EFFICIENT HEVC AND VVC VIDEO COMPRESSION HARDWARE DESIGNS

by
HOSSEIN MAHDAVI

Submitted to the Graduate School of Engineering and Natural Sciences
in partial fulfillment of
the requirements for the degree of Doctor of Philosophy

Sabancı University
December 2023

EFFICIENT HEVC AND VVC VIDEO COMPRESSION HARDWARE DESIGNS

DATE OF APPROVAL: 29/12/2023

ABSTRACT

EFFICIENT HEVC AND VVC VIDEO COMPRESSION HARDWARE DESIGNS

HOSSEIN MAHDAVI

Electronics Engineering, PhD Thesis, 2023

Thesis Supervisor: Assist. Prof. Dr. Murat Kaya Yapıcı

Thesis Co-Advisor: Prof. Dr. İlker Hamzaoğlu

Keywords: HEVC, VVC, Fractional Interpolation, 2D Transform, HLS, Affine Motion Estimation

Digital video usage has significantly increased in recent years. Since both the spatial and temporal resolutions of videos increased, new video compression standards such as High Efficiency Video Coding (HEVC) and Versatile Video Coding (VVC) are developed to achieve higher compression efficiency. VVC has higher compression efficiency than HEVC at the cost of higher computational complexity. Approximate computing can be used to reduce the computational complexity of error tolerant applications such as video compression. Dedicated hardware implementations are required for real time video compression.

In this thesis, we propose efficient exact HEVC and VVC hardware implementations. To reduce the computational complexity of HEVC and VVC algorithms, we propose approximate VVC fractional interpolation (FI) filters, HEVC two-dimensional (2D) discrete cosine transform (DCT) using approximate constant multiplication, and approximate VVC affine motion estimation (AME). We propose efficient approximate HEVC and VVC hardware implementations using approximate algorithms and approximate hardware.

In this thesis, approximate VVC FI filters are proposed. The proposed approximate filters reduce computational complexity of VVC FI at the expense of very small quality loss. Three VVC FI hardware implementing the proposed approximate VVC FI filters are also proposed. A novel VVC FI hardware using memory based constant multiplication is proposed. A new technique called decomposed coefficients is proposed for implementing HEVC FI (HFI) and VVC FI (VFI). The proposed technique decomposes the coefficients of FIR filters such that the number of additions is reduced. A new approximate constant multiplication technique is used to propose a HEVC 2D DCT hardware, in which common constant multiplications are calculated once so that the number of multiplications is reduced. The first FPGA implementations of VVC FI and HEVC fractional motion estimation (FME) using an HLS tool in the literature are proposed. Novel FPGA implementations of HEVC DCT algorithm using an HLS tool are proposed. An approximate VVC AME hardware is proposed using a proposed approximate absolute difference (AD) hardware, approximate adder tree, and sub-sampling.

ÖZET

VERİMLİ HEVC VE VVC VİDEO SIKIŞTIRMA DONANIM TASARIMLARI

HOSSEIN MAHDAVI

Elektronik Müh., Doktora Tezi, 2023

Tez Danışmanı: Yard. Doç. Dr. Murat Kaya Yapıcı

Tez Yardımcı Danışmanı: Prof. Dr. İlker Hamzaoğlu

Anahtar Kelimeler: HEVC, VVC, Kesirli İnterpolasyon, 2D Dönüşüm, HLS, Afin Hareket Tahmini

Son yıllarda dijital video kullanımı çok arttı. Videoların hem uzamsal hem de zamansal çözünürlükleri arttığı için, daha yüksek sıkıştırma verimliliği elde etmek için, Yüksek Verimli Video Kodlama (HEVC) ve Çok Yönlü Video Kodlama (VVC) gibi yeni video sıkıştırma standartları geliştirildi. VVC daha yüksek hesaplama karmaşıklığı pahasına HEVC'den daha yüksek sıkıştırma verimliliğine sahiptir. Yaklaşık hesaplama, video sıkıştırma gibi hataya dayanıklı uygulamaların hesaplama karmaşıklığını azaltmak için kullanılabilir. Gerçek zamanlı video sıkıştırma için özel donanım gerçeklemeleri gerekmektedir.

Bu tezde, verimli tam doğru HEVC ve VVC donanım gerçeklemeleri öneriyoruz. HEVC ve VVC algoritmalarının hesaplama karmaşıklığını azaltmak için, yaklaşık VVC kesirli interpolasyon (FI) filtreleri, yaklaşık sabit çarpma kullanan HEVC iki boyutlu (2D) ayrık kosinüs dönüşümü (DCT) ve yaklaşık VVC afin hareket tahmini (AME) öneriyoruz. Yaklaşık algoritmalar ve yaklaşık donanım kullanarak verimli yaklaşık HEVC ve VVC donanım gerçeklemeleri öneriyoruz.

Bu tezde yaklaşık VVC FI filtreleri önerildi. Önerilen yaklaşık filtreler, çok küçük bir kalite kaybı pahasına VVC FI'nın hesaplama karmaşıklığını azaltmaktadır. Önerilen

yaklaşık VVC FI filtrelerini gerçekleyen üç VVC FI donanımı da önerildi. Bellek tabanlı sabit çarpma kullanan yeni bir VVC FI donanımı önerildi. HEVC FI (HFI) ve VVC FI'nı (VFI) gerçeklemek için ayrıştırılmış katsayılar adlı yeni bir teknik önerildi. Önerilen teknik, FIR filtrelerinin katsayılarını toplama sayısını azaltacak şekilde ayrıştırmaktadır. Yeni bir yaklaşık sabit çarpma tekniği kullanılarak çarpma sayısını azaltacak şekilde ortak sabit çarpmaları bir defa hesaplayan HEVC 2D DCT donanımı önerildi. Literatürdeki bir HLS yazılımı kullanılarak VVC FI ve HEVC kesirli hareket tahmininin (FME) ilk FPGA gerçeklemeleri önerildi. HEVC DCT algoritmasının HLS yazılımı kullanılarak yeni FPGA gerçeklemeleri önerildi. Önerilen bir yaklaşık mutlak fark (AD) donanımı, yaklaşık toplayıcı ağacı ve alt örnekleme kullanılarak yaklaşık bir VVC AME donanımı önerildi.

ACKNOWLEDGEMENT

I am sincerely grateful to my advisor, Dr. Murat Kaya Yapıcı, for all his unconditional support. It was a great honor for me to work under his supervision.

I would like to thank my co-advisor, Dr. İlker Hamzaoğlu. I would appreciate all his skills, support, advice, and life lessons throughout my studies and research. I have been honored and privileged to work under his supervision and guidance.

I want to thank my thesis committee members Dr. Hüseyin Özkan, Dr. Onur Varol, Dr. H. Fatih Uğurdağ, and Dr. Hasan F. Ateş for their invaluable feedback.

My thanks also go to the previous members of "System-on-Chip Design and Test Lab", Berke Ayrancıoğlu, Waqar Ahmad, and Hasan Azgın for their friendship and support.

My acknowledgements also go to Sabanci University for supporting me with scholarship throughout my studies.

Last but not least, my deepest gratitude to my beloved wife, Arghavan, for her constant support, encouragement, and patience.

To my parents and siblings

To my beloved wife Arghavan

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

AD	Absolute Difference
AME	Affine Motion Estimation
ALF	Adaptive Loop Filter
AMT	Adaptive Multiple Transform
ASIC	Application Specific Integrated Circuits
BRAM	Block Ram
CABAC	Context Adaptive Binary Arithmetic Coding
CU	Coding Unit
DBF	Deblocking Filter
DSP	Digital Signal Processor
DCT	Discrete Cosine Transform
DST	Discrete Sine Transform
FI	Fractional Interpolation
FIR	Finite Impulse Response
FHD	Full High Definition
FPGA	Field Programmable Gate Array
HD	High Definition
HEVC	High Efficiency Video Coding
HLS	High-Level Synthesis
HM	HEVC Test Model
IDCT	Inverse Discrete Cosine Transform
IDST	Inverse Discrete Sine Transform
JEM	Joint Exploration Test Model
JCT-VC	Joint Collaborative Team on Video Coding
MV	Motion Vector
MCM	Multiple Constant Multiplication
PSNR	Peak Signal to Noise Ratio

PU	Prediction Unit
SAD	Sum of Absolute Differences
QFHD	Quad Full HD
QP	Quantization Parameter
SAO	Sample Adaptive Offset
TU	Transform Unit
VCD	Value Change Dump
VVC	Versatile Video Coding

CHAPTER I

INTRODUCTION

Uncompressed video sequences produce an enormous amount of data, and the widespread use of video has steadily increased. Furthermore, the production of video content has shifted away from exclusive professional studios to personal production, real-time video chat, remote home monitoring, and even always-on wearable cameras. Hence, video traffic is the main load on communication networks and data storage in spite of the significant developments in video compression standards [1]. Digital video content now comprises around 80% of all the internet traffic. Mobile internet video traffic is also growing dramatically [2].

Video compression standards exploit temporal and spatial redundancy for achieving compression. Intra-frame prediction exploits the spatial redundancy between adjacent blocks in a frame, whereas motion-compensated prediction exploits the extensive temporal redundancy between frames. In either case, the resultant prediction error, which is derived from the difference between the original block and its prediction, is transmitted using transform coding. Transform coding comprises decorrelating linear transform, scalar quantization of the transform coefficients and entropy coding.

ITU and ISO recently developed Versatile Video Coding (VVC) standard [3]-[8]. VVC has higher compression efficiency than High Efficiency Video Coding (HEVC) standard. However, it has higher computational complexity than HEVC [9]-[14]. Approximate computing can be used to reduce the computational complexity of error

tolerant applications such as video compression. Dedicated hardware implementations are required for real time video compression.

In this thesis, we propose efficient exact HEVC and VVC hardware implementations. To reduce the computational complexity of HEVC and VVC algorithms, we propose approximate VVC fractional interpolation (FI) filters, HEVC two-dimensional (2D) discrete cosine transform (DCT) using approximate constant multiplication, and approximate VVC affine motion estimation (AME). We propose efficient approximate HEVC and VVC hardware implementations using approximate algorithms and approximate hardware.

1.1 HEVC Video Compression Standard

High efficiency video coding (HEVC) achieves 50% more compression than H.264 at the cost of higher computational complexity [15]. Figure 1.1 shows the top-level block diagram of an HEVC encoder. An HEVC encoder has a forward path and a reconstruction path. The forward path is utilized to encode a video frame using intra and inter predictions and to generate the bit stream after the transform and quantization process.

In the encoding process, the frame is divided into coding units (CU), which can vary in size from 8×8, 16×16, 32×32, to 64×64 pixels. Every CU is encoded in either intra or inter mode determined by the mode decision. Both intra and inter prediction methods employ prediction unit (PU) partitioning within the CUs. PU sizes vary from 4×4 to 64×64. Mode decision decides whether a PU is encoded in intra or inter mode according to video quality and bit-rate. After mode decision decides the prediction mode, the predicted block is subtracted from the original block, resulting in the residual block. Subsequently, the residual block is transformed by DCT / discrete sine transform (DST) and quantized. Transform unit (TU) sizes vary from 4×4 to 32×32. Lastly, the encoded bitstream is generated by entropy coder.

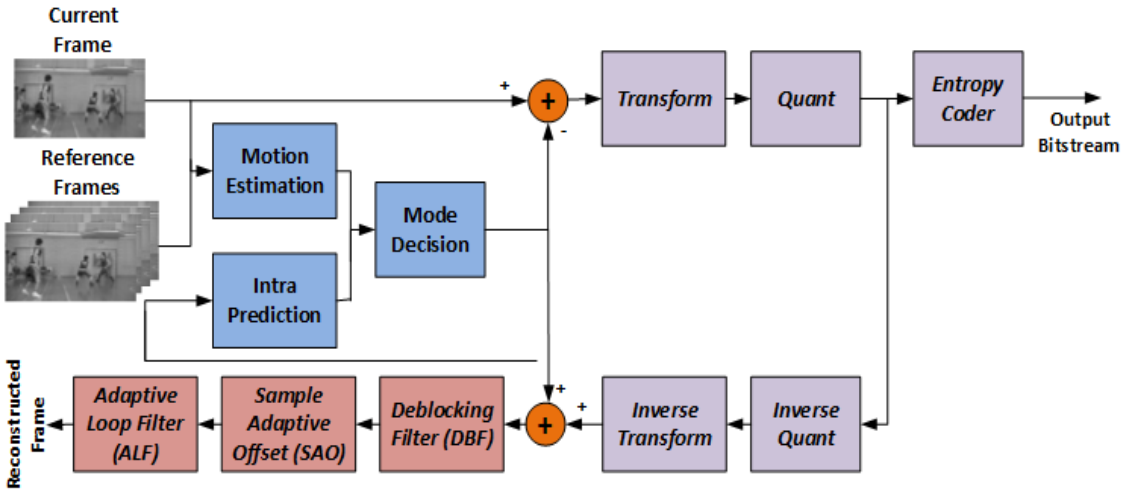

Figure 1.1 HEVC Encoder Block Diagram

The reconstruction path in the encoder guarantees that both encoder and decoder utilize identical reference frames for intra and inter prediction since a decoder never receives original images. It starts with inverse quantization and inverse transform. The coefficients of quantized transform are inverse quantized and inverse transformed to form the reconstructed residual block. Because quantization is a lossy operation, the reconstructed residual block is not identical to the original residual block. The reconstructed residual block is added to the predicted block to generate the reconstructed frame. Then, deblocking filter (DBF) decreases the effects of blocking artifacts in the reconstructed frame.

Intra prediction predicts the pixels of a block from the pixels in neighboring blocks that have already been encoded and reconstructed. In HEVC, intra PU sizes vary from 4×4 to 64×64 for the luminance component. Intra prediction for a PU can have up to 35 modes [16].

Inter prediction predicts the pixels of a block in the current frame from the pixels in the previous frames blocks that have already been encoded and reconstructed. In HEVC, inter PU sizes vary from 8×4/4×8 to 64×64. HEVC inter prediction utilizes integer pixel motion estimation and fractional (sub-pixel) motion estimation. First, integer pixel motion estimation is done for an inter PU. Then, fractional (sub-pixel) motion estimation is done for the same inter PU. In HEVC, two different 7-tap and one 8-tap finite impulse response (FIR) filters are used for fractional interpolations.

In HEVC, integer based DCT is used. TUs are square-shaped with sizes from 4×4 to 32×32. HEVC also uses DST for the 4×4 intra prediction.

1.2 VVC Video Compression Standard

VVC standard has better coding efficiency than HEVC at the cost of much higher computational complexity. VVC has a similar top-level block diagram to HEVC. In VVC, the main blocks of HEVC are improved to achieve better compression at the cost of higher computational complexity.

VVC intra prediction is similar to HEVC intra prediction. In VVC, angular intra prediction has 65 modes. Moreover, 4-tap cubic and 4-tap gaussian filters are used in angular intra prediction modes of VVC.

VVC inter prediction uses the same two-stage search as HEVC. VVC utilizes seven 8-tap and eight 7-tap FIR filters for fractional interpolation.

In VVC, integer based DCT is used similar to HEVC. However, VVC utilizes an adaptive multiple transform (AMT) method. VVC TU sizes vary from 4×4 to 64×64 [17].

Entropy coder uses context adaptive binary arithmetic coding (CABAC) similar to HEVC with some improvements.

1.3 Thesis Contributions

Fractional interpolation (FI) is a computationally complex algorithm used in the HEVC and VVC video encoder and decoder. FI accounts for 25% and 50% of the HEVC encoder and decoder complexity, respectively. In this thesis, approximate VVC FI filters are proposed [18]. The proposed approximate VVC FI filters reduce computational complexity of VVC FI at the expense of very small quality loss. Three VVC FI hardware implementing the proposed approximate VVC FI filters are also proposed. The proposed approximate VVC FI hardware have higher speed, smaller area, and up to 51% lower power consumption than the exact VVC FI hardware. Therefore, they can be used in consumer electronics devices requiring high speed, small area, low power consuming VVC encoder hardware.

A novel VVC FI hardware using memory based constant multiplication is proposed [19]. The proposed hardware stores pre-computed products of an input pixel with multiple constant coefficients in memory. It implements multiplications with constant coefficients using these pre-computed products. Several optimizations are

proposed to reduce memory size. The proposed VVC FI hardware can process 49 full HD (1920×1080) video frames per second (fps). It has up to 9.4% less power consumption than VVC FI hardware in the literature.

In this thesis, a novel technique is proposed for implementing HEVC FI [20]. It is also used for VVC FI. The proposed technique decomposes the coefficients of FIR filters such that the number of additions is reduced. In this thesis, an HEVC FI hardware, a VVC FI hardware, and two approximate VVC FI hardware are designed and implemented using the proposed technique. The proposed HEVC FI hardware has higher performance, less area, and less power consumption than the best HEVC FI hardware in the literature. The proposed VVC FI hardware has higher performance, less area, and less power consumption than the best VVC FI hardware in the literature. The proposed two approximate VVC FI hardware have the same performance, less area, and less power consumption than the best approximate VVC FI hardware in the literature.

In this thesis, an approximate constant multiplication technique, which has been proposed in [21], is used to propose an HEVC 2D DCT for all transform unit (TU) sizes. We use the approximate constant multiplication for multiplications with only the DCT coefficients that do not cause high average percentage error. There are some common constant multiplications that are calculated once so that the number of multiplications is reduced. The proposed approximate HEVC DCT hardware, in the worst case, can process 76 QFHD (3840×2160) frames per second.

High-level synthesis (HLS) is used to increase productivity. In this thesis, we propose the first HLS implementations of VVC FI algorithm in the literature [22]. Three different C++ codes are developed based on the software implementation of VVC FI in the VVC test model software encoder (VTM) [23]. All these C++ codes are synthesized to Verilog RTL using Xilinx Vivado HLS tool. The Verilog RTL codes are implemented to Xilinx Virtex-7 FPGA using Xilinx Vivado tool. The best proposed VVC FI HLS implementation can process 62 full HD (1920×1080) video frames per second. We propose the first HEVC fractional motion estimation (FME) HLS implementations by developing two different C++ codes based on the HEVC reference software encoder (HM) [24]. We propose novel HEVC 2D DCT HLS implementations by developing two different C++ codes based on the HEVC reference software encoder (HM) [24].

VVC uses affine motion estimation (AME) which considers rotation, zooming, and shearing motions of blocks during block matching motion estimation (ME). AME

achieves higher video compression than translational ME at the cost of much more computational complexity. In this thesis, an approximate VVC AME hardware is proposed using a proposed approximate absolute difference (AD) hardware, approximate adder tree, and sub-sampling. The proposed approximate AD hardware reduces the bit length of each AD value from 8 to 5. A new approximate adder tree is proposed to decrease the bit length of the adders. To further reduce the computational complexity of VVC AME, sub-sampling is used.

1.4 Thesis Organization

The rest of the thesis is organized as follows.

Chapter II explains VVC FI. It presents the proposed approximate VVC FI filters. MCMF1, BF2 and MCMF2 hardware designs are explained, and their implementation results are presented. The proposed VVC FI hardware using memory based constant multiplication is described and its implementation results are given.

Chapter III explains the proposed decomposed coefficients technique for implementing FI hardware. It describes the proposed exact and approximate HEVC and VVC FI hardware using the proposed technique.

Chapter IV explains the approximate constant multiplier proposed in [21]. It explains the proposed approximate HEVC DCT using the approximate constant multiplier and gives the experimental results.

Chapter V explains the first FPGA implementations of VVC FI algorithm using an HLS tool in the literature. It describes the first FPGA implementations of HEVC FME algorithm using an HLS tool in the literature. It describes novel FPGA implementations of HEVC 2D DCT algorithm using an HLS tool.

Chapter VI explains the VVC AME. It explains the proposed approximate absolute difference hardware, approximate adder tree, and sub-sampling that are used in the proposed approximate VVC AME hardware. It gives the implementation results.

Chapter VII presents conclusions and future work.

CHAPTER II

APPROXIMATE AND EXACT VERSATILE VIDEO CODING FRACTIONAL INTERPOLATION FILTERS AND THEIR HARDWARE IMPLEMENTATIONS

VVC fractional interpolation (FI) has higher computational complexity than HEVC FI. HEVC FI uses 3 different (one 8-tap, two 7-tap) finite impulse response (FIR) filters. HEVC FI interpolates 15 fractional pixels (3 horizontal half pixels, 3 vertical half pixels, 9 quarter pixels) for every integer pixel. Each FIR filter is used to interpolate 5 fractional pixels. VVC FI uses 15 different (seven 8-tap, eight 7-tap) FIR filters. VVC FI interpolates 255 fractional pixels (15 horizontal half pixels, 15 vertical half pixels, 225 quarter pixels) for every integer pixel. Each FIR filter is used to interpolate 17 fractional pixels.

An approximate VVC FI filter (F1) and its baseline hardware (BF1) are proposed in [25]. In this thesis, we propose a more efficient hardware for implementing F1 (MCMF1) using common offset values and Hcub multiplierless constant multiplication (MCM) technique [26]. In this thesis, we propose another approximate VVC FI filter (F2), its baseline hardware (BF2), and a more efficient hardware for implementing F2 (MCMF2) using common offset values and Hcub MCM technique [26].

