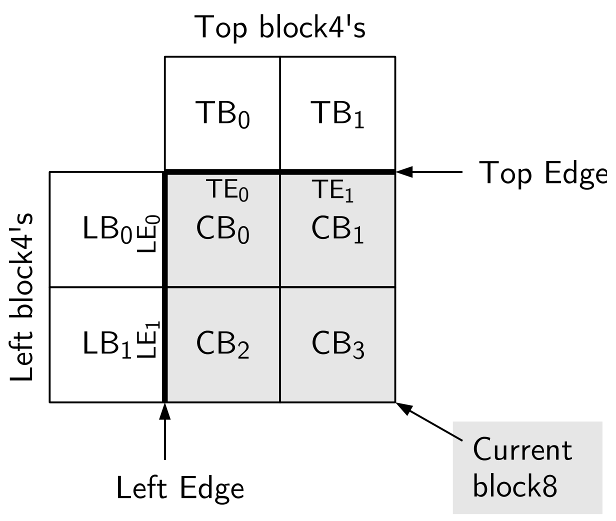


Figure 3-6: Block4 and edge indices in a block8

12) in the LPU and is part of CU_{16}, the CU starts at (32, 0). This determines the relative location of the current block in its CU. From this and the PU partition type, the PU edge is determined.

A 6-state FSM is used to compute the four 4-pixel boundary strengths - 2 each for the top and left edges. The actions in each state are described below with respect to block and edge indices shown in Figure 3-6.

1. $State = 0$:
   - Read LB$_1$ TU and PU info from register file.
   - Read CB$_2$ TU and PU info from Info SRAMs.
   - Compute and save bS for LE$_1$.
   - If current block is at bottom of LPU, save CB$_2$ TU info to top-row buffer.
   - Increment *State*.
2. $State = 1$:
   - Read LB$_0$ TU and PU info from register file.
   - Read CB$_0$ TU and PU info from Info SRAMs.
   - Compute bS for LE$_0$.

- Write out bS for left edge as maximum of bS's of LE_0 and LE_1.
- Save CB_0 TU and PU info to temporary register.
- Increment *State*.

3. *State* = 2:

    - Read TB_0 TU info from top-row buffer if edge is at top of LPU. Read from TU Info SRAM otherwise. Read TB_0 PU info from PU Info SRAM.
    - Read CB_0 TU and PU info from temporary register.
    - Compute and save bS for TE_0.
    - Increment *State*.

4. *State* = 3:

    - Read TB_1 TU info from top-row buffer if edge is at top of LPU. Read from TU Info SRAM otherwise. Read TB_1 PU info from PU Info SRAM.
    - Save TB_1 TU and PU info to temporary register.
    - Increment *State*.

5. *State* = 4:

    - Read CB_1 TU and PU info from Info SRAMs.
    - Read TB_1 TU and PU info from temporary register.
    - Compute bS for TE_1.
    - Write out bS for top edge as maximum of bS's of TE_0 and TE_1.
    - Write CB_1 TU and PU info to register file.
    - Increment *State*.

6. *State* = 5:

    - Read CB_3 TU and PU info from Info SRAMs.
    - Write CB_3 TU and PU info to register file.

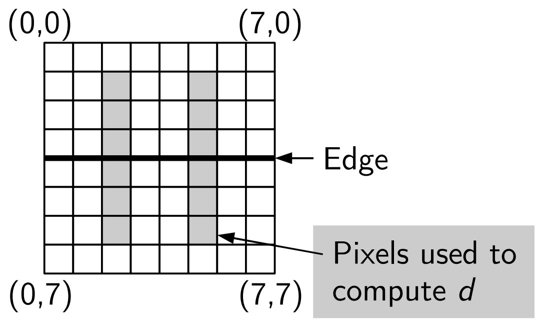


Figure 3-7: Pixels for edge parameters

- If current block is at bottom of LPU, save CB_3 TU info to top-row buffer.
- Reset *State* to 0.

3.2.6 Edge Params

The Edge Params block determines if the top and left edges of a block8 contain a blocking artifact and thus decides if the edges are to be filtered or not. It reads pixels on each side of the edge and computes an absolute second derivative d. From bS and QP, it computes two parameters β and tC. Filtering is performed only if bS > 0 and $d < \beta$. tC is used for per-pixel decision of strong and weak filtering.

Figure 3-7 shows the pixels used to compute the edge parameter d.

$$\begin{aligned} dP &= |p_{2,1} - 2p_{2,2} + p_{2,3}| + |p_{5,1} - 2p_{5,2} + p_{5,3}| \\ dQ &= |p_{2,4} - 2p_{2,5} + p_{2,6}| + |p_{5,4} - 2p_{5,5} + p_{5,6}| \\ d &= dP + dQ \end{aligned}$$

β and tC are computed from look-up tables with QP as the index for luma filtering and scaled QP for chroma. If bS > 1, the index is incremented by 4.