F1 and F2 approximate VVC FI filters reduce computational complexity of VVC FI at the expense of very small quality loss. F2 filter causes slightly more quality loss than F1 filter.

MCMF1, BF2 and MCMF2 approximate VVC FI hardware are implemented using Verilog HDL. Verilog RTL codes are implemented to a 28 nm FPGA. The FPGA implementations are verified on an FPGA board.

F2 approximate VVC FI filters are proposed and used for fractional motion estimation. Fractional motion estimation is done in two steps. First, fractional interpolation is performed. Then, search operation is performed using the interpolated fractional pixels. The proposed approximate VVC FI filters are used in the fractional interpolation step.

There is no approximate VVC FI filter in the literature for fractional motion estimation. VVC standard uses an adaptive motion vector resolution (AMVR) scheme for coding motion vector differences with different precision. Alternative half-sample interpolation filters for the AMVR scheme are proposed in [27]. These filters are not proposed and used for fractional motion estimation.

Exact VVC FI hardware are proposed in [28] and [29]. BF1, MCMF1, BF2 and MCMF2 approximate VVC FI hardware are proposed for fractional motion estimation. They calculate 255 fractional pixels (FPs) for every integer pixel. Since exact VVC FI hardware proposed in [28] calculates 1 fractional pixel for every integer pixel, it can only be used for fractional motion compensation. Since exact VVC FI hardware proposed in [29] calculates 255 FPs for every integer pixel, it can be used for fractional motion estimation.

The proposed approximate VVC FI hardware have higher speed, smaller area, and up to 51% lower power consumption than the exact VVC FI hardware proposed in [29]. BF1 and MCMF1 hardware can process 47 full HD (1920×1080) video frames per second. BF2 and MCMF2 hardware can process 49 full HD (1920×1080) video frames per second. BF2 and MCMF2 hardware have higher speed, smaller area and lower power consumption than BF1 and MCMF1 hardware. However, they have slightly worse rate-distortion performance than BF1 and MCMF1 hardware.

The proposed approximate VVC FI hardware can be used in a VVC encoder hardware to perform fractional interpolation. VVC encoder hardware is expected to be used in consumer electronics devices requiring real time video encoding with high compression efficiency. VVC encoder hardware can be integrated into a System-on-Chip used in consumer electronics devices as a hardware accelerator connected to the on-chip bus.

Several HEVC FI hardware are proposed in the literature [30]-[33]. Approximate HEVC FI filters are proposed in [34]. BF1, MCMF1, BF2 and MCMF2 hardware are compared with them.

In this thesis, we also propose a new VVC FI hardware using memory based constant multiplication for all prediction unit (PU) sizes. The proposed hardware stores pre-computed products of an input pixel with multiple constant coefficients in memory. Multiplications with constant coefficients are implemented using these pre-computed products. Several optimizations are proposed to decrease memory size.

2.1 VVC Fractional Interpolation

VVC FI uses seven 8-tap and eight 7-tap FIR filters. Coefficients of the first nine FIR filters are shown in Table 2.1. P_{-3},…,P_4 represent input pixels, and their sub-indices represent indices of coefficients. F_8 FIR filter equation is shown in (2.1).

Table 2.1 Coefficients of VVC FI FIR Filters

FIR Filters	P_{-3}	P_{-2}	P_{-1}	P_0	P_1	P_2	P_3	P_4
F_1	0	1	-3	63	4	-2	1	0
F_2	-1	2	-5	62	8	-3	1	0
F_3	-1	3	-8	60	13	-4	1	0
F_4	-1	4	-10	58	17	-5	1	0
F_5	-1	4	-11	52	26	-8	3	-1
F_6	-1	3	-9	47	31	-10	4	-1
F_7	-1	4	-11	45	34	-10	4	-1
F_8	-1	4	-11	40	40	-11	4	-1
F_9	-1	4	-10	34	45	-11	4	-1

$$F_8 = \begin{pmatrix} -P_{-3} + 4 \times P_{-2} - 11 \times P_{-1} + 40 \times P_0 \\ + 40 \times P_1 - 11 \times P_2 + 4 \times P_3 - P_4 \end{pmatrix} \gg 6 \quad (2.1)$$

As can be seen in the table, the coefficients of F_9 and F_7 FIR filters are symmetric. Similarly, the coefficients of F_{10} and F_6, F_{11} and F_5, F_{12} and F_4, F_{13} and F_3, F_{14} and F_2, F_{15} and F_1 are symmetric. Therefore, the coefficients of F_{10}, F_{11}, F_{12}, F_{13}, F_{14} and F_{15} are not shown in the table.

Integer pixels and FPs are shown in Figure 2.1. There are 15 horizontal half pixels and 15 vertical half pixels between two adjacent horizontal and vertical integer pixels, respectively. They are interpolated from closest integer pixels using fifteen FIR

filters F_1, F_2, …, F_{14}, F_{15}. There are 225 quarter pixels between adjacent horizontal and vertical half pixels. They are interpolated from the closest horizontal half pixels using fifteen FIR filters F_1, F_2, …, F_{14}, F_{15}.

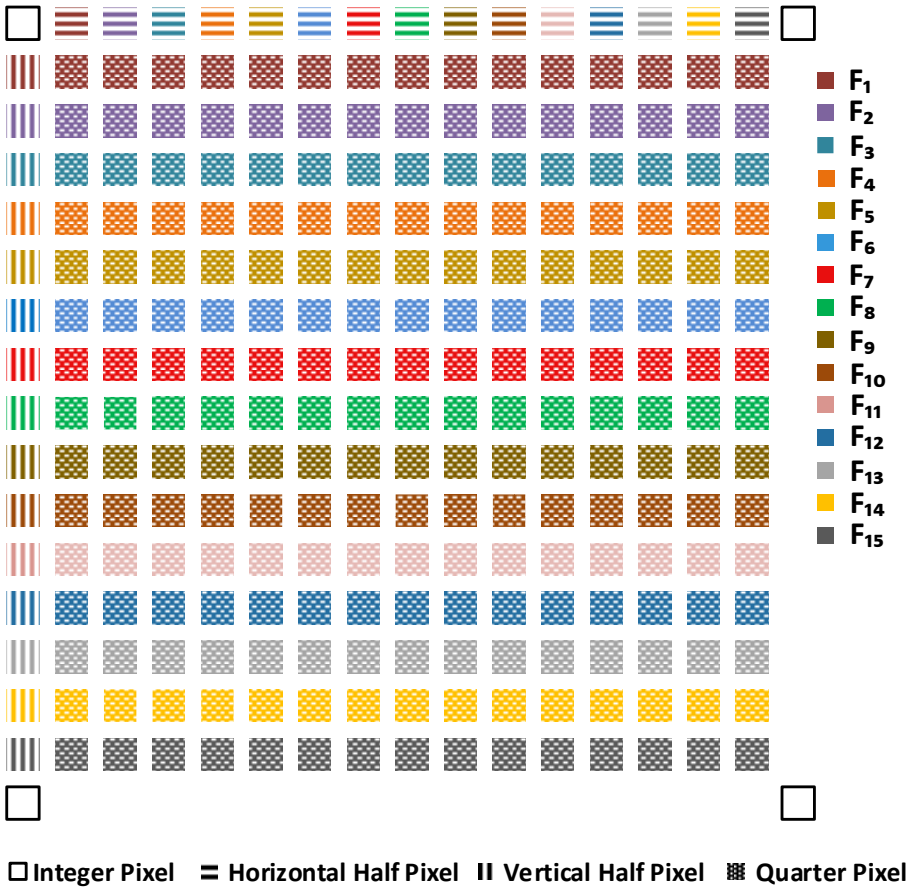

Figure 2.1 Integer pixels and fractional pixels in VVC standard

2.2 Proposed Approximate VVC Fractional Interpolation Filters

The approximate VVC FI F1 FIR filter is proposed in [25]. The coefficients of these fourteen 3-tap FIR filters and one 4-tap FIR filter are shown in Table 2.2.

Small coefficients of VVC FI FIR filters have less effect on their results. The values of adjacent pixels are similar because of spatial correlation. Therefore, coefficients of F1 FIR filters are determined by assuming that pixels multiplied with small coefficients are similar. For example, for VVC FI FIR filter F_2 shown in Table 2.1, if the values of pixels (P_{-3}, P_{-2}, P_{-1}) multiplied with first three coefficients (-1, 2, -5)

are the same, the result of F_2 filter can be calculated by multiplying one pixel with -4 (-1+2-5 = -4).

In this thesis, we propose another approximate VVC FI filter (F2). The coefficients of F2 FIR filters (fourteen 3-tap and one 4-tap) are shown in Table 2.3. These coefficients are determined by replacing most of the coefficients of F1 FIR filters with closest 2^n values. Therefore, multiplications with most of the coefficients of F2 FIR filters are performed using only shift operations. This reduces the number of adders required to implement the multiplications.

Table 2.2 Coefficients of Proposed Approximate F1 FIR Filters

F1 FIR Filters	P_{-1}	P_0	P_1	P_2
$F1F_1$	-2	63	3	0
$F1F_2$	-4	62	6	0
$F1F_3$	-6	60	10	0
$F1F_4$	-7	58	13	0
$F1F_5$	-8	52	20	0
$F1F_6$	-7	47	24	0
$F1F_7$	-8	45	27	0
$F1F_8$	-8	40	40	-8
$F1F_9$	0	27	45	-8
$F1F_{10}$	0	24	47	-7
$F1F_{11}$	0	20	52	-8
$F1F_{12}$	0	13	58	-7
$F1F_{13}$	0	10	60	-6
$F1F_{14}$	0	6	62	-4
$F1F_{15}$	0	3	63	-2

Table 2.3 Coefficients of Proposed Approximate F2 FIR Filters

F2 FIR Filters	P_{-1}	P_0	P_1	P_2
$F2F_1$	-2	64	2	0
$F2F_2$	-4	64	4	0
$F2F_3$	-8	64	8	0
$F2F_4$	-8	56	16	0
$F2F_5$	-8	56	16	0
$F2F_6$	-8	40	32	0
$F2F_7$	-8	40	32	0
$F2F_8$	-8	40	40	-8
$F2F_9$	0	32	40	-8
$F2F_{10}$	0	32	40	-8
$F2F_{11}$	0	16	56	-8
$F2F_{12}$	0	16	56	-8
$F2F_{13}$	0	8	64	-8
$F2F_{14}$	0	4	64	-4
$F2F_{15}$	0	2	64	-2

VVC FI filter used for fractional motion estimation in VVC test model software encoder [23] is replaced with the proposed approximate VVC FI filters F1 and F2. First ten frames of several test videos [35] are coded with low delay P test configuration using VVC test model software encoder with VVC FI FIR filters, F1 FIR filters and F2 FIR filters.

BD-Rate and BD-PSNR results are shown in Table 2.4. F1 and F2 filters reduce computational complexity of VVC FI at the expense of very small PSNR loss and bit rate increase. F1 filter has slightly better rate-distortion performance than F2 filter.

Table 2.4 BD-Rate and BD-PSNR Results

Video		F1 Filter		F2 Filter	
		BD-Rate (%)	BD-PSNR (dB)	BD-Rate (%)	BD-PSNR (dB)
2560×1600	People on Street	1.59	-0.059	1.99	-0.074
	Traffic	0.42	-0.011	0.54	-0.024
1920×1080	Tennis	0.52	-0.015	0.44	-0.013
	Kimono	0.03	-0.001	0.20	-0.006
	Basketball Drive	2.49	-0.047	2.80	-0.053
	Park Scene	2.26	-0.073	2.61	-0.084
1280×720	Vidyo1	0.85	-0.033	1.03	-0.039
	Vidyo4	0.23	-0.006	0.65	-0.020
	Kristen and Sara	1.62	-0.055	2.36	-0.081
	Four People	0.69	-0.027	0.87	-0.036
Average		1.07	-0.032	1.35	-0.043

2.3 Proposed Approximate VVC Fractional Interpolation Hardware

We proposed an approximate baseline VVC FI hardware for implementing F1 filter (BF1) in [25]. In this thesis, we propose a more efficient hardware for implementing F1 filter (MCMF1) using common offset values and Hcub MCM technique. In this thesis, we propose an approximate baseline VVC FI hardware for implementing F2 filter (BF2). We also propose more efficient hardware for implementing F2 filter (MCMF2) using common offset values and Hcub MCM technique.

Proposed BF1 hardware is shown in Figure 2.2. Interconnects boxes represent interconnects in BF1 hardware. They are used to simplify drawing interconnects in the figure. BF1 hardware implements multiplications with filter coefficients using adders and shifters. It has 8 filter datapaths hardware. As shown in Figure 2.3, 15 F1 FIR filters are implemented using 15 parallel datapaths in one filter datapaths hardware. C, D, E, F

inputs represent the pixels given to each filter datapaths. For example, P_{-1}, P_0, P_1, P_2 are given to Filter Datapaths 4 as inputs C, D, E, F, respectively. BF1 hardware interpolates 8×15 FPs in a clock cycle using 15 integer pixels or 15 horizontal half pixels.

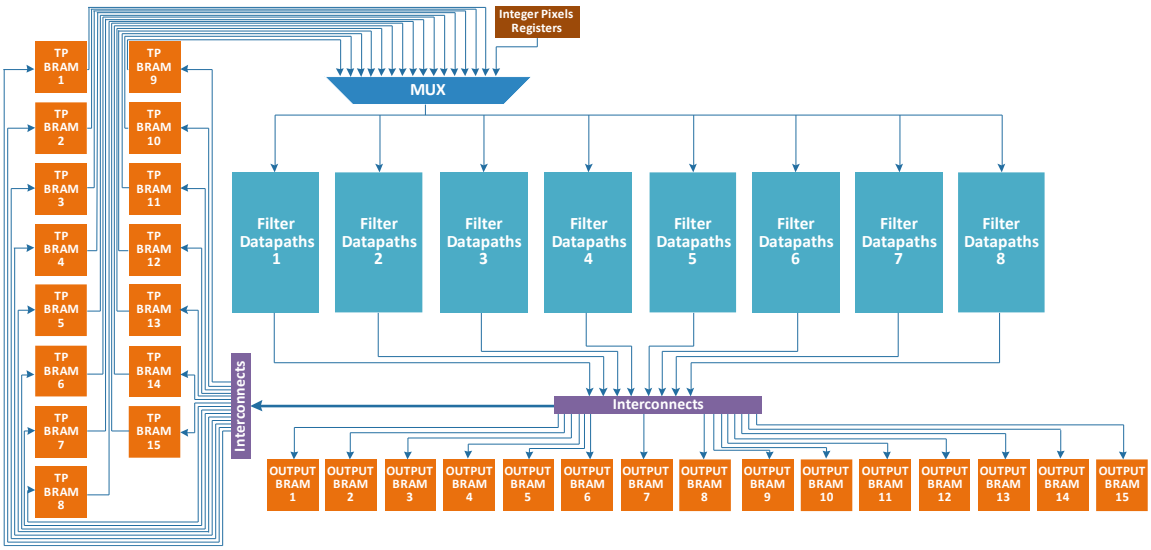

Figure 2.2 Approximate baseline VVC FI hardware for implementing F1 filter (BF1)

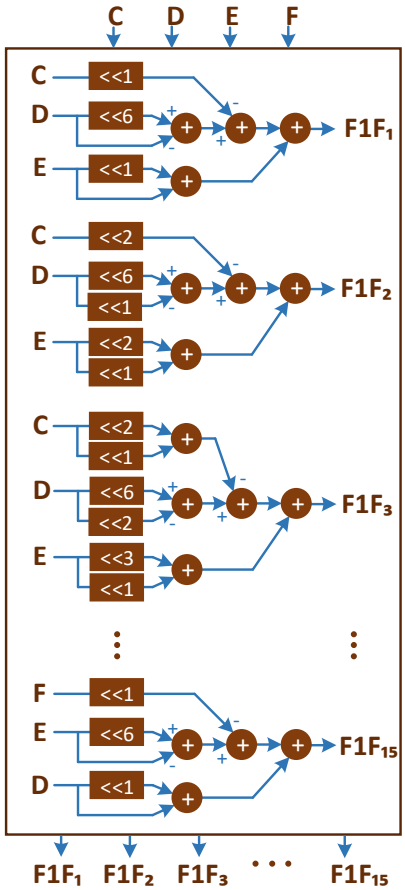

Figure 2.3 Filter datapaths hardware in BF1 hardware.

Since 255 FPs should be interpolated for every integer pixel, 64×255 FPs should be interpolated for an 8×8 PU. As shown in Figure 2.4, BF1 hardware interpolates FPs for an 8×8 PU in 147 clock cycles. It has 4 pipeline stages. It interpolates 8×15×15 horizontal half pixels, which will be used to interpolate quarter pixels, in 15 clock cycles and stores them into the transpose memories. It interpolates 8×8×15 vertical half pixels in 8 clock cycles. It interpolates 8×8×255 quarter pixels in 8×15 clock cycles using the horizontal half pixels in the transpose memories.

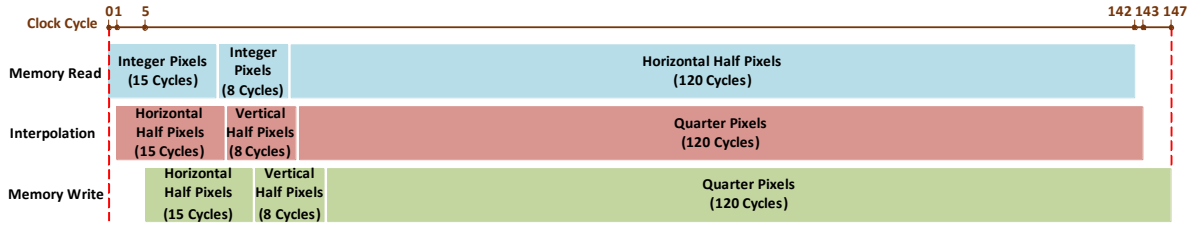

Figure 2.4 Scheduling of BF1, MCMF1, BF2, MCMF2 hardware.

BF1 hardware uses 30 Block RAMs (BRAM) as shown in Figure 2.2. It uses 15 BRAMs as transpose memories (TP BRAM) for storing the horizontal half pixels which are used to interpolate quarter pixels. It uses 15 BRAMs as output memories (OUTPUT BRAM) for storing output FPs.

Proposed MCMF1 hardware is shown in Figure 2.5. Interconnects boxes represent interconnects in MCMF1 hardware. They are used to simplify drawing interconnects in the figure.

MCMF1 hardware interpolates 8×15 FPs in parallel using 15 integer pixels or 15 horizontal half pixels in a clock cycle. It calculates three common offset values shown in Table 2.5 for 15 F1 FIR filters to reduce number of constant multiplications. These offset values are calculated in offset datapath (OD) using input pixels.

Since common offset values are used, each F1 FIR filter should be calculated using the coefficients shown in Table 2.5, and the result should be added with the required common offset value. As an example, offset 1 (O_1) equation and the first F1 FIR filter with offset ($F1OF_1$) equation are shown in (2.2) and (2.3), respectively.

$$O_1 = (-8 \times P_{-1} + 64 \times P_0 + 8 \times P_1) \tag{2.2}$$

$$F1OF_1 = (6 \times P_{-1} - P_0 - 5 \times P_1 + O_1) \gg 6 \tag{2.3}$$

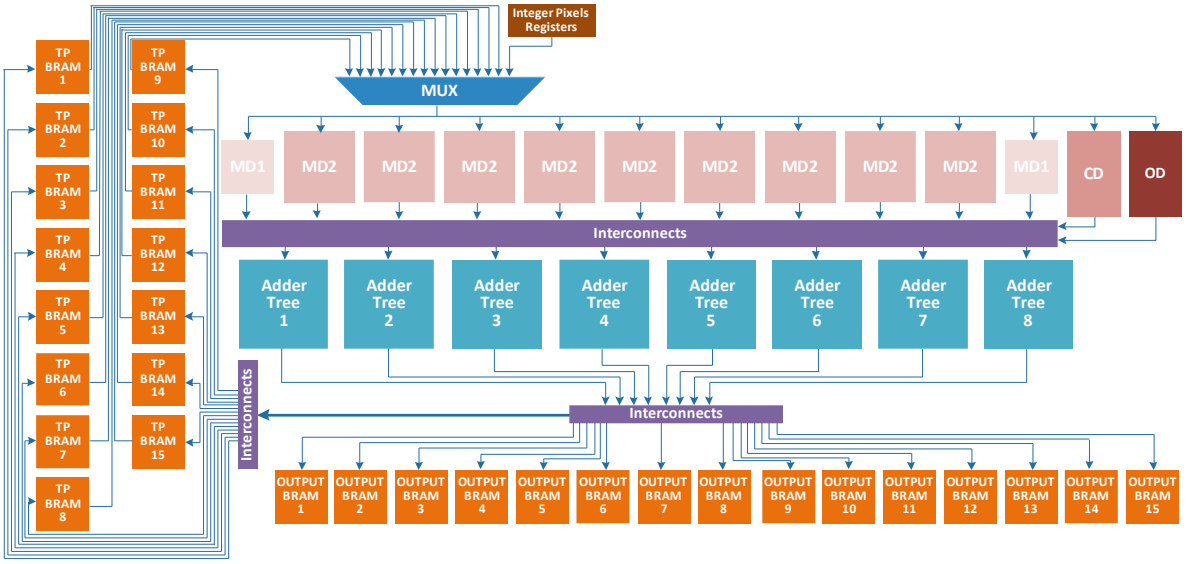

Figure 2.5 Proposed approximate MCM VVC FI hardware for implementing F1 filter (MCMF1)

Table 2.5 Coefficients of Proposed Approximate F1 FIR Filters with Offset

		Coefficients				
		P_{-1}	P_0	P_1	P_2	
Offsets	O_1	-8	64	8	0	Required Offset
	O_2	0	8	64	-8	
	O_3	-8	8	8	-8	
F1 FIR Filters with Offset	$F1OF_1$	6	-1	-5	0	O_1
	$F1OF_2$	4	-2	-2	0	O_1
	$F1OF_3$	2	-4	2	0	O_1
	$F1OF_4$	1	-6	5	0	O_1
	$F1OF_5$	0	-12	12	0	O_1
	$F1OF_6$	1	-17	16	0	O_1
	$F1OF_7$	0	-19	19	0	O_1
	$F1OF_8$	0	32	32	0	O_3
	$F1OF_9$	0	19	-19	0	O_2
	$F1OF_{10}$	0	16	-17	1	O_2
	$F1OF_{11}$	0	12	-12	0	O_2
	$F1OF_{12}$	0	5	-6	1	O_2
	$F1OF_{13}$	0	2	-4	2	O_2
	$F1OF_{14}$	0	-2	-2	4	O_2
	$F1OF_{15}$	0	-5	-1	6	O_2

As can be seen in Table 2.5, each input pixel should be multiplied with multiple constant coefficients. The constant multiplications of each input pixel when F1 FIR filters are calculated with and without using common offset values are shown in Table 2.6. In the table, P_{-4} to P_6 represent input pixels for FIR filters.