3.2.7 Transpose RegFile

The transpose register file stores one block8 in one 8×8 luma pixel and two 4×4 chroma pixel transpose memories implemented using registers. For luma filtering, 8

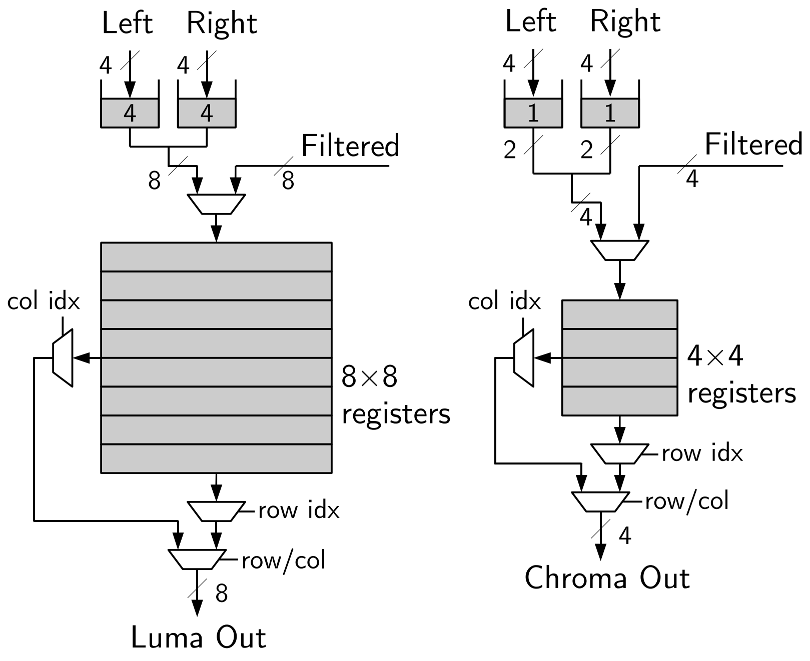


Figure 3-8: Deblocking Transpose RegFile. Both luma and chroma have 4-pixel wide inputs, Left and Right to match the widths of the last LPU buffer and pipeline buffer respectively.

rows of new pixels are written to the regfile from the pipeline buffer. The rows are read out for filtering the vertical edges and the partly filtered pixels are written back to the transpose memory. Once the vertical edge is filtered, the pixels are read out along the columns for horizontal edge filtering. 4 luma rows and 2 chroma rows from the next block8 can be prefetched and stored in input FIFO's as shown in Figure 3-8.

3.2.8 Last LPU buffer

The last LPU buffer contains the right-most column of block4's from the previous LPU. At the end of processing of the current LPU, it is updated with the unfiltered right-most column from the current LPU. For LCU_{64}, this buffer contains 4×64 luma pixels, $2 \times 2 \times 32$ chroma pixels and 16 edge parameters.

This buffer is also used as temporary storage within the LPU as shown in Figure 3-9. The LPU is processed on a block8 basis with the blocks processed in a vertical raster scan order. When processing the current block8, the last LPU buffer contains two block4's (LB_0 and LB_1) from the block8 to its left. At the end of the processing, it is updated with the two right block4's (CB_1 and CB_3) from the current block8.

3.2.9 Filter Process

The filter process operates on block8's in a vertical scan order. Because of dependencies in the vertical and horizontal edge filters, the processing is delayed by 4 luma rows and 4 luma columns. Thus, for the current block8 shown in Figure 3-9, the block4's labelled TLB, TB_0, LB_0 and CB_0 are processed by filtering the 4-pixel edges LE_{-1}, LE_0, TE_{-1} and TE_0.

At the start of processing the current block8, the block4's TLB and TB_0 have been prefetched into the transpose regfile while processing the previous block8. The block4's LB_0 and LB_1 are in the last LPU buffer and the current block8 (CB_0 to CB_3) is in the pipeline SRAM. Edges LE_{-1} and TE_{-1} belong to the block8's to the top and left of the current block8 and their edge parameters have been computed when those block8's were processed.

When processing the current block8, the luma pixels are read from the pipeline SRAM, the edge parameters of the top and left edge are computed and the edges are filtered in the order LE_{-1}, LE_0, TE_{-1} and TE_0. As shown in Figure 3-10 The filtered blocks TLB, TB_0, LB_0 and CB_0 are written back to the pipeline SRAM and unfiltered blocks CB_1 and CB_3 are saved to the last LPU buffer. Unfiltered block4's LB_1 and CB_2 are saved in the transpose memory for the next block8 in the LPU. If the present block8 is at the bottom of the LPU, these block4's are instead written to the top-row buffer. Furthermore, if the present block8 happens to be at the bottom of the frame, the block4's are written to the top-row buffer after filtering.

The process is similar for chroma pixels except that the processing is delayed by 2 rows and 2 columns. Luma and chroma filtering is performed in parallel to hide the latency of the processing which is necessary to avoid idle cycles on the pipeline SRAM. As explained subsection 3.2.1, the pipeline buffer must be always busy to achieve a 2-pixel per cycle throughput.