Table 2.6 Constant Multiplications in F1 FIR Filters

	Input Pixel	Constants	Datapath	Calculated Products
Without Offset	P_{-4}, P_6	1,2,4,6,7,8	MD1	3,7
	P_{-3} ... P_5	1,2,3,4,6,7,8,10, 13,20,24,27,40, 45,47,52, 58,60,62,63	MD2	3,5,7,13,15,27,29,31, 45,47,63
With Offset	P_{-4}, P_6	1,2,4,6	MD1	3
	P_{-3} … P_5	1,2,4,5,6,12, 16,17,19,32	MD2	3,5,17,19

Proposed MCMF1 hardware uses Hcub MCM technique [26] for implementing multiplications with multiple constant coefficients to reduce the number of adders. As shown in Table 2.6, since constant coefficients of input pixels (P_{-4}, P_6) and (P_{-3} … P_5) are different, two different MCM datapaths, MD1 and MD2, are used.

When the common offset values are used, the number of calculated products in MD1 is reduced from 2 to 1 and number of calculated products in MD2 is reduced from 11 to 4. Therefore, MCMF1 hardware uses the common offset values.

MD1, MD2 and OD in MCMF1 hardware are shown in Figure 2.6. MD1 takes pixel P_x as input, and it calculates $3\times P_x$ using adder and shifter. MD2 takes pixel P_x as input, and it calculates $3\times P_x$, $5\times P_x$, $17\times P_x$, $19\times P_x$ using adders and shifters. Since MCMF1 hardware interpolates 8×15 FPs in parallel, OD calculates 8 sets of three common offset values using adders and shifters. Each set of three offset values is used for interpolating 15 FPs.

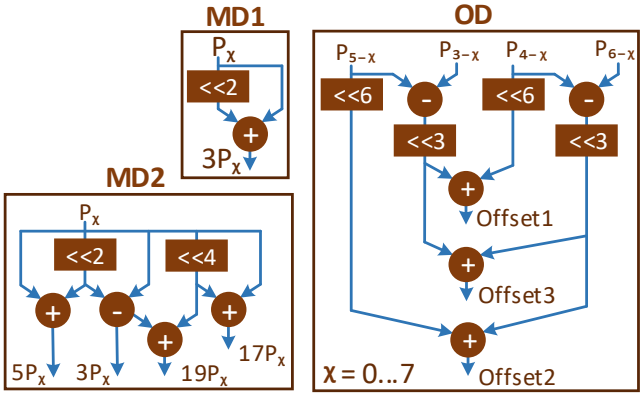

Figure 2.6 MD1, MD2, and OD in MCMF1 hardware.

As it can be seen in Table 2.5, there are common sub-expressions in F1 FIR filters with offset. The sub-expression ($-2\times P_0$ $-2\times P_1$) is common in FIR filters $F1OF_2$

and $F1OF_{14}$. The sub-expression $(-12\times P_0 +12\times P_1)$ in FIR filter $F1OF_5$ is negated version of the sub-expression $(12\times P_0-12\times P_1)$ in FIR filter $F1OF_{11}$. The sub-expression $(-19\times P_0+19\times P_1)$ in FIR filter $F1OF_7$ is negated version of the sub-expression $(19\times P_0-19\times P_1)$ in FIR filter $F1OF_9$. The common sub-expressions in F1 FIR filters with offset are calculated once in common datapath (CD) and the results are used in corresponding F1 FIR filters with offset.

MCMF1 hardware interpolates 8×15 FPs in parallel in a clock cycle. 15 FPs for an input pixel are interpolated using 15 F1 FIR filters shown in Table 2.5. There are common sub-expressions in F1 FIR filters with offset used for interpolating FPs for adjacent input pixels. FIR filter $F1OF_{13}$ equation for each input pixel, i.e. $(2\times P_0-4\times P_1+2\times P_2)$, is the same as FIR filter $F1OF_3$ equation for the adjacent input pixel. Negated version of the sub-expression $(-6\times P_1+P_2)$ in FIR filter $F1OF_{12}$ equation for each input pixel exists in FIR filter $F1OF_1$ equation for the adjacent input pixel. Negated version of the sub-expression $(-P_1+6\times P_2)$ in FIR filter $F1OF_{15}$ equation for each input pixel exists in FIR filter $F1OF_4$ equation for the adjacent input pixel. All these common sub-expressions are also calculated once in CD and their results are used in corresponding FIR filters.

As shown in Figure 2.5, after results of MD1, MD2, CD and OD are generated, adder trees calculate FPs by adding these results. As shown in Figure 2.4, MCMF1 hardware interpolates the FPs for an 8×8 PU in 147 clock cycles same as BF1 hardware.

MCMF1 hardware uses 30 Block RAMs (BRAM) as shown in Figure 2.5. It uses 15 BRAMs as transpose memories (TP BRAM) for storing the horizontal half pixels which are used to interpolate quarter pixels. It uses 15 BRAMs as output memories (OUTPUT BRAM) for storing output FPs.

Proposed BF2 hardware is similar to BF1 hardware shown in Figure 2.2. BF2 hardware also has 8 filter datapaths hardware. However, BF2 filter datapaths hardware is different than BF1 filter datapaths hardware. 15 F1 FIR filters are implemented using 15 parallel datapaths in one BF1 filter datapaths hardware. However, as shown in Figure 2.7, 11 F2 FIR filters are implemented using 11 parallel datapaths in one BF2 filter datapaths hardware. C, D, E, F inputs represent the pixels given to each filter datapaths. For example, P_{-1}, P_0, P_1, P_2 are given to Filter Datapaths 4 as inputs C, D, E, F, respectively.

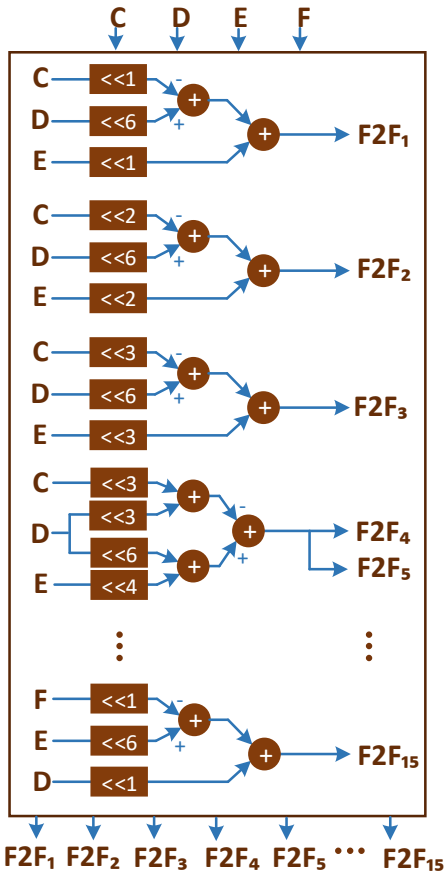

Figure 2.7 Filter datapaths hardware in BF2 hardware.

As shown in Table 2.3, F2 FIR filters $F2F_5$, $F2F_7$, $F2F_{10}$, $F2F_{12}$ are the same as F2 FIR filters $F2F_4$, $F2F_6$, $F2F_9$, $F2F_{11}$, respectively. Therefore, BF2 hardware only calculates FIR filters $F2F_5$, $F2F_7$, $F2F_{10}$, $F2F_{12}$ and their results are used for FIR filters $F2F_4$, $F2F_6$, $F2F_9$, $F2F_{11}$, respectively.

Most of the coefficients of F2 FIR filters are 2^n values. Multiplications with these coefficients are performed using only shift operations. Therefore, BF2 filter datapaths hardware has less adders than BF1 filter datapaths hardware.

As shown in Figure 2.4, BF2 hardware interpolates the FPs for an 8×8 PU in 147 clock cycles same as BF1 hardware. BF2 hardware uses 15 BRAMs as transpose memories (TP BRAM) for storing the horizontal half pixels which are used to interpolate quarter pixels. However, since it calculates 11 FIR filters, it uses 11 BRAMs as output memories (OUTPUT BRAM) for storing output FPs.

Proposed MCMF2 hardware is similar to MCMF1 hardware shown in Figure 2.5. MD1 and OD in MCMF2 and MCMF1 hardware are the same. MCMF1 hardware has 2 MD1 and 9 MD2. However, MCMF2 hardware has 11 MD1, and it does not have any MD2. CD in MCMF2 hardware is different than the CD in MCMF1 hardware.

Adder trees in MCMF2 and MCMF1 hardware are also different.

MCMF2 hardware interpolates 8×15 FPs in parallel using 15 integer pixels or 15 horizontal half pixels in a clock cycle. It calculates three common offset values shown in Table 2.7 for 15 F2 FIR filters to reduce number of constant multiplications. These offset values are calculated in OD using input pixels.

Table 2.7 Coefficients of Proposed Approximate F2 FIR Filters with Offset

		Coefficients				
		P_{-1}	P_0	P_1	P_2	
Offsets	O_1	-8	64	8	0	Required Offset
	O_2	0	8	64	-8	
	O_3	-8	8	8	-8	
F2 FIR Filters with Offset	$F2OF_1$	6	0	-6	0	O_1
	$F2OF_2$	4	0	-4	0	O_1
	$F2OF_3$	0	0	0	0	O_1
	$F2OF_4$	0	-8	8	0	O_1
	$F2OF_5$	0	-8	8	0	O_1
	$F2OF_6$	0	-24	24	0	O_1
	$F2OF_7$	0	-24	24	0	O_1
	$F2OF_8$	0	32	32	0	O_3
	$F2OF_9$	0	24	-24	0	O_2
	$F2OF_{10}$	0	24	-24	0	O_2
	$F2OF_{11}$	0	8	-8	0	O_2
	$F2OF_{12}$	0	8	-8	0	O_2
	$F2OF_{13}$	0	0	0	0	O_2
	$F2OF_{14}$	0	-4	0	4	O_2
	$F2OF_{15}$	0	-6	0	6	O_2

Since common offset values are used, each F2 FIR filter should be calculated using the coefficients shown in Table 2.7, and the result should be added with the required common offset value. As shown in Table 2.7, FIR filters $F2OF_4$, $F2OF_6$, $F2OF_9$, $F2OF_{11}$ are the same as FIR filters $F2OF_5$, $F2OF_7$, $F2OF_{10}$, $F2OF_{12}$, respectively. Therefore, MCMF2 hardware only calculates FIR filters $F2OF_4$, $F2OF_6$, $F2OF_9$, $F2OF_{11}$, and their results are used for FIR filters $F2OF_5$, $F2OF_7$, $F2OF_{10}$, $F2OF_{12}$, respectively.

As it can be seen in Table 2.7, each input pixel should be multiplied with multiple constant coefficients. The constant multiplications of each input pixel when F2 FIR filters are calculated with and without using common offset values are shown in Table 2.8. In the table, P_{-4} to P_6 represent input pixels for FIR filters.

Table 2.8 Constant Multiplications in F2 FIR Filters

	Input Pixel	Constants	Datapath	Calculated Products
Without Offset	P_{-4}, P_6	2,4,8	MD1	-
	P_{-3} ... P_5	2,4,8,16, 32,40,56,64	MD2	5,7
With Offset	P_{-4}, P_6	4,6	MD1	3
	P_{-3} … P_5	4,6,8,24,32	MD1	3

Proposed MCMF2 hardware uses Hcub MCM technique [26] for implementing multiplications with multiple constant coefficients to reduce the number of adders. As shown in Table 2.8, when the common offset values are used, products calculated for input pixels (P_{-4}, P_6) and (P_{-3} … P_5) are the same. Therefore, MCMF2 hardware uses only MD1 MCM datapath. It does not use MD2 MCM datapath.

As can be seen in Table 2.7, there are common sub-expressions in F2 FIR filters with offset. The sub-expression ($-8\times P_0+8\times P_1$) in FIR filter $F2OF_4$ is negated version of the sub-expression ($8\times P_0-8\times P_1$) in FIR filter $F2OF_{11}$. The sub-expression ($-24\times P_0+24\times P_1$) in FIR filter $F2OF_6$ is negated version of the sub-expression ($24\times P_0-24\times P_1$) in FIR filter $F2OF_9$. The common sub-expressions in F2 FIR filters with offset are calculated once in CD and the results are used in corresponding F2 FIR filters with offset.

MCMF2 hardware interpolates 8×15 FPs in parallel in a clock cycle. 15 FPs for an input pixel are interpolated using 11 F2 FIR filters shown in Table 2.7. There are common sub-expressions in F2 FIR filters with offset used for interpolating FPs for adjacent input pixels. Negated version of the sub-expression ($-6\times P_0+6\times P_2$) in FIR filter $F2OF_{15}$ equation for each input pixel exists in FIR filter $F2OF_1$ equation for the adjacent input pixel. Negated version of the sub-expression ($-4\times P_0+4\times P_2$) in FIR filter $F2OF_{14}$ equation for each input pixel exists in FIR filter $F2OF_2$ equation for the adjacent input pixel. All these common sub-expressions are also calculated once in CD and their results are used in corresponding FIR filters.

After results of MD1, MD2, CD and OD are generated, adder trees calculate FPs by adding these results. As shown in Figure 2.4, MCMF2 hardware interpolates the FPs for an 8×8 PU in 147 clock cycles same as MCMF1 hardware.

MCMF2 hardware uses 15 BRAMs as transpose memories (TP BRAM) for storing the horizontal half pixels which are used to interpolate quarter pixels. However,

since it calculates 11 FIR filters, it uses 11 BRAMs as output memories (OUTPUT BRAM) for storing output FPs.

2.4 Implementation Results of the Proposed Approximate VVC FI Hardware

Proposed BF1, MCMF1, BF2, MCMF2 approximate VVC FI hardware are implemented using Verilog HDL. Verilog RTL codes are implemented to a 28 nm FPGA. The FPGA implementations are verified with post implementation timing simulations. The simulation results matched the results of a software implementation of the proposed approximate VVC FI filters F1 and F2.

Power consumptions of the FPGA implementations are estimated using a gate level power estimation tool. Post implementation timing simulations are performed for one frame of full HD (1920×1080) video sequences Tennis (T) and Kimono (K) at 100 MHz [35]. For each FPGA implementation, signal activities of its post implementation timing simulation are stored into a value change dump (VCD) file, and its power consumption is estimated using this VCD file.

Implementation results of the FPGA implementations are shown in Table 2.9. In the table, they are also compared with implementation results of the exact VVC FI baseline and MCM hardware proposed in [29]. The proposed approximate VVC FI hardware are similar to the exact VVC FI hardware proposed in [29]. However, since the proposed approximate VVC FI FIR filters are different than the exact VVC FI FIR filters, their MCM datapaths, adder trees, common datapath and offset datapath are different. Since they implement the proposed approximate VVC FI FIR filters, the proposed BF1, MCMF1, BF2, MCMF2 hardware are smaller and faster than the exact VVC FI hardware proposed in [29].

BF2 and MCMF2 hardware are smaller and faster than BF1 and MCMF1 hardware. However, they have slightly worse rate-distortion performance than BF1 and MCMF1 hardware. MCMF1 hardware is smaller than BF1 hardware. MCMF2 hardware is smaller than BF2 hardware.

Power consumptions of the FPGA implementations are shown in Table 2.10. In the table, they are also compared with power consumptions of the exact VVC FI baseline and MCM hardware proposed in [29]. Since they implement the proposed approximate VVC FI FIR filters, the proposed BF1, MCMF1, BF2, MCMF2 hardware have lower power consumption than the exact VVC FI hardware proposed in [29].

MCMF1 and MCMF2 hardware has up to 44% and 51% lower power consumption than the exact VVC FI MCM hardware proposed in [29], respectively. MCMF2 hardware has lower power consumption than MCMF1 hardware. However, MCMF2 hardware has slightly worse rate-distortion performance than MCMF1 hardware. MCMF1 hardware has lower power consumption than BF1 hardware. MCMF2 hardware has lower power consumption than BF2 hardware.

Table 2.9 Implementation Results of the Proposed Approximate VVC FI Hardware

	Exact Baseline VVC FI [29]	Exact MCM VVC FI [29]	Proposed BF1 [25]	Proposed MCMF1	Proposed BF2	Proposed MCMF2
FPGA	28 nm	28 nm	28 nm	28 nm	28 nm	28 nm
Slices	5205	3718	3083	2636	2397	2205
DFFs	6408	3461	3515	3290	2279	2114
LUTs	16334	11599	9313	7973	6974	6357
BRAMs	30	30	30	30	30	30
Max. Freq. (MHz)	208	200	227	227	236	236
Frames per Second	42 1920×1080	40 1920×1080	47 1920×1080	47 1920×1080	49 1920×1080	49 1920×1080

Table 2.10 Power Consumption Results of Proposed Approximate VVC FI Hardware

	Exact Baseline VVC FI [29]		Exact MCM VVC FI [29]		Proposed BF1 [25]		Proposed MCMF1		Proposed BF2		Proposed MCMF2	
Video	T	K	T	K	T	K	T	K	T	K	T	K
Clock (mW)	52	52	27	27	26	26	23	23	16	16	16	16
Signal (mW)	162	218	144	193	76	107	58	83	61	82	60	82
Logic (mW)	141	194	109	151	60	90	42	64	38	54	33	48
BRAM (mW)	93	95	93	94	94	95	88	89	81	82	76	77
Total (mW)	448	559	373	465	256	318	211	259	196	234	185	223
Reduction (%) Compared to [29]	---	---	---	---	31%	31%	43%	44%	47%	49%	49%	51%

MCMF1 and MCMF2 hardware are compared with several HEVC FI hardware in the literature in Table 2.11. The results shown as "---" are not reported in the corresponding paper. Since VVC FI algorithm is different than HEVC FI algorithm, MCMF1 and MCMF2 hardware are different than the HEVC FI hardware. Since VVC FI has higher computational complexity than HEVC FI, implementation results of MCMF1 and MCMF2 hardware are worse than implementation results of the HEVC FI hardware.

Table 2.11 Comparison of the Proposed Hardware with HEVC FI Hardware

	[30]	[31]	[32]	[33]	Filter 1 [34]	Filter 2 [34]	Proposed MCMF1	Proposed MCMF2
Standard	HEVC	HEVC	HEVC	HEVC	HEVC	HEVC	VVC	VVC
FPGA	40 nm	65 nm	40 nm	65 nm	40 nm	40 nm	28 nm	28 nm
Slices	1557	---	---	2181	834	731	2636	2205
LUTs	3929	28486	24202	5017	2008	1567	7973	6357
Freq. (MHz)	200	120	200	283	278	278	227	236
fps	30 3840×2160	---	60 1920×1080	30 2560×1600	45 3840×2160	45 3840×2160	47 1920×1080	49 1920×1080
Power	93 mW	---	171 mW	89 mW	88 mW	80 mW	237 mW	207 mW

2.5 Proposed VVC FI Hardware Using Memory Based Constant Multiplication

In this thesis, a novel VVC FI hardware using memory based constant multiplication for all prediction unit (PU) sizes is proposed. Memory based constant multiplication is an efficient computation technique [36], [37]. The proposed hardware stores pre-computed products of an input pixel with multiple constant coefficients in memory. It implements multiplications with constant coefficients using these pre-computed products. Several optimizations are proposed to reduce memory size.

The proposed VVC FI hardware interpolates 255 fractional pixels for each integer pixel. Therefore, it can be used for fractional motion estimation. It is implemented using Verilog HDL. It, in the worst case, can process 49 full HD (1920×1080) video frames per second. It has up to 9.4% less power consumption than VVC FI hardware in the literature.

VVC FI hardware proposed in [29] uses a common offset value and Hcub multiplierless constant multiplication (MCM) algorithm [26] to reduce number of additions. It interpolates 255 fractional pixels for each integer pixel. VVC FI hardware proposed in [28] interpolates one fractional pixel for each integer pixel. Therefore, it can only be used for fractional motion compensation.

Approximate VVC FI hardware proposed in [25] interpolates 255 fractional pixels for each integer pixel. It has smaller area and lower power consumption than the VVC FI hardware proposed in [29]. However, since it is an approximate VVC FI hardware, it has worse rate-distortion performance than the VVC FI hardware proposed in [29].

HEVC FI hardware proposed in [38] uses memory based constant multiplication. However, none of the VVC FI hardware in the literature uses memory based constant multiplication.

The proposed VVC FI hardware for all PU sizes is shown in Figure 2.8. The splitters represent wire interconnections. They are used to simplify drawing interconnects in the figure. In the proposed hardware, all fractional pixels (half pixels and quarter pixels) are interpolated for the luma component of an 8×8 PU. For larger PU sizes, the PU is decomposed into 8×8 blocks, and the blocks are interpolated separately.

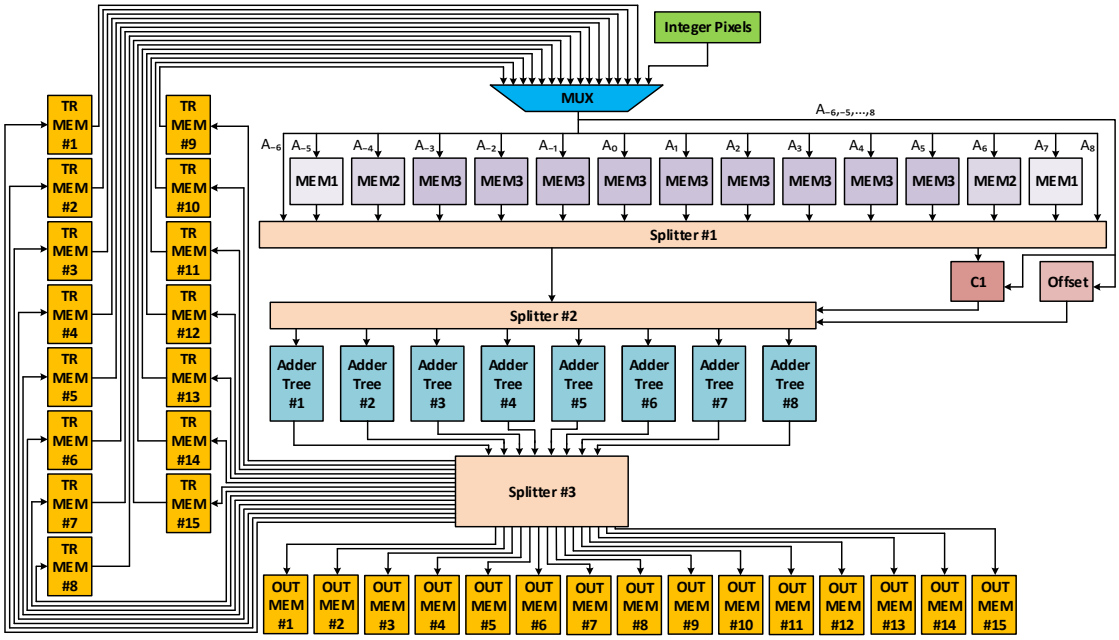

Figure 2.8 Proposed VVC FI hardware using memory based constant multiplication

The proposed hardware interpolates fractional pixels using the common offset value proposed in [29]. When this common offset value is used, coefficients shown in Table 2.12 should be used for VVC FI FIR filters [29].

Since coefficients of FIR filters F_9 to F_{15} are symmetric with coefficients of FIR filters F_7 to F_1, their coefficients are not shown in Table 2.12. Sub-expression $(A_{-3} - 3A_{-2})$ is common for FIR filters F_1, F_{12}, F_{13}, F_{14}, and F_{15}. Sub-expression $(A_4 - 3A_3)$ is common for FIR filters F_1, F_2, F_3, F_4, and F_{15}.

Table 2.12 Coefficients of VVC FI FIR Filters with Offset [29]

FIR Filters	Coefficients							
	A_{-3}	A_{-2}	A_{-1}	A_0	A_1	A_2	A_3	A_4
Offset	-1	4	-8	32	32	-8	4	-1
F_1	1	-3	5	31	-28	6	-3	1
F_2	0	-2	3	30	-24	5	-3	1
F_3	0	-1	0	28	-19	4	-3	1
F_4	0	0	-2	26	-15	3	-3	1
F_5	0	0	-3	20	-6	0	-1	0
F_6	0	-1	-1	15	-1	-2	0	0
F_7	0	0	-3	13	2	-2	0	0
F_8	0	0	-3	8	8	-3	0	0

Same as the hardware proposed in [29], the proposed hardware calculates the offset values in Offset datapath using input pixels. It calculates the common sub-expressions once in C1 datapath and uses the results in corresponding equations.

The proposed hardware interpolates 8×15 fractional pixels in parallel using 15 integer pixels or 15 horizontal half pixels in each clock cycle. 15 input pixels $A_{-6}, \ldots, A_8$ should be multiplied with multiple constant coefficients during the interpolation of 8×15 fractional pixels. The constant coefficient multiplications necessary for each input pixel are shown in Table 2.13.

Table 2.13 Constant Coefficient Multiplications for Input Pixels

Input Pixel	Constant Coefficient Multiplications	Hardware	Stored Products
A_{-6}	1	---	---
A_{-5}	-1, -2, -3	MEM1	-3
A_{-4}	-1, -2, ±3, 4, 5, 6	MEM2	-3, 5
A_{-3},…, A_5	-1, 2, -6, 8, 13, ±15, -19, 20, -24, 26, ±28, 30, 31	MEM3	-3, 5, -7, 13, -15, -19, 31
A_6	-1, -2, ±3, 4, 5, 6	MEM2	-3, 5
A_7	-1, -2, -3	MEM1	-3
A_8	1	---	---

In the proposed hardware, memory based constant multiplication technique is used for implementing constant coefficient multiplications. As it can be seen in Table 2.13, constant coefficients for input pixels "A_{-5}, A_7", "A_{-4}, A_6", and "$A_{-3}, \ldots, A_5$" are different. Therefore, three memories, MEM1, MEM2, and MEM3, are used for storing pre-computed products of an input pixel with multiple constant coefficients. Since multiplications with the coefficients that are powers of 2 are calculated using shift operation, there is no need to pre-compute and store multiplications with them. Therefore, for input pixel A, only the constant multiplication $-3A$ is stored in MEM1,

and only the constant multiplications $-3A$ and $5A$ are stored in MEM2. $6A$ is calculated from $-3A$ using shift operation.

For input pixels $A_{-3}, \dots, A_5$, seven constant multiplications $-3A$, $5A$, $-7A$, $13A$, $-15A$, $-19A$, and $31A$ are stored in MEM3. $20A$, $-24A$, $26A$, $\pm 28A$, and $30A$ are calculated from $5A$, $-3A$, $13A$, $-7A$, and $-15A$, respectively, using shift operations. After constant coefficient multiplications are performed by memory based constant multiplication technique, the fractional pixels are calculated using adder trees.

The proposed hardware uses 8-bit unsigned input pixel A as the address of the memories MEM1, MEM2, and MEM3. MEM1 stores one constant multiplication $-3A$ in each address. MEM2 and MEM3 store two and seven constant multiplications in each address, respectively. Therefore, MEM1, MEM2, and MEM3 store $2^8 \times 1$, $2^8 \times 2$, and $2^8 \times 7$ constant multiplications, respectively. Multiplication of an 8-bit unsigned input pixel with constant coefficients -3, 5, -7, 13, -15, -19, and 31 are 11 bits, 11 bits, 12 bits, 12 bits, 13 bits, 14 bits, and 13 bits, respectively. Therefore, in each address of MEM1, MEM2, and MEM3 11 bits, 11+11=22 bits, and 11+11+12+12+13+14+13=86 bits should be stored, respectively. We propose several optimizations to reduce the sizes of these memories.

$-3A$ can be implemented with addition and shift operations as shown in equation (2.4) and Figure 2.9. As shown in Figure 2.9, the least significant two bits of $-3A$ are equal to the least significant two bits of A. There is no need to store these two bits in memories. As shown in Figure 2.9, the third least significant bit of $-3A$ can be calculated by adding 1, $\overline{A[0]}$, and $A[2]$. The result of this 1-bit addition sum = $-3A[2]$ and carry-out = carry[2] are shown in equations (2.5) and (2.6), respectively.

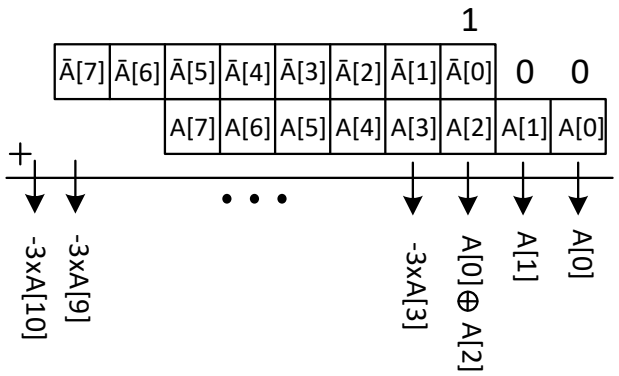

Figure 2.9 Implementation of -3A with addition and shift operations.

$$-3A = A + ((\bar{A} + 1) \ll 2) \quad (2.4)$$

$$-3A[2] = 1\hat{}\overline{A[0]}\hat{}A[2] = A[0]\hat{}A[2] \quad (2.5)$$

$$carry[2] = (1\&\overline{A[0]}) \mid \left(A[2]\&(1\hat{}\overline{A[0]})\right) = \overline{A[0]} \mid (A[2]\&A[0]) = \overline{A[0]} \mid A[2] \quad (2.6)$$

$-3A[3]$ can be calculated by adding $\overline{A[1]}$, $A[3]$ and carry[2] as shown in equation (2.7).

$$-3A[3] = \overline{A[1]}\hat{}A[3]\hat{}carry[2] = \overline{A[1]}\hat{}A[3]\hat{}(\overline{A[0]} \mid A[2]) \quad (2.7)$$

For an 8-bit A, i.e. $0 \leq A \leq 255$, $-3A[9] = 0$ only for $A = 0$ and $171 \leq A \leq 255$, i.e. $A = 00000000$ and $10101011 \leq A \leq 11111111$. By representing the ranges $10101011 \leq A \leq 10111111$ with $A[7]\&\overline{A[6]}\&A[5]\&(A[4] \mid (A[3]\&(A[2] \mid (A[1]\&A[0]))))$ and $11000000 \leq A \leq 11111111$ with $A[7]\&A[6]$, $-3A[9]$ can be calculated as shown in equation (2.8). Since A is an unsigned number, $-3A$ is always negative. Therefore, its sign bit, i.e. $-3A[10]$, is always 1. Therefore, in each address of MEM1, only 5 bits $-3A[8], \ldots, -3A[4]$ are stored instead of 11 bits. The other bits are obtained from A as explained.

$$\begin{aligned} -3A[9] = & \\ & \overline{(A[7]\&A[6]) \mid \left(A[7]\&\overline{A[6]}\&A[5]\&\left(A[4] \mid (A[3]\&(A[2] \mid (A[1]\&A[0]))\right)\right) \mid (\overline{A[7]}\&\overline{A[6]}\&\ldots\&\overline{A[0]})} \\ & = \overline{A[7]\&\left(A[6] \mid \left(A[5]\&\left(A[4] \mid (A[3]\&(A[2] \mid (A[1]\&A[0]))\right)\right)\right)\&(A[7] \mid A[6] \mid \ldots \mid A[0])} \end{aligned} \quad (2.8)$$

5A, -7A, 13A, -15A, -19A, and 31A can be implemented with addition and shift operations as shown in equations (2.9)-(2.14) and Figure 2.10.

$$5A = (A \ll 2) + A \quad (2.9)$$

$$-7A = A + ((\bar{A} + 1) \ll 3) \quad (2.10)$$

$$13A = (A \ll 3) + 5A \quad (2.11)$$

$$-15A = A + ((\bar{A} + 1) \ll 4) \quad (2.12)$$

$$-19A = 13A + ((\bar{A} + 1) \ll 5) \quad (2.13)$$

$$31A = (A \ll 4) + 15A \quad (2.14)$$

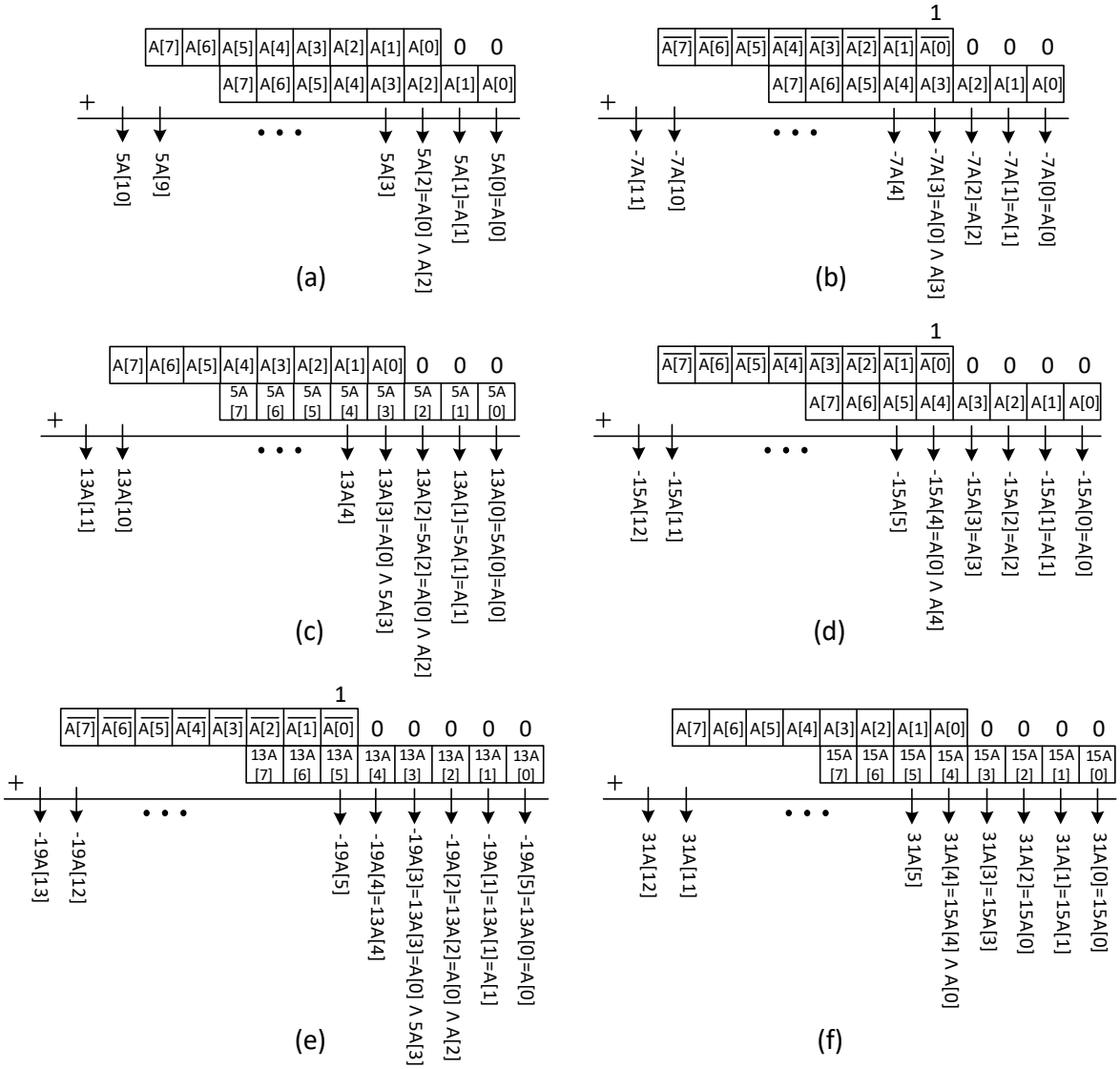

Figure 2.10 Implementation of constant multiplications with addition and shift operations (a) 5A (b) -7A (c) 13A (d) -15A (e) -19A (f) 31A.

As shown in equations (2.15), to calculate $31A[5]$ six least significant bits of $15A$ should first be calculated by calculating two's complement of $-15A$.

For an 8-bit A, i.e. $0 \leq A \leq 255$, $5A[10] = 1$ only for $205 \leq A \leq 255$, i.e. $11001101 \leq A \leq 11111111$, $-7A[10] = 1$ only for $1 \leq A \leq 146$, i.e. $00000001 \leq A \leq 10010010$, $13A[11] = 1$ only for $158 \leq A \leq 255$, i.e. $10011110 \leq A \leq 11111111$, $-15A[11] = 1$ only for $1 \leq A \leq 136$, i.e. $00000001 \leq A \leq 10001000$, $-19A[12] = 1$ only for $1 \leq A \leq 215$, i.e. $00000001 \leq A \leq 11010111$, $31A[12] = 1$ only for $133 \leq A \leq 255$, i.e. $10000101 \leq A \leq 11111111$, resulting in equations

(2.16)-(2.21). Since A is an unsigned number, $-7A$, $-15A$, and $-19A$ are always negative. Therefore, their sign bits, i.e. $-7A[11]$, $-15A[12]$, and $-19A[13]$, are always 1.

$$5A[3] = A[3]\text{^}A[1]\text{^}(A[0]\&A[2])$$

$$-7A[4] = \overline{A[1]}\text{^}A[4]\text{^}(\overline{A[0]} \mid A[3])$$

$$13A[4] = 5A[4]\text{^}A[1]\text{^}(A[0]\&5A[3])$$

$$-15A[5] = \overline{A[1]}\text{^}A[5]\text{^}(\overline{A[0]} \mid A[4]) \quad (2.15)$$

$$-19A[5] = A[0]\text{^}13A[5]$$

$$-19A[6] = \overline{A[1]}\text{^}13A[6]\text{^}(\overline{A[0]} \mid 13A[5])$$

$$31A[5] = 15A[5]\text{^}A[1]\text{^}(A[0]\&15A[4])$$

$$5A[10] = A[7]\&A[6]\&(A[5] \mid A[4] \mid (A[3]\&A[2]\&(A[1] \mid A[0]))) \quad (2.16)$$

$$-7A[10] = \overline{\left(\overline{A[7]}\&(A[6] \mid A[5] \mid \ldots \mid A[0])\right) \mid (A[7]\&A[6] \mid A[5] \mid \left(A[4]\&\left(A[3] \mid A[2] \mid (A[1]\&A[0])\right)\right))} \quad (2.17)$$

$$13A[11] = A[7]\&(A[6] \mid A[5] \mid (A[4]\&A[3]\&A[2]\&A[1])) \quad (2.18)$$

$$-15A[11] = (\overline{A[7]}\&(A[6] \mid A[5] \mid \ldots \mid A[0])) \mid \overline{A[6] \mid A[5] \mid A[4] \mid (A[3]\&(A[2] \mid A[1] \mid A[0]))} \quad (2.19)$$

$$-19A[12] = (\overline{A[7]}\&(A[6] \mid A[5] \mid \ldots \mid A[0])) \mid (A[7]\&\overline{A[6]\&\left(A[5] \mid (A[4]\&A[3])\right)}) \quad (2.20)$$

$$31A[12] = A[7]\&(A[6] \mid A[5] \mid A[4] \mid A[3] \mid (A[2]\&(A[1] \mid A[0]))) \quad (2.21)$$

Therefore, it is necessary to store only six bits $5A[9], \ldots, 5A[4]$, five bits $-7A[9], \ldots, -7A[5]$, six bits $13A[10], \ldots, 13A[5]$, five bits $-15A[10], \ldots, -15A[6]$, five bits $-19A[11], \ldots, -19A[7]$, and six bits $31A[11], \ldots, 31A[6]$ in the memories.

Therefore, 5+6=11 bits are stored in each address of MEM2 instead of 22 bits, and 5+6+5+6+5+5+6=38 bits are stored in each address of MEM3 instead of 86 bits.

As shown in Figure 2.8, the proposed hardware uses 30 Block RAMs (BRAM). It uses 15 BRAMs as output memories (OUT MEM) to store fractional pixels and 15 BRAMs as transpose memories (TR MEM) to store horizontal half pixels required for interpolating quarter pixels. Each BRAM address stores eight interpolated pixels.