Pixel storage in the DRAM uses a mapping based on 8×4 luma blocks and packed 4×4 chroma blocks. Since luma processing is delayed by 4 rows and 4 columns, an

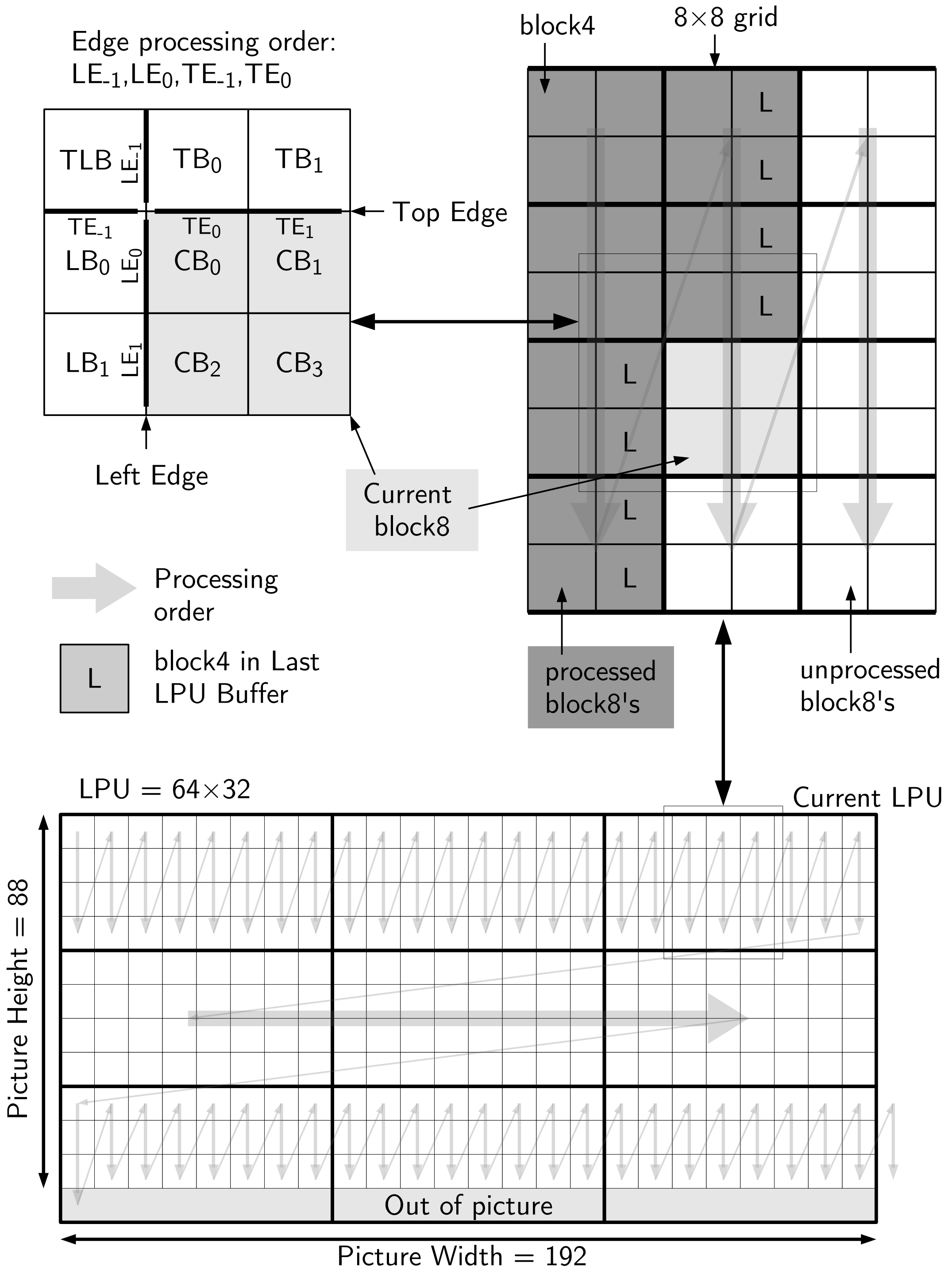


Figure 3-9: Deblocking processing order for a 192 × 88 picture coded with LCU_{32}. Note the anomalous order in the bottom LPU row and the last LPU required to filter the last LPU buffer.

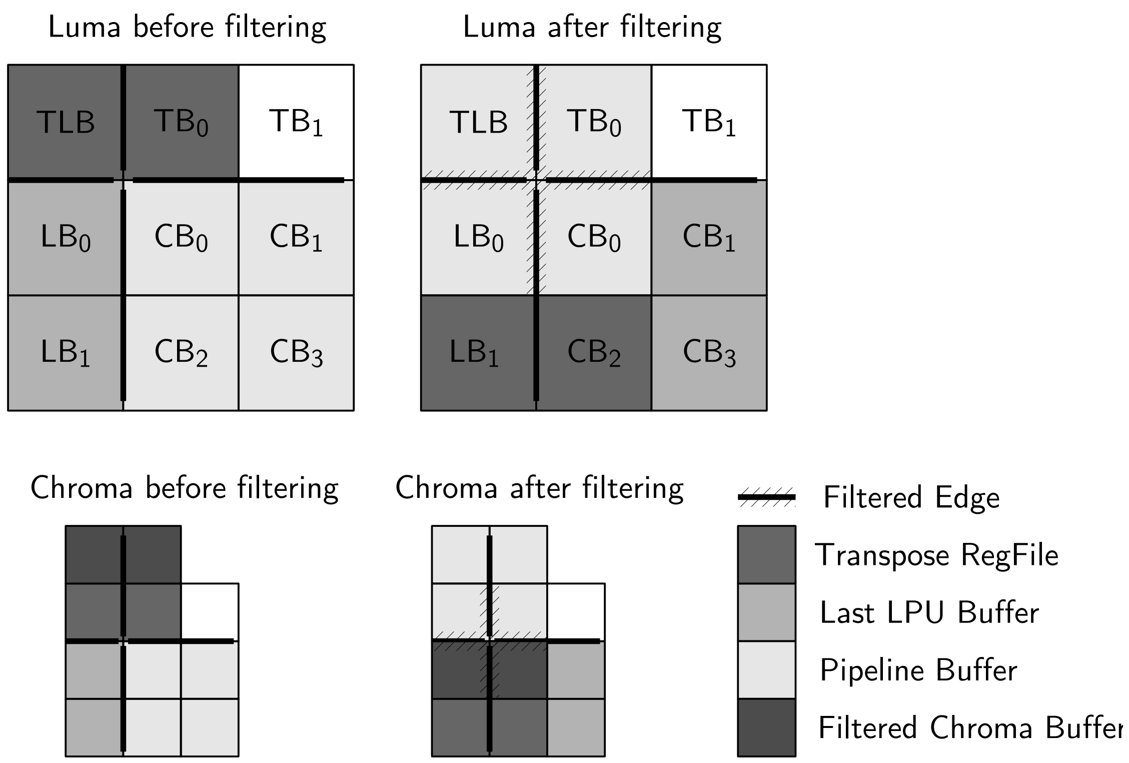


Figure 3-10: Pixel locations before and after filtering

extra 4 columns delay is added before writeback to DRAM using another last LPU buffer. Chroma requires an extra delay of 2 rows and 2 columns. The 2 column-delay can use the last LPU buffer, but the 2 row delay needs a $2 \times 2 \times 1920$ pixel buffer. This buffer is merged with the deblocking top-row buffer by delaying the writeback of 2 chroma rows with a filtered chroma buffer shown in Figure 3-10.

The order in which processing is performed is designed with the following concerns in mind:

1. Correctness: Apart from respecting dependencies such as filtering vertical edges before the horizontal edges, it is also necessary to avoid overwriting unread pixels or uncommitted pixels. This was found to necessitate reading all the pixels in the block8 before the writing back the filtered pixels. This is seen in timeline for the pipeline buffer in Figure 3-11.

2. Throughput: Maintaining a throughput of 2 pixels per cycle implies that each block8 containing 64 luma and 32 chroma pixels complete processing in 48

cycles. A key requirement is that the pipeline buffer be kept busy with read or write for all cycles.Figure 3-11 shows the simulated timeline for processing one block8 that meets the cycle budget.

3. Simplicity: This requires reducing the number of special cases. Only the left-most and right most block8's in an LPU need to access the last LPU buffer. But, by performing a vertical block8 raster scan within the LPU and using the last LPU buffer for all block8's, this special case was eliminated. The special case of top-most and bottom-most block8's which require access to the top-row buffer can similarly be eliminated. However, system requirements for sharing the top-row buffer with intra-prediction make it difficult to use it for all block8's in deblocking. Three more special cases remain.

   (a) The last LPU in the frame must put filtered pixels in the last LPU buffer as opposed to the usual. This is achieved by adding a dummy column to the last LPU.

   (b) Similarly, the bottom-most block8's in the frame must filter pixels before storing them in the top-row buffer. The timeline for this is shown in Figure B-1.

   (c) Also, when the picture height is not a multiple of LPU height, the last LPU row has smaller LPU's as pictured in Figure 3-9. A few dummy block8's must be added to process pixels in the last LPU buffer from the previous LPU row.

   Appendix B explains the special cases of top-most and bottom-most block8 in LPU and bottom-most block8 in frame in more detail. These require modifications to the normal block8 timeline of Figure 3-11 which are also shown there.

FIFO-based pixel and information passing is extensively used in the design to ensure correctness. Local state machines are used to generate SRAM requests, receive responses and handle special cases. A global state machine is used only to reset these