First, it takes 15 clock cycles to interpolate 8×15×15 horizontal half pixels required for interpolating quarter pixels. After all horizontal half pixels are calculated and stored in the transpose BRAMs in 15 clock cycles, 15 pixels required for interpolating quarter pixels can always be read in one clock cycle from 15 different transpose BRAMs. Then, it takes 8 clock cycles to interpolate 8×8×15 vertical half pixels. Finally, in 8×15 clock cycles, 8×8×255 quarter pixels are interpolated using horizontal half pixels. There are four pipeline stages in the proposed hardware. Therefore, the proposed hardware interpolates all the fractional pixels for an 8×8 PU in 147 clock cycles.

2.6 Implementation Results of the Proposed VVC FI Hardware Using Memory Based Constant Multiplication

The proposed VVC FI hardware using memory based constant multiplication for all PU sizes is implemented using Verilog HDL. The Verilog RTL code is verified with RTL simulations.

The proposed hardware is compared with the VVC FI hardware proposed in [29]. To have a fair comparison, Verilog RTL codes of both hardware are synthesized and implemented to the same Xilinx XC7VX330T-3FFG1157 FPGA using Xilinx Vivado 2017.4. Both FPGA implementations are verified with post implementation timing simulations.

Implementation results are shown in Table 2.14. The proposed VVC FI hardware works at 235 MHz, and it can process 49 full HD (1920×1080) video frames per second. The proposed VVC FI hardware has less DFFs but more LUTs and slices than the VVC FI hardware proposed in [29].

Power consumptions of both FPGA implementations are estimated for Tennis and Kimono full HD (1920×1080) videos [35] using Xilinx Vivado. To estimate power consumption of an FPGA implementation, its post implementation timing simulation is done at 100 MHz using Mentor Graphics QuestaSim for one frame of each video

sequence, and signal activities are stored in a SAIF file. Xilinx Vivado estimates power consumption of that FPGA implementation using this SAIF file.

Power consumption results are shown in Table 2.15. The proposed VVC FI hardware has up to 9.4% less power consumption than the VVC FI hardware proposed in [29].

Table 2.14 Implementation Results of the Proposed Memory Based VVC FI Hardware

	[29]	Proposed VVC FI Hardware
FPGA	Xilinx Virtex7	Xilinx Virtex7
Slices	3121	3348
DFFs	3589	3525
LUTs	10731	11842
BRAMs	30	30
Max. Freq. (MHz)	219	235
Frames per Second	46 1920×1080	49 1920×1080

Table 2.15 Power Consumption of the Proposed Memory Based VVC FI Hardware

	[29]		Proposed VVC FI Hardware	
Video	Tennis	Kimono	Tennis	Kimono
Clock (mW)	25	25	29	29
Signal (mW)	172	238	160	220
Logic (mW)	203	288	169	237
BRAM (mW)	137	138	137	138
Total (mW)	537	689	495	624
Power Reduction Compared to [29]	---	---	7.8%	9.4%

CHAPTER III

NOVEL DECOMPOSED COEFFICIENTS BASED HEVC AND VVC FRACTIONAL INTERPOLATION HARDWARE

Motion estimation (ME) is the most computationally complex part of HEVC and VVC video encoders. ME consists of integer ME and fractional ME. Fractional ME requires interpolation of fractional pixels. Fractional interpolation (FI) is one of the most computationally intensive parts of HEVC and VVC video encoders and decoders. On average, one fourth of the HEVC encoder complexity and 50% of the HEVC decoder complexity are caused by FI.

In HEVC fractional interpolation (HFI), 3 horizontally interpolated pixels (HIPs), 3 vertically interpolated pixels (VIPs), 9 horizontally and vertically interpolated pixels (HVIPs) are interpolated for every integer pixel (IP). In VVC fractional interpolation (VFI), 15 HIPs, 15 VIPs, 225 HVIPs are interpolated for every IP. Hence, VFI has much higher computational complexity than HFI. Therefore, it is required to develop HFI hardware and VFI hardware to implement HFI and VFI in real-time, respectively.

In this thesis, we propose novel decomposed coefficients technique for implementing HFI, and we propose HFI hardware using the proposed technique. The decomposed coefficients technique reduces the number of additions by decomposing the coefficients of the FIR filters. We apply the decomposed coefficients technique to

exact and approximate VFI algorithms, and we propose exact VFI hardware and approximate VFI hardware using the proposed technique.

The proposed FI hardware are implemented using Verilog HDL. The proposed HFI hardware has higher performance, less area, and less power consumption than the best HFI hardware in the literature. It can process 50 quad full HD (QFHD) (3840×2160) video frames per second (fps). The proposed VFI hardware has higher performance, less area, and less power consumption than the best VFI hardware in the literature. It can process 48 full HD (FHD) (1920×1080) video fps. The proposed approximate VFI hardware have the same performance, less area, and less power consumption than the best approximate VFI hardware in the literature. They can process 49 and 52 full HD (1920×1080) video fps.

Several HFI hardware [30]-[33], [38], [39] and several VFI hardware [18], [22], [25], [28], [29] are proposed in the literature.

The HFI hardware proposed by Kalali and Hamzaoglu [30] uses the Hcub multiplierless constant multiplication (MCM) algorithm. It calculates common sub-expressions in filter equations only once. Hence, the number and size of the adders, and adder tree depth are reduced. The HFI hardware proposed by Lung and Shen [31] uses a new data reuse technique and a highly parallel architecture to improve throughput. The HFI hardware proposed by Pastuszak and Trochimiuk [32] has 2 parallel datapaths for IP and fractional pixel (FP) motion estimation which share the same memories. The HFI hardware proposed by Diniz et al. [33] uses a reconfigurable datapath which can process different filter types.

The HFI hardware proposed by Mert et al. [38] uses memory-based constant multiplication. The multiplications of an input pixel with multiple constant coefficients of FIR filters are pre-calculated and stored in memory. A high-level synthesis (HLS) based hardware implementation of HFI is proposed by Sjövall et al. [39]. It has higher performance than manual HFI hardware implementation at the cost of much larger area.

The VFI hardware proposed by Mert et al. [29] implements 15 FIR filters in parallel. It calculates 255 FPs for every IP. The exact VFI hardware proposed by Mert et al. [29] uses Hcub MCM algorithm and calculates a common offset for all the equations of 15 FIR filters. It also calculates common sub-expressions once and uses their results in different equations. The coefficients of offset and filters in this hardware are shown in Table II. Since the coefficients of filters F_9 to F_{15} are symmetric with the coefficients of filters F_7 to F_1, they are not shown in the table.

The VFI hardware proposed by Azgin et al. [28] can only be used for motion compensation. It calculates 1 FP for every IP. The approximate VFI hardware proposed by Azgin et al. [25] and Mahdavi et al. [18] include 14 3-tap and one 4-tap FIR filters. They have less area and power consumption than the exact VFI hardware at the expense of a very small quality loss. The approximate VFI hardware proposed by Mahdavi et al. [18] uses common offset values and Hcub MCM algorithm.

Small coefficients of VFI FIR filters have less effect on the filter result. Due to spatial correlation, neighboring pixels have similar values. Two approximate VFI FIR filters, F1 and F2, are proposed by Mahdavi et al. [18]. Approximate VFI F1 FIR filters are proposed by assuming that the pixels multiplied with smaller coefficients are similar. Approximate VFI F2 FIR filters are proposed by substituting most of the coefficients in F1 with the closest 2^n values. Hence, most of the multiplications of F2 FIR filters are implemented using only shift operations.

MCMF1 and MCMF2 hardware are also proposed by Mahdavi et al. [18] for implementing the approximate VFI F1 and F2 FIR filters, respectively. Both MCMF1 and MCMF2 hardware use Hcub MCM algorithm and calculate 3 common offsets (O_1, O_2, O_3). The coefficients of offsets and filters in MCMF1 and MCMF2 hardware are shown in Table 2.5 and Table 2.7, respectively. In Table 2.7, F2 FIR filters $F2OF_4$, $F2OF_6$, $F2OF_9$, $F2OF_{11}$ are the same as $F2OF_5$, $F2OF_7$, $F2OF_{10}$, $F2OF_{12}$, respectively. Hence, in MCMF2 hardware, only $F2OF_4$, $F2OF_6$, $F2OF_9$, $F2OF_{11}$ are calculated, and their results are also used for $F2OF_5$, $F2OF_7$, $F2OF_{10}$, $F2OF_{12}$, respectively.

An HLS based hardware implementation of VFI is proposed by Hamzaoglu et al. [22]. It has higher performance than manual VFI hardware implementation at the expense of much larger area.

3.1 Fractional Interpolation FIR Filters

In HFI, one 8-tap and two 7-tap FIR filters are used. These 3 FIR filters type A, type B, type C are shown in equations (3.1), (3.2), and (3.3), respectively. The value of shift1 is determined based on bit depth of the pixel. Figure 3.1 shows IPs "$A_{x,y}$", HIPs "$a_{x,y}$, $b_{x,y}$, $c_{x,y}$", VIPs "$d_{x,y}$, $h_{x,y}$, $n_{x,y}$", and HVIPs "$e_{x,y}$, $f_{x,y}$, $g_{x,y}$, $i_{x,y}$, $j_{x,y}$, $k_{x,y}$, $p_{x,y}$, $q_{x,y}$, $r_{x,y}$" in a prediction unit (PU). The nearest IPs in horizontal direction are used for interpolating HIPs (a, b, c) and the nearest IPs in vertical direction are used for interpolating VIPs (d, h, n). The HVIPs are interpolated from the nearest HIPs.

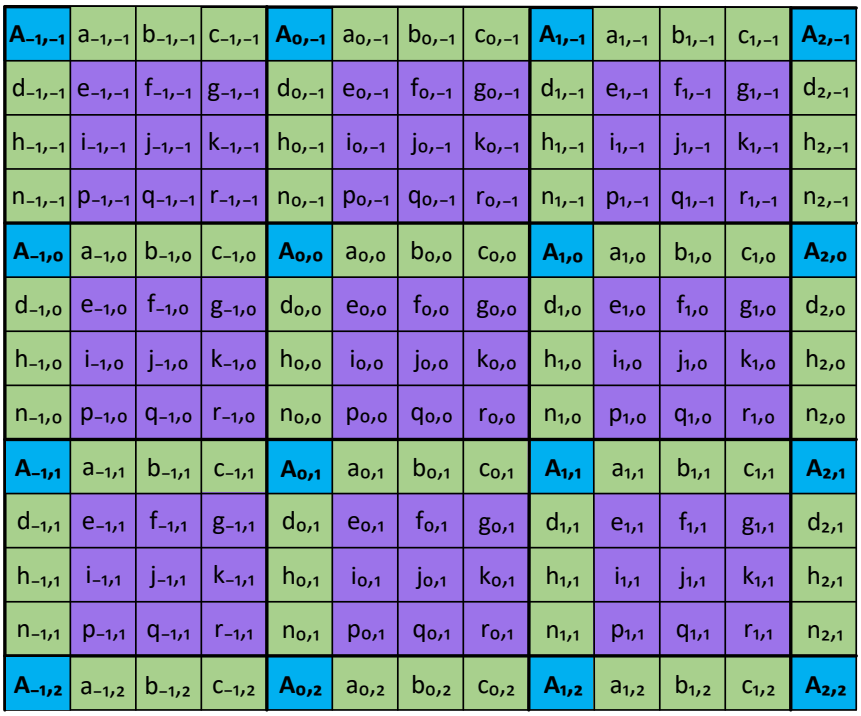

Figure 3.1 Integer pixels, HIPs, VIPs, HVIPs in HEVC FI.

In VFI, 8 7-tap and 7 8-tap FIR filters are used. Table 2.1 shows the coefficients of these FIR filters. In VFI, there are 15 HIPs between 2 neighboring horizontal IPs and 15 VIPs between 2 neighboring vertical IPs. The HIPs and VIPs are interpolated from nearest IPs using 15 FIR filters. The HVIPs are interpolated from nearest HIPs using 15 FIR filters.

3.2 Proposed HEVC FI Hardware

In this thesis, a novel technique is proposed, which reduces number of additions by decomposing coefficients of the FIR filters used for HEVC FI. Decomposition of the coefficients in type A, type B and type C FIR filters are shown in equations (3.9), (3.10), and (3.11), respectively where common sub-expressions are highlighted with different colors. Although the number of coefficients increases, more common sub-expressions are obtained which reduces the number of additions.

In the proposed HEVC FI hardware, 8 type A FIR filters, 8 type B FIR filters and 8 type C FIR filters are calculated in parallel in each clock cycle. Therefore, 24 fractional pixels in the same row or column are interpolated in each clock cycle using 15 integer pixels or 15 horizontal half-pixels in the same row or column.

For example, $a_{-3,0}, \dots, a_{0,0}, \dots, a_{4,0}$, $b_{-3,0}, \dots, b_{0,0}, \dots, b_{4,0}$, $c_{-3,0}, \dots, c_{0,0}, \dots, c_{4,0}$ horizontal half-pixels in row 0 are interpolated using $A_{-6,0}, \dots, A_{0,0}, \dots, A_{8,0}$ integer pixels in row 0. FIR filters used in HEVC standard to interpolate 8 of these 24 half-pixels are shown in equations (3.1)-(3.8).

$$a_{0,0} = \begin{pmatrix} -A_{-3,0} + 4A_{-2,0} - 10A_{-1,0} + 58A_{0,0} \\ +17A_{1,0} - 5A_{2,0} + A_{3,0} \end{pmatrix} \gg Shift1 \quad (3.1)$$

$$b_{0,0} = \begin{pmatrix} -A_{-3,0} + 4A_{-2,0} - 11A_{-1,0} + 40A_{0,0} \\ +40A_{1,0} - 11A_{2,0} + 4A_{3,0} - A_{4,0} \end{pmatrix} \gg Shift1 \quad (3.2)$$

$$c_{0,0} = \begin{pmatrix} A_{-2,0} - 5A_{-1,0} + 17A_{0,0} + 58A_{1,0} \\ -10A_{2,0} + 4A_{3,0} - A_{4,0} \end{pmatrix} \gg Shift1 \quad (3.3)$$

$$a_{1,0} = \begin{pmatrix} -A_{-2,0} + 4A_{-1,0} - 10A_{0,0} + 58A_{1,0} \\ +17A_{2,0} - 5A_{3,0} + A_{4,0} \end{pmatrix} \gg Shift1 \quad (3.4)$$

$$b_{1,0} = \begin{pmatrix} -A_{-2,0} + 4A_{-1,0} - 11A_{0,0} + 40A_{1,0} \\ +40A_{2,0} - 11A_{3,0} + 4A_{4,0} - A_{5,0} \end{pmatrix} \gg Shift1 \quad (3.5)$$

$$b_{-1,0} = \begin{pmatrix} -A_{-4,0} + 4A_{-3,0} - 11A_{-2,0} + 40A_{-1,0} \\ +40A_{0,0} - 11A_{1,0} + 4A_{2,0} - A_{3,0} \end{pmatrix} \gg Shift1 \quad (3.6)$$

$$c_{-1,0} = \begin{pmatrix} A_{-3,0} - 5A_{-2,0} + 17A_{-1,0} + 58A_{0,0} \\ -10A_{1,0} + 4A_{2,0} - A_{3,0} \end{pmatrix} \gg Shift1 \quad (3.7)$$

$$c_{2,0} = \begin{pmatrix} A_{0,0} - 5A_{1,0} + 17A_{2,0} + 58A_{3,0} \\ -10A_{4,0} + 4A_{5,0} - A_{6,0} \end{pmatrix} \gg Shift1 \quad (3.8)$$

The same FIR filters with decomposed coefficients are shown in equations (3.9)-(3.16).

$$a_{0,0} = \begin{pmatrix} -A_{-3,0} + 4A_{-2,0} - 10A_{-1,0} + 40A_{0,0} + 18A_{0,0} + 18A_{1,0} \\ -A_{1,0} - A_{2,0} - 4A_{2,0} + A_{3,0} \end{pmatrix} \gg Shift1 \quad (3.9)$$

$$b_{0,0} = \begin{pmatrix} -A_{-3,0} + 4A_{-2,0} - A_{-1,0} - 10A_{-1,0} + 40A_{0,0} \\ +40A_{1,0} - 10A_{2,0} - A_{2,0} + 4A_{3,0} - A_{4,0} \end{pmatrix} \gg Shift1 \quad (3.10)$$

$$c_{0,0} = \begin{pmatrix} A_{-2,0} - 4A_{-1,0} - A_{-1,0} - A_{0,0} + 18A_{0,0} + 18A_{1,0} \\ +40A_{1,0} - 10A_{2,0} + 4A_{3,0} - A_{4,0} \end{pmatrix} \gg Shift1 \quad (3.11)$$

$$a_{1,0} = \begin{pmatrix} -A_{-2,0} + 4A_{-1,0} - 10A_{0,0} + 40A_{1,0} + 18A_{1,0} + 18A_{2,0} \\ -A_{2,0} - A_{3,0} - 4A_{3,0} + A_{4,0} \end{pmatrix} \gg Shift1 \quad (3.12)$$

$$b_{1,0} = \begin{pmatrix} -A_{-2,0} + 4A_{-1,0} - A_{0,0} - 10A_{0,0} + 40A_{1,0} + 40A_{2,0} \\ -10A_{3,0} - A_{3,0} + 4A_{4,0} - A_{5,0} \end{pmatrix} \gg Shift1 \quad (3.13)$$

$$b_{-1,0} = \begin{pmatrix} -A_{-4,0} + 4A_{-3,0} - A_{-2,0} - 10A_{-2,0} + 40A_{-1,0} + 40A_{0,0} \\ -10A_{1,0} - A_{1,0} + 4A_{2,0} - A_{3,0} \end{pmatrix} \gg Shift1 \quad (3.14)$$

$$c_{-1,0} = \begin{pmatrix} A_{-3,0} - 4A_{-2,0} - A_{-2,0} - A_{-1,0} + 18A_{-1,0} + 18A_{0,0} \\ +40A_{0,0} - 10A_{1,0} + 4A_{2,0} - A_{3,0} \end{pmatrix} \gg Shift1 \quad (3.15)$$

$$c_{2,0} = \begin{pmatrix} A_{0,0} - 4A_{1,0} - A_{1,0} - A_{2,0} + 18A_{2,0} + 18A_{3,0} \\ +40A_{3,0} - 10A_{4,0} + 4A_{5,0} - A_{6,0} \end{pmatrix} \gg Shift1 \qquad (3.16)$$

The sub-expressions in equations (3.9)-(3.11) that are common with the sub-expressions in equations (3.9)-(3.16) are highlighted with different colors. Some common sub-expressions are negated versions of each other. For example, negated version of the sub-expression "$A_{-2,0} - 4A_{-1,0}$" in equation (3.11), i.e. "$-A_{-2,0} + 4A_{-1,0}$", exists in equations (3.12) and (3.13). Therefore, it can be calculated only once, and its result can be used in equations (3.12) and (3.13) by negating it.

The FIR filters in equations (3.9)-(3.16) with decomposed coefficients have more common sub-expressions than the FIR filters in equations (3.1)-(3.8) with original coefficients. There are even more common sub-expressions in all 24 FIR filters with decomposed coefficients which are used to interpolate 24 fractional pixels in parallel in each clock cycle. All these common sub-expressions and the number of adders used to implement them are shown in Table 3.1.

Table 3.1 Common Sub-Expressions in the Proposed HEVC FI Hardware

General form	Sub-expressions	Adders
$-A_{x-1,0}+4\times A_{x,0}$	$-A_{-6,0}+4\times A_{-5,0}$, $-A_{-5,0}+4\times A_{-4,0}$, … , $-A_{3,0}+4\times A_{4,0}$	10
$-10\times A_{x-1,0}+40\times A_{x,0}$	$-10\times A_{-4,0}+40\times A_{-3,0}$, $-10\times A_{-3,0}+40\times A_{-2,0}$, … , $-10\times A_{3,0}+40\times A_{4,0}$	8
$4\times A_{x-1,0}-A_{x,0}$	$4\times A_{-2,0}-A_{-1,0}$, $4\times A_{-1,0}-A_{0,0}$, … , $4\times A_{7,0}-A_{8,0}$	10
$40\times A_{x-1,0}-10\times A_{x,0}$	$40\times A_{-2,0}-10\times A_{-1,0}$, $40\times A_{-1,0}-10\times A_{0,0}$, … , $40\times A_{5,0}-10\times A_{6,0}$	8
$A_{x-1,0}+A_{x,0}$	$A_{-4,0}+A_{-3,0}$, $A_{-3,0}+A_{-2,0}$, … , $A_{5,0}+A_{6,0}$	10
$18\times A_{x-1,0}+18\times A_{x,0}$	$18\times A_{-3,0}+18\times A_{-2,0}$, $18\times A_{-2,0}+18\times A_{-1,0}$, …, $18\times A_{4,0}+18\times A_{5,0}$	8
Total adders		54

Figure 3.2 shows the proposed datapaths for implementing all sub-expressions including common sub-expressions shown in Table 3.1. Since the common sub-expressions are calculated only once, they reduce the number of adders used in the proposed HEVC FI hardware. In addition, as shown in Figure 3.2, first the sub-expressions shown in rows 1, 3, 5 in Table 3.1 are calculated. Then, using the results of these sub-expressions, the sub-expressions shown in rows 2, 4, 6 in Table 3.1 are calculated. This also reduces the number of adders used in the proposed hardware.

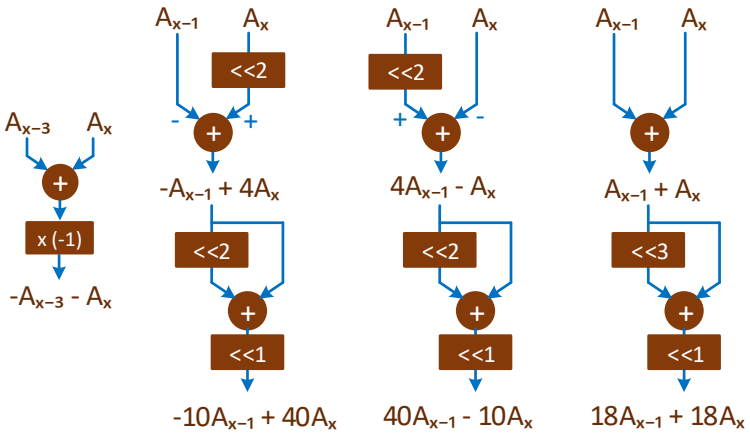

Figure 3.2 Sub-Expressions datapaths in the proposed HEVC FI hardware.

Adder trees are used to add the results of sub-expressions shown in Figure 3.2 for calculating FIR filters. There are also common sub-expressions in the adder trees. Common sub-expression "$-A_{-3,0} +4\times A_{-2,0} -10\times A_{-1,0} +40\times A_{0,0}$" exists in equations (3.9) and (3.10). Therefore, it is calculated only once, and its result is used in (3.9) and (3.10). One adder is used to calculate this sub-expression by adding the common sub-expressions "$-A_{-3,0} +4\times A_{-2,0}$" and "$-10\times A_{-1,0} +40\times A_{0,0}$". Common sub-expression "$40\times A_{1,0} -10\times A_{2,0} +4\times A_{3,0} -A_{4,0}$" exists in equations (3.10) and (3.11). Therefore, it is calculated only once, and its result is used in (3.10) and (3.11). One adder is used to calculate this sub-expression by adding the common sub-expressions "$40\times A_{1,0} -10\times A_{2,0}$" and "$4\times A_{3,0} -A_{4,0}$".

To calculate (3.9), three adders are used in adder trees to add the results of "$-A_{-3,0} +4\times A_{-2,0} -10\times A_{-1,0} +40\times A_{0,0}$", "$18\times A_{0,0} +18\times A_{1,0}$", "$-A_{1,0} -A_{2,0}$", and "$-4\times A_{2,0} +A_{3,0}$". To calculate (3.10), two adders are used in adder trees to add the results of "$-A_{-3,0} +4\times A_{-2,0} -10\times A_{-1,0} +40\times A_{0,0}$", "$-A_{-1,0} -A_{2,0}$", "$40\times A_{1,0} -10\times A_{2,0} +4\times A_{3,0} -A_{4,0}$". To calculate (3.11), three adders are used in adder trees to add the results of "$A_{-2,0} -4\times A_{-1,0}$", "$-A_{-1,0} -A_{0,0}$", "$18\times A_{0,0} +18\times A_{1,0}$", and "$40\times A_{1,0} -10\times A_{2,0} +4\times A_{3,0} -A_{4,0}$". Therefore, 10 adders are used in adder trees to interpolate 3 FPs. 80 adders are used in adder trees to interpolate 3×8 = 24 FPs in each CC.

Figure 3.3 shows the proposed HFI hardware for all PU sizes. The splitters represent interconnects in the proposed hardware. They are used to simplify the figure. In Figure 3.3, Sub-Expressions block represent the sub-expression datapaths shown in Figure 3.2.

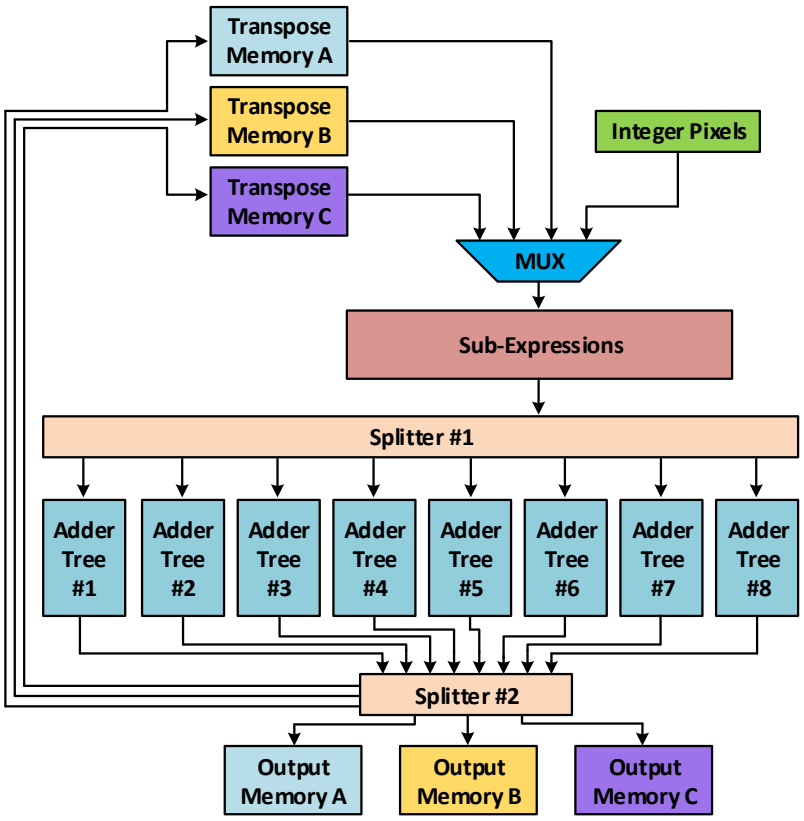

Figure 3.3 Proposed HEVC FI hardware.

The proposed hardware interpolates all fractional pixels for luma component of an 8×8 PU. The larger PU sizes are decomposed into 8×8 blocks and these 8×8 blocks are interpolated separately.

First, 8×15 horizontal a, b, c half-pixels are interpolated in 15 clock cycles, and they are stored into transpose memories A, B, C, respectively. Then, 8×8 vertical d, h, n half-pixels are interpolated in 8 clock cycles. Finally, 9×8×8 quarter-pixels are interpolated in 24 clock cycles using the half-pixels in transpose memories A, B, C. There are three pipeline stages in the proposed hardware. Therefore, all fractional pixels for an 8×8 PU are interpolated in 50 clock cycles.

3.3 Proposed VVC FI Hardware

In the proposed VFI hardware, FIR filter coefficients are decomposed to other coefficients in the forms of powers of 2 as shown in Table 3.2. A_{-3} to A_4 represent the input pixels. The proposed VFI hardware also uses the common offset proposed by Mert et al. [29].

Table 3.2 Decomposed Coefficients in Proposed VVC FI Hardware

	A_{-3}	A_{-2}	A_{-1}	A_0	A_1	A_2	A_3	A_4
Offset	-1	4	-8	32	32	-8	4	-1
F_1	1	1-4	4+1	-1+32	-32+4	2+4	-4+1	1
F_2	0	-2	2+1	-2+32	-32+8	1+4	-4+1	1
F_3	0	-1	0	32-4	-4-16+1	4	-4+1	1
F_4	0	0	-2	2+8+16	-16+1	-1+4	-4+1	1
F_5	0	0	1-4	16+4	-4-2	0	-1	0
F_6	0	-1	-1	16-1	-1	-2	0	0
F_7	0	0	-1-2	8+4+1	2	-2	0	0
F_8	0	0	-1-2	8	8	-2-1	0	0
F_9	0	0	-2	2	1+4+8	-2-1	0	0
F_{10}	0	0	-2	-1	-1+16	-1	-1	0
F_{11}	0	-1	0	-2-4	16+4	-4+1	0	0
F_{12}	1	1-4	4-1	1-16	16+8+2	-2	0	0
F_{13}	1	1-4	4	1-16-4	-4+32	0	-1	0
F_{14}	1	1-4	4+1	8-32	32-2	1+2	-2	0
F_{15}	1	1-4	4+2	4-32	32-1	1+4	-4+1	1

In the proposed VFI hardware, 8×15 FPs are interpolated in parallel in each CC using 15 IPs or 15 HIPs and 8 sets of 15 FIR filters with decomposed coefficients. There are more common sub-expressions in these 8×15 FIR filters.

Table 3.3 shows all the common sub-expressions in the proposed VFI hardware, and the number of adders used to implement them. In each row, for each common sub-expression general form, all specific sub-expressions, which are negated or shifted versions of each other, are shown. For example, in the second row, "$8\times A_{-2} -2\times A_{-1}$" and "$32\times A_{-2} -8\times A_{-1}$" are obtained by shifting "$4\times A_{-2} -A_{-1}$" 1 bit and 3 bits to the left, respectively.

Adder trees are used to add the results of sub-expressions for calculating the proposed VFI FIR filters. There are also common sub-expressions in the adder trees. Common sub-expression "$4\times A_2 -4\times A_3 +A_3 +A_4$", denoted by C_A, is calculated using 1 adder which adds the results of common sub-expressions "$4\times A_2 -4\times A_3$" and "$A_3 +A_4$". Common sub-expression "$A_{-3} +A_{-2} -4\times A_{-2} +4\times A_{-1}$", denoted by C_B, is calculated using 1 adder which adds the results of "$A_{-3} +A_{-2}$" and "$-4\times A_{-2} +4\times A_{-1}$". Common sub-expression "$-2\times A_{-1} +2\times A_0 +A_1 -A_2$", denoted by C_C, is calculated using 1 adder which adds the results of "$-2\times A_{-1} +2\times A_0$" and "$A_1 -A_2$". Common sub-expression "$-A_{-1} +A_0 +2\times A_1 -2\times A_2$", denoted by C_D, is calculated using 1 adder which adds the results of "$-A_{-1} +A_0$" and "$2\times A_1 -2\times A_2$". Common sub-expression "$-2\times A_{-1} +8\times A_0 +8\times A_1 -2\times A_2$", denoted by C_O, is calculated using 1 adder which adds the results of "$-2\times A_{-1} +8\times A_0$" and "$8\times A_1 -2\times A_2$".

Table 3.3 Common Sub-Expressions in the Proposed VVC FI Hardware

General form	Sub-expressions	Adders
$-A_{x-2}+4\times A_{x-1}$	$A_{-6}-4\times A_{-5}$, $A_{-5}-4\times A_{-4}$, … , $A_4-4\times A_5$ $-2\times A_{-4}+8\times A_{-3}$, $-2\times A_{-3}+8\times A_{-2}$, … , $-2\times A_4+8\times A_5$ $-4\times A_{-4}+16\times A_{-3}$, $-4\times A_{-3}+16\times A_{-2}$, … , $-4\times A_4+16\times A_5$ $-8\times A_{-4}+32\times A_{-3}$, $-8\times A_{-3}+32\times A_{-2}$, … , $-8\times A_3+32\times A_4$	11
$4\times A_{x-1}-A_x$	$4\times A_{-3}-A_{-2}$, $4\times A_{-2}-A_{-1}$, … , $4\times A_7-A_8$ $8\times A_{-3}-2\times A_{-2}$, $8\times A_{-2}-2\times A_{-1}$, …, $8\times A_5-2\times A_6$ $32\times A_{-2}-8\times A_{-1}$, $32\times A_{-2}-8\times A_{-1}$, …, $32\times A_5-8\times A_6$	11
$A_{x-2}+A_{x-1}$	$A_{-6}+A_{-5}$, $A_{-5}+A_{-4}$, … , A_7+A_8 $-4\times A_{-3}-4\times A_{-2}$, $-4\times A_{-2}-4\times A_{-1}$, … , $-4\times A_4-4\times A_5$	14
$A_{x-1}-A_x$	$A_{-2}-A_{-1}$, $A_{-1}-A_0$, … , A_5-A_6 $-A_{-4}+A_{-3}$, $-A_{-3}+A_{-2}$, … , $-A_3+A_4$ $-2\times A_{-5}+2\times A_{-4}$, $-2\times A_{-4}+2\times A_{-3}$, … , $-2\times A_3+2\times A_4$ $2\times A_{-2}-2\times A_{-1}$, $2\times A_{-1}-2\times A_0$, … , $2\times A_6-2\times A_7$ $-4\times A_{-5}+4\times A_{-4}$, $-4\times A_{-4}+4\times A_{-3}$, … , $-4\times A_2+4\times A_3$ $4\times A_{-3}-4\times A_{-2}$, $4\times A_{-2}-4\times A_{-1}$, … , $4\times A_6-4\times A_7$ $16\times A_{-3}-16\times A_{-2}$, $16\times A_{-2}-16\times A_{-1}$, … , $16\times A_4-16\times A_5$ $-16\times A_{-3}+16\times A_{-2}$, $-16\times A_{-2}+16\times A_{-1}$, … , $-16\times A_4+16\times A_5$ $32\times A_{-3}-32\times A_{-2}$, $32\times A_{-2}-32\times A_{-1}$, … , $32\times A_4-32\times A_5$ $-32\times A_{-3}+32\times A_{-2}$, $-32\times A_{-2}+32\times A_{-1}$, … , $-32\times A_4+32\times A_5$	12
$A_{x-3}+A_x$	$A_{-4}+A_{-1}$, $A_{-3}+A_0$, … , A_3+A_6 $-A_{-4}-A_{-1}$, $-A_{-3}-A_0$, … , $-A_3-A_6$	8
$A_{x-3}-A_x$	$A_{-3}-A_0$, $A_{-2}-A_1$, … , A_4-A_7 $-A_{-5}+A_{-2}$, $-A_{-4}+A_{-1}$, … , $-A_2+A_5$	10
$A_{x-4}-A_x$	$A_{-4}-A_0$, $A_{-3}-A_1$, … , A_3-A_7 $-A_{-5}+A_{-1}$, $-A_{-4}+A_0$, … , $-A_2+A_6$	9
$-A_{x-3}-A_{x-2}-A_{x-1}-A_x$	$-A_{-5}-A_{-4}-A_{-3}-A_{-2}$, $-A_{-4}-A_{-3}-A_{-2}-A_{-1}$, … , $-A_4-A_5-A_6-A_7$	10
$-4\times A_{x-2}+16\times A_{x-1}+4\times A_{x-1}-4\times A_x$	$-4\times A_{-4}+16\times A_{-3}+4\times A_{-3}-4\times A_{-2}$, … , $-4\times A_4+16\times A_5+4\times A_5-4\times A_6$	9
Total adders		94

Offset is calculated using 2 adders which add "$C_O<<2$", "$-A_3+4\times A_{-2}$", and "$4\times A_3-A_4$". Common sub-expression denoted by C_E is calculated using 1 adder which adds C_A and Offset. Common sub-expression denoted by C_F is calculated using 1 adder which adds C_B and Offset. Common sub-expression denoted by C_G is calculated using 2 adders which add C_A, C_B, and Offset. Common sub-expressions C_A, C_B, C_C, C_D, C_O, C_E, C_F, C_G are calculated only once and their results are used in different FIR filters. For example, C_G is used in FIR filters F_1 and F_{15}. To calculate F_1, 4 adders are used in adder trees which add C_G, "$-A_0+4\times A_1$", "$32\times A_0-32\times A_1$", A_{-1}, "$A_2<<1$". Similarly, FIR filters F_2, F_3, F_4, F_5, F_6, F_7, F_8, F_9, F_{10}, F_{11}, F_{12}, F_{13}, F_{14}, F_{15} are calculated using 4, 4, 3, 3, 3, 3, 2, 3, 3, 3, 3, 4, 4, 4 adders, respectively. Therefore, 61 adders are used in adder trees for calculating Offset and FIR filters to interpolate 15 FPs.

Figure 3.4 shows the proposed VFI hardware for all PU sizes. The splitters represent interconnects in the proposed hardware. TR MEM and OUT MEM are the

transpose memories and output memories, respectively. Sub-Expressions block calculates all common sub-expressions shown in Table 3.3.

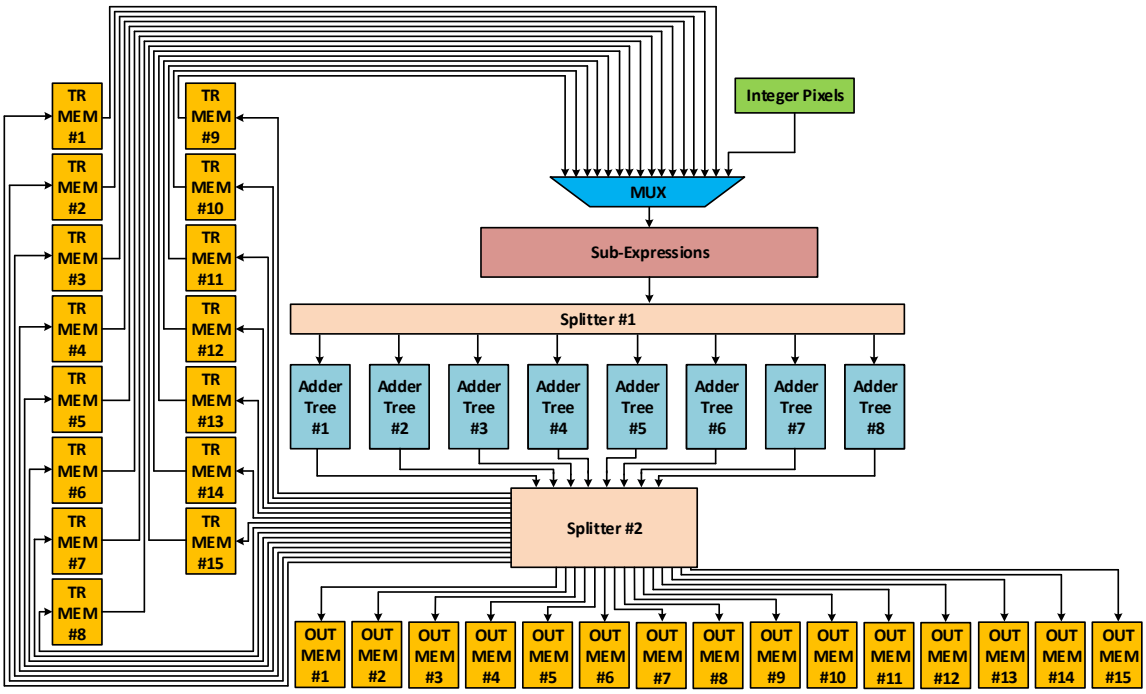

Figure 3.4 Proposed VVC FI Hardware

Figure 3.5 shows the proposed datapaths used in the Sub-Expressions block. Common sub-expressions "$-A_{x-3} - A_{x-2} - A_{x-1} - A_x$" and "$-4\times A_{x-2} + 16\times A_{x-1} + 4\times A_{x-1} - 4\times A_x$" are calculated using other common sub-expressions. Hence, only 1 extra adder is used for implementing each of them.

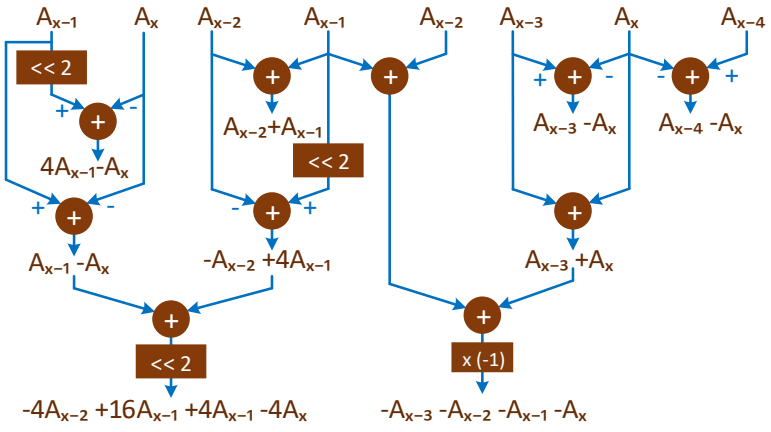

Figure 3.5 Common sub-expression datapaths in the proposed VVC FI hardware.

8×15×15 HIPs are interpolated in 15 CCs and stored in transpose memories. 8×8×15 VIPs are interpolated in 8 CCs. 8×8×225 HVIPs are interpolated in 8×15 CCs using HIPs. There are 4 pipeline stages in the proposed hardware. Hence, all the FPs for an 8×8 PU are interpolated in 147 CCs.

3.4 Proposed Approximate VFI Hardware DCF1

In the proposed hardware, the coefficients of F1 FIR filters are decomposed into other coefficients in the forms of powers of 2 as shown in Table 3.4. Hence, it is called Decomposed Coefficients of F1 (DCF1) hardware. The coefficients of common sub-expressions are shown with the same color in the table.