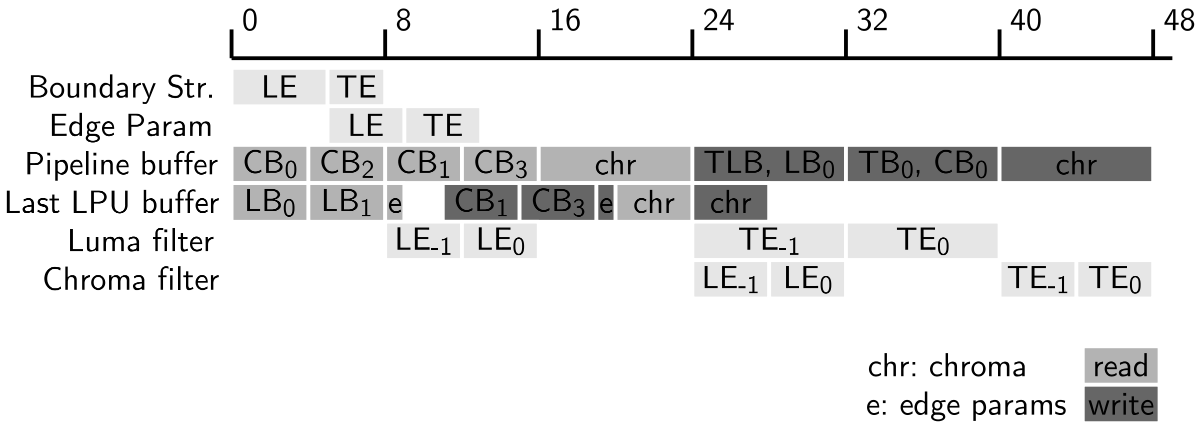


Figure 3-11: Processing timeline for block8 deblocking from RTL simulation

state machines. By avoiding a complex global state machine that manages everything in a rigid manner, a modular easy-to-modify design is achieved. A significant amount of scheduling is also handled by the Bluespec compiler guided by user-defined directives.

3.3 Results

The complete deblocking filter shown in Figure 3-4 requires 44 kgates of logic and 3.3 kbits of SRAM in TSMC 40nm technology. The breakdown of area is listed in Table 3.3. A significant portion of the area is taken by register arrays in transpose memory and boundary strength blocks. The state machine for scheduling the timeline and handling special cases explained in the previous section also take up a significant area. Post place-and-route power estimate is 2.4 mW including 1.3 mW of dynamic power, 0.7 mW clock network power and 0.4 mW leakage. The author thanks Chao-Tsung for the power numbers. The Virtex-6 FPGA implementation takes 6k LUT's, 2.7k registers and 2 BlockRAM's.

Module	Logic area (kgates)	SRAM size (kbits)
Transpose RegFile	11	0
Filter Process	6.1	0
Boundary Strength	12.6	0
Edge Params	3.0	0
Last LPU buffer	0	3.3
FIFOs	3.4	0
Scheduling and state	7.9	0
Total	44	3.3

Table 3.3: Area breakdown for deblocking filter

Chapter 4

Conclusion

Circuit implementations for inverse transform and deblocking filter blocks supporting all features of HEVC HM-4.0 have been described. The throughput of both the blocks support Quad-Full HD (3840×2160) decoding at 30fps when running at 200MHz. A 25MHz FPGA prototype was developed to demonstrate real-time HD (720p) decoding.

The two blocks present different design challenges. The inverse transform algorithm can be easily described as a matrix multiplication but the design is complicated by the large matrices ($32 \times 32 \times 16$-bit) involved. Techniques to share computations by identifying common factors in matrices are shown to be useful in generating an area-efficient design. The deblocking filter uses complicated algorithms to determine filtering parameters which result in complex data dependencies. However, the actual computation works on small data sets of 8 pixels which results in a low area implementation. The challenge then, is to respect the dependencies while maintaining the required throughput. These differences are best seen in Table 4.1 which compare lines-of-code (in Bluespec SystemVerilog, excluding test-benches) and area of the two blocks.

Module	Lines of code	Area
Inverse Transform	1826	104 kgates
Deblocking Filter	3639	44 kgates

Table 4.1: Comparison of Inverse Transform and Deblocking Filter designs

For both blocks, Bluespec SystemVerilog proved to be an invaluable tool as it abstracts away many wire-level details. For example, the designer does not have to write logic to check if a FIFO is full before pushing to it. As a result, debugging can be done in a software-like manner with *printf*'s. The powerful type system results in a more readable code as the design is described as operations on meaningful video coding parameters and vectors of pixels rather than on bit-vectors.

The Bluespec library has several FIFO modules with different behaviors. For example, the pipeline FIFO is a single-element FIFO which can be simultaneously pushed and popped only when it is full. The bypass FIFO is a single-element FIFO which can bypass the input to output so that push and pop can occur in the same cycle when the FIFO is empty. It is found that intelligently using these primitives in place of registers greatly simplifies the design. At the end of the deblocking filter design, a standard approach to designing complex digital architectures was found to evolve. The steps in this approach are described as follows:

1. Make a list of all interfaces to the module. Make sure the interfaces do not have implicit conditions such as latencies. For example, the multicycle read interface to SRAM should be split into two zero-latency sub-interfaces - request and response.

2. Devise a processing timeline that meets all dependencies and throughout requirements. At this stage, important state variables such as amount of scratchpad memory to store prefetched and intermediate data should be fixed. Also add pipelining for complex logic that is expected to generate long critical paths. If area is a concern, a rough estimate in terms of adders, multipliers, registers and muxes can be computed.

3. Describe the processing timeline using the least number of guarded atomic actions (rules). Determine rules that interact with each other through registers (rules for FSM's, rigid pipelines) and rules that interact through FIFOs (rules for elastic pipelines, request-response rules). If rules interact through both FIFO's

and register, they are effectively rigidly connected i.e. the register based interaction takes precedence.

4. Tweak the FIFO sizes and types to get the required throughout with the smallest size. Note that bypass FIFO's can increase the critical path due to the bypass path.

Key lessons learned in complex circuit architecture design are summarized below.

1. Using FIFO interfaces allows modules to remain weakly coupled. As a result changes in the internal design of one module do not affect the design of other modules. For example, SRAM's can be encapsulated to present a request-response FIFO interface to other modules. Now, if the SRAM is pipelined to reduce critical path, the design of other modules is unaffected. The circuit overhead to achieve this is minimal.

2. Elastic pipelines which use FIFO's as pipeline buffers are easier to design and verify than rigid pipelines controlled by state machines. It can be shown that the two methods are logically equivalent - instead of a globally stored state, the state is distributed among FIFO pointers. The advantage of a localised state is again weak coupling and ease of debug.

3. Little's law (number of elements in flight equals throughput times latency) is surprisingly useful in taking many design decisions such as FIFO depths and amount of prefetching.

Future work on coding tools for HEVC could explore several directions:

1. Scalable design: Scaling can be utilized on various levels to provide power savings.

    - Voltage scaling: A low-voltage design would enable the chip to efficiently decode smaller picture sizes and frame rates in mobile application. This design is especially challenging in scaled technology nodes (40nm and smaller) due to process variation and lack of low-voltage SRAM libraries.

- Gating: Parts of the processing blocks can be turned off for long periods of time depending on the video content. For example, when decoding a video with LCU_{16}, the 32-pt transform logic can be power gated for a period of time much longer than the break-even threshold [17] for power gating. Leakage is seen to be a significant power draw in the current design which makes power gating very attractive.

2. Circuit optimizations: Multiple-constant Multiplication (MCM) was used to optimize the transform matrix multiplications in this work. Distributed arithmetic is an alternate method that has been previously explored for MPEG-2 DCT[18]. It would be interesting to compare the areas of these two approaches for the large HEVC transforms. Other techniques such as multiple voltage and clock domains, multi-threshold CMOS can also be explored.

3. New coding tools: HEVC has introduced two new filters called Adaptive Loop Filter and Sample Adaptive Offset. Circuit implementations for these filters is definitely an exciting research direction.

Appendix A

Transform Optimizations

A.1 Partial butterfly structure

$$\text{Rename } g\_aiT32 = g, y_{DEC} = y, x_{DEC} = x \text{ in (2.1)} \tag{A.1}$$

For $m = 0 \ldots 31$,

$$\begin{aligned} y[m] &= \sum_{n=0}^{31} x[n] \cdot g[m][n] \\ &= \sum_{n=0}^{15} x[2n] \cdot g[m][2n] + \sum_{n=0}^{15} x[2n+1] \cdot g[m][2n+1] \end{aligned} \tag{A.2}$$

$$\begin{aligned} y[31-m] &= \sum_{n=0}^{31} x[n] \cdot g[31-m][n] \\ &= \sum_{n=0}^{15} x[2n] \cdot g[31-m][2n] + \sum_{n=0}^{15} x[2n+1] \cdot g[31-m][2n+1] \\ &= \sum_{n=0}^{15} x[2n] \cdot g[m][2n] - \sum_{n=0}^{15} x[2n+1] \cdot g[m][2n+1] \quad \text{from (2.11)} \end{aligned} \tag{A.3}$$

$$\text{Let } e[m] = \sum_{n=0}^{15} x[2n] \cdot g[m][2n] \tag{A.4}$$

$$\text{and } o[m] = \sum_{n=0}^{15} x[2n+1] \cdot g[m][2n+1] \tag{A.5}$$

From (A.2), (A.3), (A.4), (A.5),

$$y[m] = e[m] + o[m] \tag{A.6}$$

$$y[31-m] = e[m] - o[m] \tag{A.7}$$

$e[m]$ is simply the 16-pt IDCT of the even-numbered inputs and can be also be decomposed into even and odd parts $ee[m]$ and $eo[m]$ respectively. $ee[m]$ can be further decomposed. This recursive decomposition leads to a partial butterfly structure as shown in Figure 2-9.

$$\begin{aligned} e[m] &= ee[m] + eo[m] \\ e[15-m] &= ee[m] - eo[m] \\ ee[m] &= eee[m] + eeo[m] \\ ee[7-m] &= eee[m] - eeo[m] \\ eee[m] &= eeee[m] + eeeo[m] \\ eee[3-m] &= eeee[m] - eee0[m] \end{aligned}$$

A.2 Multiple Constant Multiplication for 32-pt IDCT

The 32-pt IDCT block contains a 16×16 matrix multiplication with the odd-indexed input coefficients. The matrix contains 15 unique coefficients: 90, 88, 85, 82, 78, 73, 67, 61, 54, 46, 38, 31, 22, 13, 4. The Spiral-generated MCM uses 13 adders to implement these 15 multiplications.