Table 3.4 Approximate F1 FIR Filters with Offset Used in DCF1

		A_{-1}	A_0	A_1	A_2	Required Offset
Offsets	Off_1	-8	64	8	0	
	Off_2	0	8	64	-8	
	Off_3	-8	8	8	-8	
FIR Filters with Offset Used in DCF1	$F1OF_1$	2+4	-1	-1-4	0	Off_1
	$F1OF_2$	4	-2	-2	0	Off_1
	$F1OF_3$	2	-2-2	2	0	Off_1
	$F1OF_4$	1	-1-1-4	4+1	0	Off_1
	$F1OF_5$	0	-8-4	4+8	0	Off_1
	$F1OF_6$	1	-1-16	16	0	Off_1
	$F1OF_7$	0	-1-2-16	16+2+1	0	Off_1
	$F1OF_8$	0	32	32	0	Off_3
	$F1OF_9$	0	1+2+16	-16-2-1	0	Off_2
	$F1OF_{10}$	0	16	-16-1	1	Off_2
	$F1OF_{11}$	0	4+8	-8-4	0	Off_2
	$F1OF_{12}$	0	1+4	-4-1-1	1	Off_2
	$F1OF_{13}$	0	2	-2-2	2	Off_2
	$F1OF_{14}$	0	-2	-2	4	Off_2
	$F1OF_{15}$	0	-4-1	-1	4+2	Off_2

In DCF1 hardware, 8×15 FPs are interpolated in parallel in each CC using 15 IPs or 15 HIPs and 8 sets of 15 F1 FIR filters with decomposed coefficients. There are more common sub-expressions in these 8×15 FIR filters. Table 3.5 shows all the common sub-expressions in DCF1 hardware, and the number of adders used to implement them.

Block diagram of DCF1 hardware is similar to the block diagram of the proposed VFI hardware shown in Figure 3.4. Their Sub-Expressions and Adder Tree blocks are different. Figure 3.6 shows the proposed datapaths used in the Sub-Expressions block of DCF1 hardware. Some common sub-expressions are calculated using the results of other common sub-expressions.

Table 3.5 Common Sub-Expressions in DCF1 Hardware

General form	Sub-expressions	Adders
$-A_{x-1}+A_x$	$-A_{-3}+A_{-2}$, $-A_{-2}+A_{-1}$, … , $-A_5+A_6$ $-2\times A_{-3}+2\times A_{-2}$, $-2\times A_{-2}+2\times A_{-1}$, … , $-2\times A_5+2\times A_6$ $-4\times A_{-3}+4\times A_{-2}$, $-4\times A_{-2}+4\times A_{-1}$, … , $-4\times A_4+4\times A_5$ $-8\times A_{-3}+8\times A_{-2}$, $-8\times A_{-2}+8\times A_{-1}$, … , $-8\times A_4+8\times A_5$ $-16\times A_{-3}+16\times A_{-2}$, $-16\times A_{-2}+16\times A_{-1}$, … , $-16\times A_4+16\times A_5$ $A_{-4}-A_{-3}$, $A_{-3}-A_{-2}$, … , A_4-A_5 $2\times A_{-4}-2\times A_{-3}$, $2\times A_{-3}-2\times A_{-2}$, … , $2\times A_4-2\times A_5$ $4\times A_{-3}-4\times A_{-2}$, $4\times A_{-2}-4\times A_{-1}$, … , $4\times A_4-4\times A_5$ $8\times A_{-3}-8\times A_{-2}$, $8\times A_{-2}-8\times A_{-1}$, … , $8\times A_4-8\times A_5$ $16\times A_{-3}-16\times A_{-2}$, $16\times A_{-2}-16\times A_{-1}$, … , $16\times A_4-16\times A_5$	10
$-A_{x-2}+A_x$	$-4\times A_{-3}+4\times A_{-1}$, $-4\times A_{-2}+4\times A_0$, … , $-4\times A_4+4\times A_6$ $-8\times A_{-4}+8\times A_{-2}$, $-8\times A_{-3}+8\times A_{-1}$, … , $-8\times A_3+8\times A_5$ $4\times A_{-4}-4\times A_{-2}$, $4\times A_{-3}-4\times A_{-1}$, … , $4\times A_3-4\times A_5$ $8\times A_{-3}-8\times A_{-1}$, $8\times A_{-2}-8\times A_0$, … , $8\times A_4-8\times A_6$	9
$A_{x-1}+A_x$	$-A_{-3}-A_{-2}$, $-A_{-2}-A_{-1}$, … , $-A_4-A_5$ $-2\times A_{-3}-2\times A_{-2}$, $-2\times A_{-2}-2\times A_{-1}$, … , $-2\times A_4-2\times A_5$ $32\times A_{-3}+32\times A_{-2}$, $32\times A_{-2}+32\times A_{-1}$, … , $32\times A_4+32\times A_5$	8
$A_{x-2}\ -A_{x-1}\ -A_{x-1}+A_x$	$A_{-4}\ -A_{-3}\ -A_{-3}+A_{-2}$, $A_{-3}\ -A_{-2}\ -A_{-1}+A_0$, … , $A_3\ -A_4\ -A_5\ +A_6$	9
$2\times A_{x-2}-A_{x-1}-A_x$	$2\times A_{-4}-A_{-3}-A_{-2}$, $2\times A_{-3}-A_{-2}-A_{-1}$, … , $2\times A_3-A_4-A_5$ $4\times A_{-4}-2\times A_{-3}-2\times A_{-2}$, $4\times A_{-3}-2\times A_{-2}-2\times A_{-1}$,… , $4\times A_3-2\times A_4-2\times A_5$	8
$-A_{x-2}-A_{x-1}+2\times A_x$	$-A_{-3}-A_{-2}+\ 2\times A_{-1}$, $-A_{-2}-A_{-1}+\ 2\times A_0$, … , $-A_4-A_5+\ 2\times A_6$ $-2\times A_{-3}-2\times A_{-2}+4\times A_{-1}$, $-2\times A_{-2}-2\times A_{-1}+\ 4\times A_0$,…, $-2\times A_4-2\times A_5+\ 4\times A_6$	8
$2\times A_{x-1}+A_{x-1}-2\times A_x-A_x$	$2\times A_{-3}+A_{-3}-2\times A_{-2}-A_{-2}$, … , $2\times A_4+A_4-2\times A_5-A_5$ $-2\times A_{-3}-A_{-3}+2\times A_{-2}+A_{-2}$, … , $-2\times A_4-A_4+2\times A_5+A_5$ $8\times A_{-3}+4\times A_{-3}-8\times A_{-2}-4\times A_{-2}$, … , $8\times A_4+4\times A_4-8\times A_5-4\times A_5$ $-8\times A_{-3}-4\times A_{-3}+8\times A_{-2}+4\times A_{-2}$, … , $-8\times A_4-4\times A_4+8\times A_5+4\times A_5$	8
$16\times A_{x-1}+2\times A_{x-1}+A_{x-1}-16\times A_x-2\times A_x-A_x$	$16\times A_{-3}+2\times A_{-3}+A_{-3}-16\times A_{-2}-2\times A_{-2}-A_{-2}$, … , $16\times A_4+2\times A_4+A_4-16\times A_5-2\times A_5-A_5$	8
Total adders		68

Interpolation order and number of pipeline stages in DCF1 hardware are the same as the proposed VFI hardware. Hence, DCF1 hardware interpolates all the FPs for an 8×8 PU in 147 CCs.

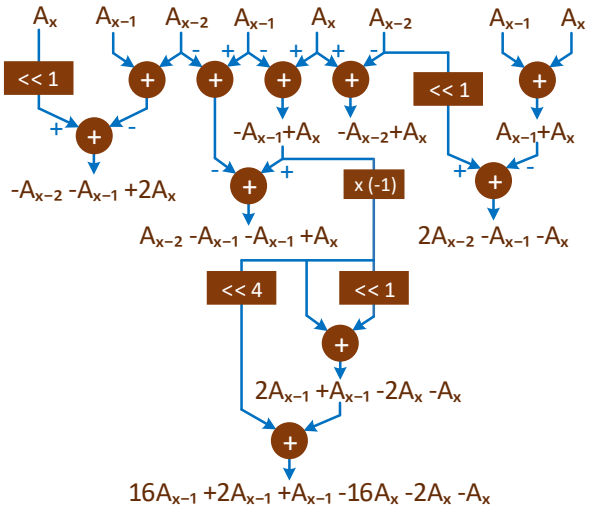

Figure 3.6 Common sub-expression datapaths in DCF1 hardware.

3.5 Proposed Approximate VFI Hardware DCF2

In the proposed hardware, the coefficients of F2 FIR filters are decomposed into other coefficients in the forms of powers of 2 as shown in Table 3.6. Hence, it is called Decomposed Coefficients of F2 (DCF2) hardware.

Table 3.6 Approximate F2 FIR Filters with Offset Used in DCF2

		A_{-1}	A_0	A_1	A_2	Required Offset	Required Final Shift
Offsets	Off_1	-1	8	1	0		
	Off_2	0	1	8	-1		
	Off_3	-1	1	1	-1		
FIR Filters with Offset Used in DCF2	$F2OF_1$	2+1	0	-2-1	0	Off_1 << 2	>>5
	$F2OF_2$	1	0	-1	0	Off_1 << 1	>>4
	$F2OF_3$	0	0	0	0	Off_1	>>3
	$F2OF_4$	0	-1	1	0	Off_1	>>3
	$F2OF_6$	0	-2-1	2+1	0	Off_1	>>3
	$F2OF_8$	0	4	4	0	Off_3	>>3
	$F2OF_9$	0	2+1	-2-1	0	Off_2	>>3
	$F2OF_{11}$	0	1	-1	0	Off_2	>>3
	$F2OF_{13}$	0	0	0	0	Off_2	>>3
	$F2OF_{14}$	0	-1	0	1	Off_2 << 1	>>4
	$F2OF_{15}$	0	-2-1	0	2+1	Off_2 << 2	>>5

The coefficients are decomposed in such a way that the adder sizes are also reduced. For example, for $F2OF_4$ FIR filter, instead of implementing $[(8\times A_1 - 8\times A_0) + O_1] >> 6$ in Table 2.7 where $O_1 = -8\times A_{-1} + 64\times A_0 + 8\times A_1$, we implement $[(A_1 - A_0) + Off_1] >> 3$ where $Off_1 = -A_{-1} + 8\times A_0 + A_1$.

F2 FIR filters $F2OF_4$, $F2OF_6$, $F2OF_9$, $F2OF_{11}$ are the same as $F2OF_5$, $F2OF_7$, $F2OF_{10}$, $F2OF_{12}$, respectively. Hence, in DCF2 hardware, only FIR filters $F2OF_4$, $F2OF_6$, $F2OF_9$, $F2OF_{11}$ are calculated, and their results are also used for $F2OF_5$, $F2OF_7$, $F2OF_{10}$, $F2OF_{12}$, respectively.

In DCF2 hardware, 8×15 FPs are interpolated in parallel in each CC using 15 IPs or 15 HIPs and 8 sets of 11 F2 FIR filters with decomposed coefficients. There are more common sub-expressions in these 8×11 FIR filters. Table 3.7 shows all the common sub-expressions in DCF2 hardware, and the number of adders used to implement them.

Table 3.7 Common Sub-Expressions in DCF2 Hardware

General form	Sub-expressions	Adders
$A_{x-2}-A_x$	$A_{-4}-A_{-2}$, $A_{-3}-A_{-1}$, … , A_4-A_6 $2\times A_{-4}-2\times A_{-2}$, $2\times A_{-3}-2\times A_{-1}$, … , $2\times A_4-2\times A_6$ $-A_{-4}+A_{-2}$, $-A_{-3}+A_{-1}$, … , $-A_4+A_6$	9
$-A_{x-1}+A_x$	$-A_{-3}+A_{-2}$, $-A_{-2}+A_{-1}$, … , $-A_4+A_5$ $-2\times A_{-3}+2\times A_{-2}$, $-2\times A_{-2}+2\times A_{-1}$, … , $-2\times A_4+2\times A_5$ $A_{-3}-A_{-2}$, $A_{-2}-A_{-1}$, … , A_4-A_5 $2\times A_{-3}-2\times A_{-2}$, $2\times A_{-2}-2\times A_{-1}$, … , $2\times A_4-2\times A_5$	8
$-2\times A_{x-1}-A_{x-1}+2\times A_x+A_x$	$-2\times A_{-3}-A_{-3}+2\times A_{-2}+A_{-2}$, … , $-2\times A_4-A_4+2\times A_5+A_5$ $2\times A_{-3}+A_{-3}-2\times A_{-2}-A_{-2}$, … , $2\times A_4+A_4-2\times A_5-A_5$	8
$-2\times A_{x-2}-A_{x-2}+2\times A_x+A_x$	$-2\times A_{-4}-A_{-4}+2\times A_{-2}+A_{-2}$, … , $-2\times A_4-A_4+2\times A_6+A_6$ $2\times A_{-4}+A_{-4}-2\times A_{-2}-A_{-2}$, … , $2\times A_4+A_4-2\times A_6-A_6$	9
Total adders		34

Block diagram of DCF2 hardware is similar to the block diagram of the proposed VFI hardware shown in Figure 3.4. Their Sub-Expressions block, Adder Tree block, and the numbers of OUT MEMs are different. DCF2 hardware uses 11 OUT MEMs because the F2 FIR filters $F2OF_5$, $F2OF_7$, $F2OF_{10}$, $F2OF_{12}$ are the same as other F2 FIR filters so their results are not calculated and stored. Figure 3.7 shows the proposed datapaths used in the Sub-Expressions block of DCF2 hardware. Some common sub-expressions are calculated using the results of other common sub-expressions.

Interpolation order and number of pipeline stages in DCF2 are the same as the proposed VFI hardware. Hence, DCF2 hardware interpolates all the FPs for an 8×8 PU in 147 CCs.

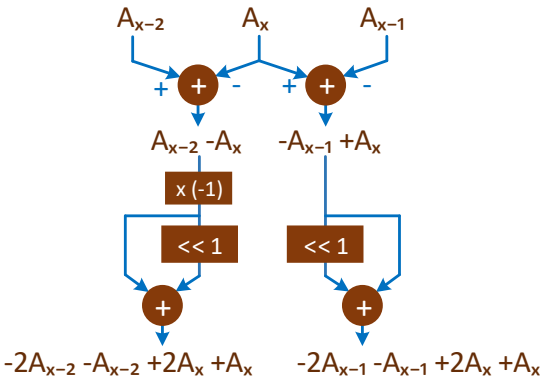

Figure 3.7 Common sub-expression datapaths in DCF2 hardware.

3.6 Comparison of Number of Adders

Table 3.8 shows the number of adders used in the proposed HFI and VFI hardware and the best HFI and VFI hardware in the literature.

Table 3.8 Number of Adders in HFI and VFI Hardware

		Number of Adders	Reduction
HFI	Kalali and Hamzaoglu [30]	176	-
	Proposed HEVC FI	142	19.3%
Exact VFI	Mert et al. [29]	633	-
	Proposed Exact VVC FI	582	8.0%
Approximate VFI	Mahdavi et al. MCMF1 [18]	341	-
	Proposed DCF1	260	23.7%
	Mahdavi et al. MCMF2 [18]	158	-
	Proposed DCF2	138	12.6%

The proposed HFI hardware uses 54 adders to calculate common sub-expressions as shown in Table 3.1. It uses 8 adders to calculate the 8 sub-expressions that are not common in the FIR filters such as "$-A_{-1,0}$ $-A_{2,0}$" in (3.10), and it uses 80 adders in adder trees. Hence, it uses 142 adders. HFI hardware proposed by Kalali and Hamzaoglu [30] uses 176 adders. Hence, the proposed HFI hardware uses 19.3% less adders than the one proposed by Kalali and Hamzaoglu [30].

In the proposed VFI hardware, to interpolate 8×15 FPs, in addition to the 94 adders used to calculate common sub-expressions shown in Table 3.3, 8×61 = 488 adders are used in adder trees. Hence, it uses 94 + 488 = 582 adders. The VFI hardware proposed by Mert *et al.* [29] uses 633 adders; 69 adders in MCM blocks, 34 adders for realizing the common sub-expressions, 42 adders for common offsets, and 488 adders

for adder trees. Hence, the proposed VFI hardware uses 8% less adders than the one proposed by Mert *et al*. [29].

DCF1 hardware uses 68 adders to calculate common sub-expressions as shown in Table 3.5. To interpolate 15 FPs, 24 adders are used in adder trees for calculating offsets and FIR filters. Hence, to interpolate 8×15 FPs, DCF1 hardware uses 68 + 8×24 = 260 adders. The MCMF1 hardware proposed by Mahdavi *et al*. [18] uses 341 adders; 38 adders in MCM blocks, 62 adders for common sub-expressions, 33 adders to calculate the offsets, and 208 adders in adder trees. Hence, DCF1 uses 23.7% less adders than MCMF1.

DCF2 hardware uses 34 adders to calculate common sub-expressions as shown in Table 3.7. To interpolate 15 FPs, 13 adders are used in adder trees for calculating offsets and FIR filters. Hence, to interpolate 8×15 FPs, DCF2 hardware uses 34 + 8×13 = 138 adders. The MCMF2 hardware proposed by Mahdavi *et al*. [18] uses 158 adders; 11 adders in MCM blocks, 42 adders for common sub-expressions, 33 adders to calculate the offsets, and 72 adders in adder trees. Hence, DCF2 uses 12.6% less adders than MCMF2.

3.7 Implementation Results

All the proposed HFI and VFI hardware are implemented using Verilog HDL. In this thesis, original HFI hardware and VFI hardware are also designed using adders and shifters for comparison, and they are implemented using Verilog HDL. Verilog RTL codes of all the proposed and original FI hardware are synthesized, placed and routed to a 28 nm FPGA. To have a fair comparison, Verilog RTL codes of the hardware proposed by Kalali and Hamzaoglu [30], Mert et al. [38], Mert et al. [29], and Mahdavi et al. [18] are synthesized, placed and routed to the same 28 nm FPGA. The FPGA implementations are verified with post place and route simulations.

The implementation results of HFI hardware are shown in Table 3.9. The power consumption results are shown in Table 3.9 and Table 3.10. The results shown as "---" have not been reported in the corresponding papers. The proposed HFI hardware has less area and less power consumption than the HFI hardware in the literature.

The proposed HFI hardware has higher performance than the manual HFI hardware implementations proposed in [25], [28], [29], [32], and [38]. Although the HFI HLS implementation proposed by Lung and Shen [29] and Sjövall et al. [39] have

higher performance than the proposed HFI hardware, they use more than 10 times LUTs.

Table 3.9 Implementation Results of HEVC FI Hardware

	Original Hardware	[30]	[31]	[32]	[33]	[38]	[39]	Proposed HEVC FI
FPGA (nm)	40	28	65	40	65	28	40	28
Slices	1669	1349	---	---	---	1370	---	1196
FFs	3448	3892	---	---	2550	3909	---	3747
LUTs	4110	2863	28486	26944	5017	3345	27100	2510
36K BRAM	3	3	---	---	2	3.5	---	3
Max. Freq. (MHz)	200	230	120	200	283	244	313	323
fps	30 QFHD	36 QFHD	---	60 FHD	30 2560×1600	37 QFHD	99 QFHD	50 QFHD
Throughput (M FPs/Second)	3840	4478	18720	1866	1843	4603	12317	6220
Power (mW)	152	196	---	171	89	210	---	83

Table 3.10 Power Consumption of HEVC FI Hardware (mW)

	[30]		[38]		Proposed	
Video	T	K	T	K	T	K
Clock	11	11	9	9	11	11
Signal	143	226	144	225	35	51
Logic	26	39	25	37	21	31
BRAM	16	16	32	33	16	16
Total	196	292	210	304	83	109

The Verilog RTL code of the proposed HFI hardware is also synthesized using 90 nm standard cell library. The gate count of the resulting ASIC implementation is calculated as 52278, including on-chip memories, based on a 2-input NAND gate area. The power consumption result is reported by the synthesis tool. Table 3.11 shows the ASIC implementation results of HFI hardware. The proposed HFI hardware is more efficient than the existing HFI hardware.

Table 3.11 ASIC Implementation Results of HEVC FI Hardware

	[32]	[30]	[38]	Proposed HEVC FI
Technology (nm)	90	90	90	90
Gate Count	265458	55738	57457	52278
Max. Freq. (MHz)	400	120	117	153
fps (Frames/Second)	30 QFHD	74 FHD	72 FHD	94 FHD
Throughput (M FPs/Second)	3732	2301	2239	2936
Power Consumption (mW)	30.2	20.5	21.6	23.1

The implementation and power consumption results of VFI hardware on the same 28 nm FPGA are shown in Table 3.12 and Table 3.13, respectively. Power consumptions of all FPGA implementations are estimated using a gate level power estimation tool for 1 frame of 1920×1080 Tennis (T) video and 1920×1080 Kimono (K) video at 100 MHz [35].