$$\begin{array}{llll}
a32 = a1 << 5 & a31 = a32 - a1 & a8 = a1 << 3 & a23 = a31 - a8 \\
a4 = a1 << 2 & a27 = a31 - a4 & a39 = a31 + a8 & a62 = a31 << 1 \\
a61 = a62 - a1 & a22 = a23 - a1 & a11 = a22 >> 1 & a26 = a27 - a1 \\
a13 = a26 >> 1 & a19 = a23 - a4 & a64 = a1 << 6 & a41 = a64 - a23 \\
a46 = a23 << 1 & a45 = a46 - a1 & a128 = a1 << 7 & a67 = a128 - a61 \\
a73 = a27 + a46 & a108 = a27 << 2 & a85 = a108 - a23 & a90 = a45 << 1 \\
a88 = a11 << 3 & a82 = a41 << 1 & a78 = a39 << 1 & a54 = a27 << 1 \\
a38 = a19 << 1 & & &
\end{array}$$

A.3 Multiple Constant Multiplication for 16-pt IDCT

The 32-pt IDCT block contains a 8×8 matrix multiplication with the even-odd-indexed $(2(2n+1))$ input coefficients. The matrix contains 8 unique coefficients: 90, 87, 80, 70, 57, 43, 25, 9 which are implemented with 8 adders as shown:

$$\begin{array}{llll}
b4 = b1 << 2 & b5 = b1 + b4 & b8 = b1 << 3 & b9 = b1 + b8 \\
b16 = b1 << 4 & b25 = b9 + b16 & b36 = b9 << 2 & b35 = b36 - b1 \\
b40 = b5 << 3 & b45 = b5 + b40 & b43 = b35 + b8 & b32 = b1 << 5 \\
b57 = b25 + b32 & b86 = b43 << 1 & b87 = b1 + b86 & b90 = b45 << 1 \\
b80 = b5 << 4 & b70 = b35 << 1 & &
\end{array}$$

Appendix B

Deblocking timeline

The simulated timelines for two special cases in deblocking are shown Figure B-1. When the block8 being processed is at the bottom of the LPU, the bottom-most block4's LB_1 and CB_2 must be written to the top-row buffer. The top block4's TLB and TB_0 for the next block8 must be fetched from the top-row buffer. In this case, to avoid overwriting in the local register storage, the writes to the top-row buffer are done before the reads. Overwriting in the top-row buffer is not a concern as the reads and writes go to different addresses. It is seen that reading and writing the chroma pixels to the top-row buffer is causes the 48 cycle budget to be exceeded. However, throughput is maintained by starting the processing of the next block8 at the end of 48 cycles itself.

For the block8's at the bottom of the picture, the block4's being written to the top-row buffer must be filtered. This can be seen in the extra processing of the L_1 edge in cycles 17-20 for luma filter and 48-51 for the chroma filter. As these blocks are actually processing more pixels, the cycle budget can be relaxed.

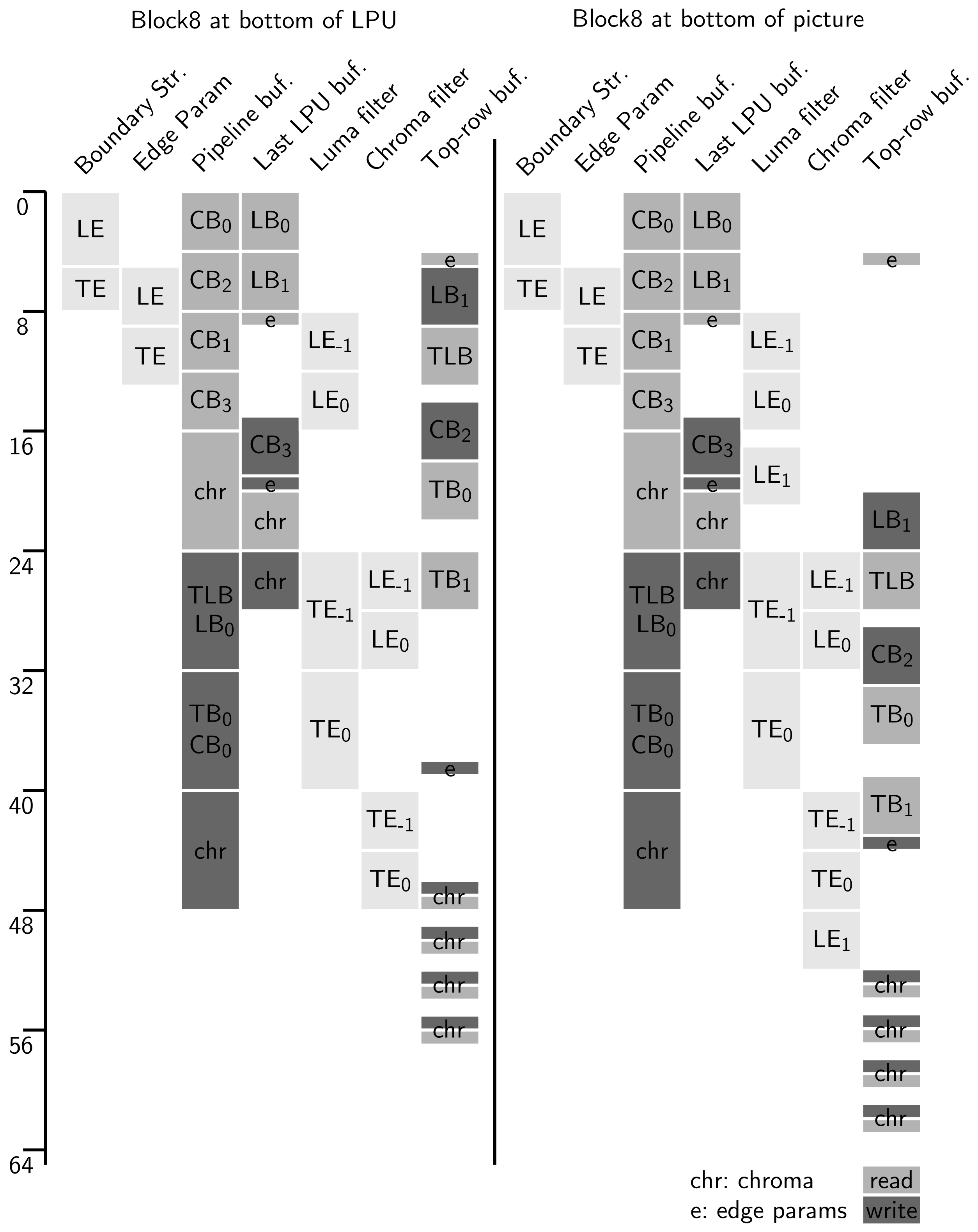


Figure B-1: Processing timeline for special cases in deblocking (from RTL simulation)

Bibliography

[1] T. Wiegand, G. Sullivan, G. Bjøntegaard, and A. Luthra, "Overview of the H.264/AVC Video Coding Standard," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 13, no. 7, pp. 560 –576, july 2003.

[2] B. Bross, W.-J. Han, O. Jens-Raimer, G. J. Sullivan, and W. Thomas, "WD4: Working Draft 4 of High-Efficiency Video Coding," Joint Collaborative Team on Video Coding JCTVC-F802, Tech. Rep., 2011.

[3] B. Li, G. J. Sullivan, and X. Jizheng, "Comparison of Compression Performance of HEVC Working Draft 4 with AVC High Profile," Joint Collaborative Team on Video Coding JCTVC-G399, Tech. Rep., 2011.

[4] M. Viitanen, J. Vanne, T. Hmlinen, and M. Gabbouj, "Complexity Analysis of Next-Generation HEVC Decoder," 2012, accepted as Lecture in International Symposium on Circuits and Systems.
Available: http://iscas2012.e-papers.org/.

[5] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck, "A Close Examination of Performance and Power Characteristics of 4G LTE Networks," *ACM Mobisys*, 2012.

[6] D. Zhou, J. Zhou, J. Zhu, P. Liu, and S. Goto, "A 2Gpixel/s H.264/AVC HP/MVC video decoder chip for Super Hi-vision and 3dtv/ftv applications," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2012 IEEE International*, feb. 2012, pp. 224 –226.

[7] "Bluespec Inc." www.bluespec.com.

[8] K. R. Rao and P. Yip, *Discrete Cosine Transform: Algorithms, Advantages and Applications.* Academic Press, Inc., 1990.

[9] A. Fuldseth, G. Bjøntegaard, M. Sadafale, and M. Budagavi, "Transform Design for HEVC with 16-bit Intermediate Data Representation," Joint Collaborative Team on Video Coding JCTVC-E243, Tech. Rep., 2011.

[10] A. Saxena and F. C. Fernandes, "CE7: Mode-dependent DCT/DST for Intra Prediction in Video Coding," Joint Collaborative Team on Video Coding JCTVC-D033, Tech. Rep., 2011.

[11] D. F. Finchelstein, "Low-power Techniques for Video Decoding," Thesis, Massachusetts Institute of Technology, 2009. [Online]. Available: http://dspace.mit.edu/handle/1721.1/52794

[12] K. K. Parhi, *VLSI Digital Signal Processing Systems: Design and Implementation*. Wiley, NY, 1999.

[13] M. Potkonjak, M. Srivastava, and A. Chandrakasan, "Efficient Substitution of Multiple Constant Multiplications by Shifts and Additions using Iterative Pairwise Matching," in *Design Automation, 1994. 31st Conference on*, june 1994, pp. 189 – 194.

[14] M. Chen, J.-Y. Jou, and H.-M. Lin, "An Efficient Algorithm for the Multiple Constant Multiplication Problem," in *VLSI Technology, Systems, and Applications, 1999. International Symposium on*, 1999, pp. 119 –122.

[15] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, "SPIRAL: Code Generation for DSP Transforms," *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, vol. 93, no. 2, pp. 232– 275, 2005.

[16] I. E. G. Richardson, *H.264 and MPEG-4 Video Compression*. Wiley, 2003.

[17] N. J. Ickes, "A Micropower DSP for Sensor Applications," Thesis, Massachusetts Institute of Technology, 2008.

[18] T. Xanthopoulos, "Low Power Data-Dependent Transform Video and Still Image Coding," Thesis, Massachusetts Institute of Technology, 1999.