Table 3.12 Implementation Results of VVC FI Hardware

	Original Hardware	[29]	[22]	Proposed VVC FI	MCMF1 [18]	Proposed DCF1	MCMF2 [18]	Proposed DCF2
Exact (E) /Approx. (A)	E	E	E	E	A	A	A	A
Slices	5205	3121	15319	2970	2047	1934	2001	1851
FFs	6408	3589	37450	4167	3394	3089	2326	2272
LUTs	16334	10731	39047	9951	7112	6467	6725	6493
36K BRAM	30	30	30	30	30	30	26	26
Freq. (MHz)	208	219	150	230	237.5	237.5	246.9	249.3
Clock Cycles (8×8 PU)	147	147	74	147	147	147	147	147
fps	43 FHD	46 FHD	62 FHD	48 FHD	49 FHD	49 FHD	51 FHD	52 FHD
Throughput (M FPs/Sec)	23092	24323	32783	25380	25909	25909	26967	27495

Table 3.13 Power Consumption of VVC FI Hardware (mW)

	[29]		Proposed VFI		MCMF1 [18]		Proposed DCF1		MCMF2 [18]		Proposed DCF2	
Video	T	K	T	K	T	K	T	K	T	K	T	K
Clock	25	25	27	27	22	22	20	20	17	16	17	17
Signal	172	238	174	238	60	83	53	73	65	87	59	76
Logic	203	288	185	253	58	82	48	65	56	77	47	62
BRAM	137	138	137	138	134	138	134	138	111	114	111	114
Total	537	689	523	656	274	325	255	296	249	294	234	269

The proposed exact VFI hardware has higher performance, less area, and up to 4.78% less power consumption than the best exact manual VFI hardware proposed by Mert et al. [29]. Although the VFI HLS implementation proposed by Hamzaoglu et al. [22] has higher performance than the proposed VFI hardware, its area is 4 times larger.

The proposed approximate VFI hardware DCF1 and DCF2 have the same performance, less area, and up to 8.92% and 8.50% less power consumption than the approximate VFI hardware MCMF1 and MCMF2 proposed by Mahdavi et al. [18], respectively. The rate-distortion performance of DCF1 hardware and DCF2 hardware are the same as the rate-distortion performance of MCMF1 hardware and MCMF2 hardware, respectively.

CHAPTER IV

A NOVEL APPROXIMATE HIGH EFFICIENCY VIDEO CODING DCT HARDWARE

Both the HEVC and H.264 standards use discrete cosine transform (DCT) / inverse discrete cosine transform (IDCT). H.264 standard utilizes only 4×4 and 8×8 transform unit (TU) sizes for DCT/IDCT. HEVC standard utilizes 4×4, 8×8, 16×16, and 32×32 TU sizes for DCT/IDCT. Larger TU sizes achieve better energy compaction. However, they exponentially increase computational complexity. Moreover, HEVC standard exploits discrete sine transform (DST) / inverse discrete sine transform (IDST) for 4×4 intra prediction in particular cases.

DCT and DST have high computational complexity. DCT and DST account for 11% of the computational complexity of an HEVC video encoder. They account for 25% of the computational complexity of an all intra HEVC video encoder.

Approximate computing enables designing faster, smaller area and lower power consuming hardware compared to accurate hardware by trading off speed, area, and power consumption with quality [18], [40]-[44]. Hence, it is used for error tolerant applications with high computational complexity. Various approximate circuits are proposed in the literature [45]-[49]. Several approximate adders and multipliers are also proposed in the literature [21], [50]-[52].

This chapter of the thesis is an extended version of [21], where a new approximate constant multiplication technique is proposed to implement the constant

multiplication in HEVC DCT. In [21], all the multiplications in HEVC DCT are implemented by using the proposed approximate constant multiplication hardware. In this thesis, the approximate constant multiplication is used for multiplication with only the DCT coefficients which do not cause high average percentage error. In the proposed approximate HEVC DCT hardware, there are some common constant multiplications that are calculated once and their results are used in multiple DCT equations. Hcub multiplierless constant multiplication (MCM) technique [26] is utilized to implement constant multiplications in the proposed hardware.

Two instances of one dimensional (1D) DCT are used in [53] to explore the 2D DCT using its separability property for proposing a low-cost and high-throughput HEVC 16×16 2D DCT hardware. The variable-size HEVC 2D DCT hardware proposed in [54] allows multiple DCT sizes to share and reuse hardware resources. The HEVC 2D DCT hardware proposed in [55] uses the maximum circuit reuse during computation. In [56], a new CORDIC-based DCT hardware is proposed using matrix decomposition, resource sharing and merging.

In [12], a computation and energy reduction technique for HEVC DCT and a low energy HEVC 2D DCT hardware are proposed. This technique decreases the computational complexity of HEVC DCT at the cost of a reduction in peak-signal-to-noise-ratio (PSNR) and increase in bitrate by calculating only some of the pre-determined low frequency coefficients of TUs and assuming that the rest are zero.

Several approximate HEVC DCT hardware are proposed in the literature [57]-[59], [14]. In [57], multiplierless 4-point DCT implementations are proposed to be used in an approximate HEVC DCT hardware. These implementations include approximate adders and subtractors which are made using cartesian genetic programming. In [58], a flexible HEVC 2D DCT implementation is proposed, which can calculate 4 distinct approximations ranging from the complete DCT to the Walsh-Hadamard transform. This is done by selectively skipping some rotations. In [59], an 8×8 orthogonal approximation of HEVC DCT is proposed and used to obtain approximate transforms for other TU sizes. This approximation method exploits the neighboring pixels correlation in images such that the odd basis vectors of the DCT kernel are quantized by considering their signs and positions rather than their values. In [14], an algorithm to compute the necessary minimum number of low-frequency DCT-output/IDCT-input coefficients for 4, 8, 16, and 32-point DCT/IDCT in HEVC is proposed. It causes a slight reduction in PSNR and increase in bitrate. A flexible transpose memory

architecture supporting all the HEVC TU sizes and an efficient 2D DCT/IDCT hardware are proposed in [14].

4.1 Approximate Constant Multiplier and Approximate HEVC DCT Hardware [21]

An approximate constant multiplication technique is proposed in [21]. It achieves reduction in complexity of constant multiplication by manipulating variable multiplicand and constant multiplier. The constant multiplication is converted to a multiplication with a smaller constant, concatenation, and constant shift operation.

Multiplication of an m bit variable M with n bit constant N is shown in equation (4.1). Constant multiplier (N) is manipulated as shown in equation (4.2). Any constant integer N can be written as shown in (4.2). The proposed technique uses the values y and z that minimize NN. Most significant bits (MSBs) and least significant bits (LSBs) of variable multiplicand (M) are split as shown in equation (4.3) using the z value obtained from equation (4.2). Then, manipulated versions of M and N are multiplied as shown in equations (4.4)-(4.9). Equation (4.9) implements exact constant multiplication. The symbols "×", "«", and "{,}" represent multiplication, left shift, and concatenation operations, respectively.

$$R = M \times N \tag{4.1}$$

$$N = 2^{y} \times (1 + 2^{z} \times NN) \tag{4.2}$$

$$M = 2^{z} \times M[m-1:z] + M[z-1:0] \tag{4.3}$$

$$M \times N = M \times 2^{y} \times (1 + 2^{z} \times NN) \tag{4.4}$$

$$M \times N = 2^{y} \times (M + M \times 2^{z} \times NN) \tag{4.5}$$

$$M \times N = 2^{y} \times (2^{z} \times M[m-1:z] + M[z-1:0] + M \times 2^{z} \times NN) \tag{4.6}$$

$$M \times N = 2^{y} \times (2^{z} \times (M \times NN + M[m-1:z]) + M[z-1:0]) \tag{4.7}$$

$$M \times N = 2^{y} \times \{(M \times NN + M[m-1:z]), M[z-1:0]\} \tag{4.8}$$

$$M \times N = \{(M \times NN + M[m-1:z]), M[z-1:0]\} << y \tag{4.9}$$

The manipulated exact multiplication shown in equation (4.9) requires multiplication of variable multiplicand (M) with a constant (NN) smaller than the

constant multiplier (N), an addition, a concatenation, and a constant shift operation. Addition operation in equation (4.9) is eliminated to achieve the proposed approximate constant multiplication equation shown in equation (4.10).

$$M \times N = \{(M \times NN), M[z-1:0]\} << y \tag{4.10}$$

Concatenation and constant shift operations use no hardware resources. So, the proposed approximate technique decreases multiplication with constant N to multiplication with a smaller constant NN. Computational complexity reduction depends on the values of constants N and NN. In the best case, NN is 1 and constant multiplication is removed. In the worst case, NN is one bit smaller than N.

Figure 4.1 shows three examples of approximate constant multiplication. These examples illustrate that constant NN is much smaller than constant N. Hence, the proposed approximation technique decreases bit length of constant multiplication. It also eliminates addition operation. In the example M×80, because NN is 1, constant multiplication is also eliminated. Thus, approximate constant multiplication hardware implementing the proposed approximation technique performs M×80 without using any hardware resources.

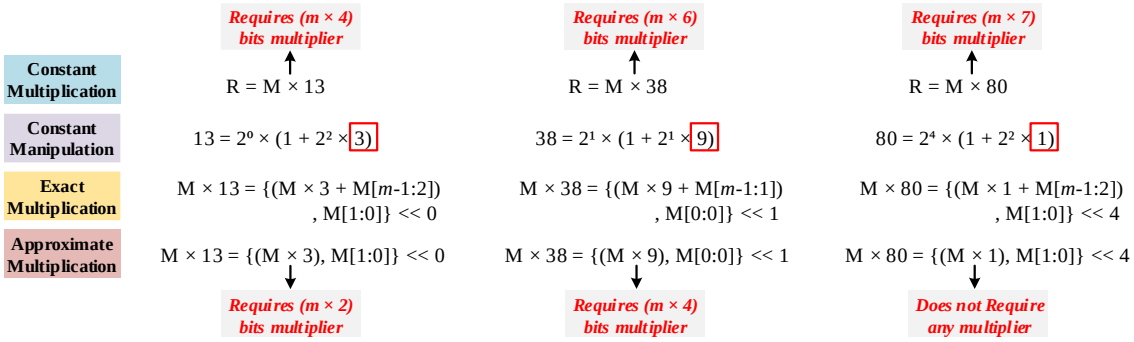

Figure 4.1 Examples of approximate constant multiplication

Figure 4.2 shows the approximate constant multiplication hardware proposed in [21]. The symbols “m” and “nn” denote bit lengths of input variable (M) and manipulated constant (NN), respectively. Because NN is always smaller than N, the proposed approximation technique decreases area and increases performance of constant multiplication hardware.

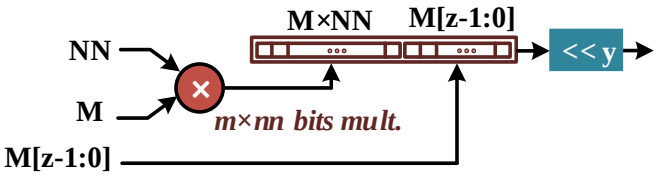

Figure 4.2 Approximate constant multiplication hardware proposed in [21]

The proposed approximate constant multiplication requires pre-determined constant multiplication, concatenation, and constant shift operations. These operations are different for each constant N. They should be determined for implementation of the datapath required for performing the approximate constant multiplication.

A python based datapath generator is used to determine constant multiplication, concatenation, and constant shift operations for an input variable and constants. If a constant is a power of 2, a constant shift operation is used to implement this constant multiplication. If a constant is a power of 2 multiple of another constant, this constant multiplication is also implemented with only a constant shift operation. The proposed approximate constant multiplication technique is used to implement the rest of constant multiplications.

HEVC uses DCT-II for transform operations. It uses 4×4, 8×8, 16×16, and 32×32 TU sizes. HEVC performs two-dimensional (2D) transform operation by applying 1D transforms in vertical and horizontal directions. The coefficients in HEVC 1D transform matrices are derived from DCT basis functions. However, integer coefficients are used for simplicity. The HEVC DCT algorithm includes 29 different constant multiplication operations. As an example, equation (4.11) shows the 4×4 DCT matrix used in HEVC.

$$DCT\_4\times4_{HEVC} = \begin{bmatrix} 64 & 64 & 64 & 64 \\ 83 & 36 & -36 & -83 \\ 64 & -64 & -64 & 64 \\ 36 & -83 & 83 & -36 \end{bmatrix} \tag{4.11}$$

Table 4.1 shows 29 different constants (N values) used in HEVC 2D DCT matrices. NN, y, and z values are determined to manipulate these constants as in equation (4.2). The corresponding approximate constant multiplication equations in the form of equation (4.10) are also shown in Table 4.1.

In Table 4.1, M is input variable and m is its bit length. Multiplications with constants 4 and 64 are exactly implemented using constant shift operations. Multiplication with an identical constant in the approximate constant multiplication equations is implemented once and the result is used in all equations. As an example, multiplication with 3 is implemented once whose result is used for multiplications with constants 13, 25, and 50.

Table 4.1 Approximate Constant Multiplication

N	Size of Exact Multiplication (bits)	NN	Size of Approximate Multiplication (bits)	y	z	Approximate Multiplication
4	-	-	-	-	-	M << 2
9	m × 4	1	-	0	3	{M, M[2:0]}
13	m × 4	3	m × 2	0	2	{(M × 3), M[1:0]}
18	m × 5	1	-	1	3	{M, M[2:0]} << 1
22	m × 5	5	m × 3	1	1	{(M × 5), M[0:0]} << 1
25	m × 5	3	m × 2	0	3	{(M × 3), M[2:0]}
31	m × 5	15	m × 4	0	1	{(M × 15), M[0:0]}
36	m × 6	1	-	2	3	{M, M[2:0]} << 2
38	m × 6	9	m × 4	1	1	{(M × 9), M[0:0]} << 1
43	m × 6	21	m × 5	0	1	{(M × 21), M[0:0]}
46	m × 6	11	m × 4	1	1	{(M × 11), M[0:0]} << 1
50	m × 6	3	m × 2	1	3	{(M × 3), M[2:0]} << 1
54	m × 6	13	m × 4	1	1	{(M × 13), M[0:0]} << 1
57	m × 6	7	m × 3	0	3	{(M × 7), M[2:0]}
61	m × 6	15	m × 4	0	2	{(M × 15), M[1:0]}
64	-	-	-	-	-	M << 6
67	m × 7	33	m × 6	0	1	{(M × 33), M[0:0]}
70	m × 7	17	m × 5	1	1	{(M × 17), M[0:0]} << 1
73	m × 7	9	m × 4	0	3	{(M × 9), M[2:0]}
75	m × 7	37	m × 6	0	1	{(M × 37), M[0:0]}
78	m × 7	19	m × 5	1	1	{(M × 19), M[0:0]} << 1
80	m × 7	1	-	4	2	{M, M[1:0]} << 4
82	m × 7	5	m × 3	1	3	{(M × 5), M[2:0]} << 1
83	m × 7	41	m × 6	0	1	{(M × 41), M[0:0]}
85	m × 7	21	m × 5	0	2	{(M × 21), M[1:0]}
87	m × 7	43	m × 6	0	1	{(M × 43), M[0:0]}
88	m × 7	5	m × 3	3	1	{(M × 5), M[0:0]} << 3
89	m × 7	11	m × 4	0	3	{(M × 11), M[2:0]}
90	m × 7	11	m × 4	1	2	{(M × 11), M[1:0]} << 1

Figure 4.3 shows the HEVC 2D DCT hardware presented in [12], which is selected to apply the proposed approximate constant multiplication technique. In [21],

all the 29 different constant multiplication operations used in HEVC 2D DCT are implemented using the proposed approximate constant multiplication technique regardless of their average percentage error.

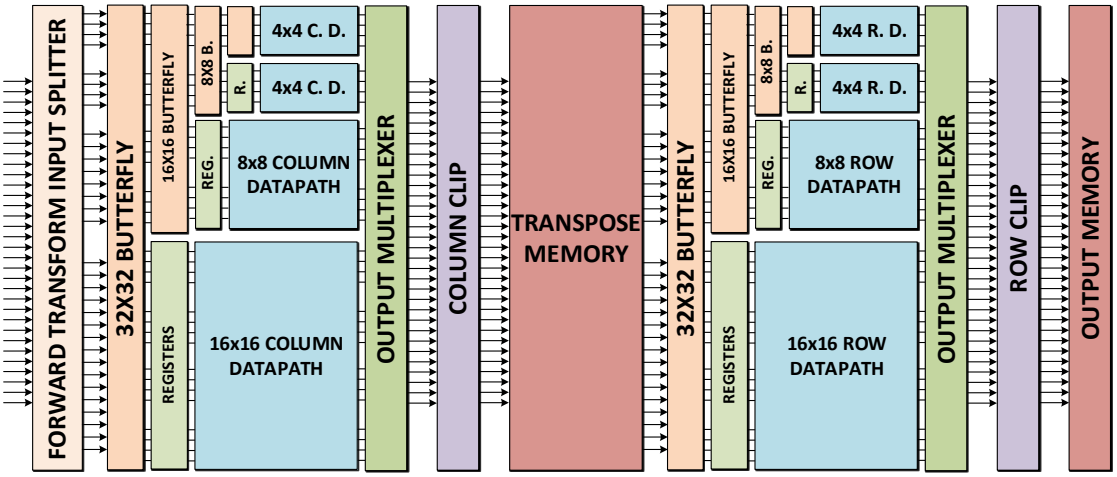

Figure 4.3 HEVC 2D DCT hardware [12]

Error produced by the proposed approximate constant multiplier is different for each constant. Errors produced for the constants used in HEVC 2D DCT are calculated as follows. Equations (4.12)-(4.14) show the calculation of average percentage error for a constant N. Input variable bit length is taken as eight bits. The constant is multiplied with all possible values of the input variable, i.e., 0 to 255, using both the exact multiplier and the proposed approximate constant multiplier. Error for the input variable value k (E_k) is obtained by calculating the absolute difference between the exact multiplication result and the approximate multiplication result as shown in equation (4.12). Equation (4.13) shows the percentage error calculation for the input variable value k (PE_k). Average percentage error for the constant N is obtained by calculating average of percentage errors for all possible values of input variable, i.e., 0 to 255, as shown in equation (4.14).

$$E_k = \left| exact(C \times k) - appr(C \times k) \right| \tag{4.12}$$

$$PE_k = \frac{E_k}{exact(C \times k)} \times 100 \tag{4.13}$$

$$average\,percentage\,error = \frac{\sum_{k=0}^{255} PE_k}{256} \tag{4.14}$$

Figure 4.4 shows the average percentage errors for the constants used in HEVC 2D DCT proposed in [21]. The proposed approximate constant multiplier causes very small errors for most of the constants.

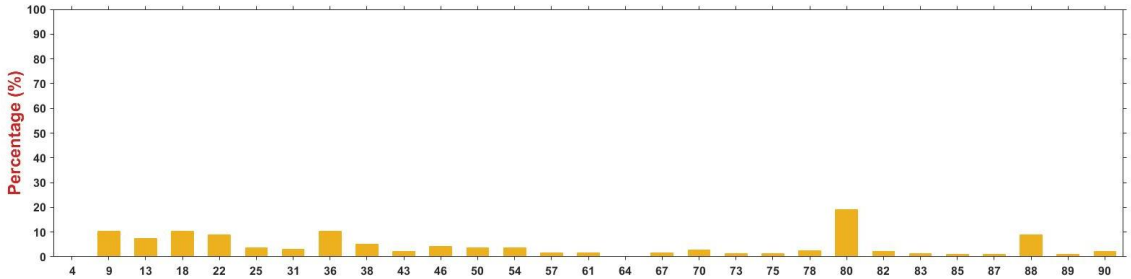

Figure 4.4 Average percentage error (%) for the constants [21]

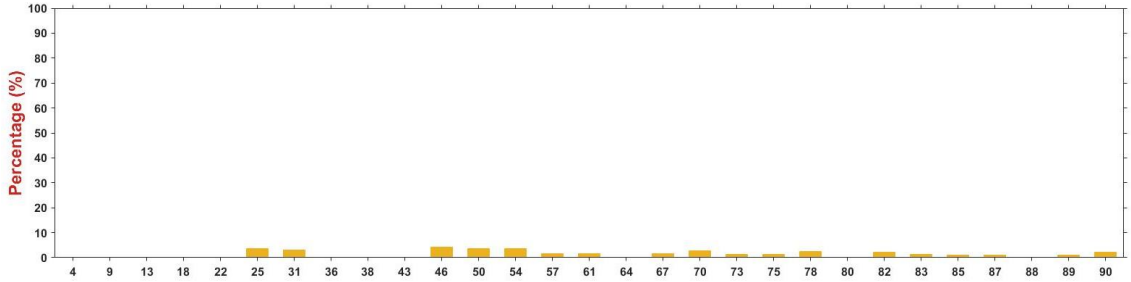

Figure 4.5 Average percentage error (%) for the constants in the proposed hardware.

4.2 Proposed Approximate HEVC DCT

In the proposed approximate HEVC DCT, to decrease quality loss, the approximate constant multiplication technique, proposed in [21], is applied only to the constant multiplications that do not cause high average percentage error. So, multiplications with N = 9, 18, 36, 80 are performed using exact constant multiplication.

Some of the constant multiplications are performed once and their results are used in different equations so that the number of multiplications is reduced. Also, more common constant multiplications are calculated without any approximation, resulting in further reduction in quality loss. For example, there is no need for an approximate calculation of M×43 because 43 is also used as NN for M×87. Therefore, instead of using approximate constant multiplication, M×43 is performed using exact constant multiplication, and its result is also used in approximate calculation of M×87.

M×13 is performed using exact constant multiplication and its result is also used in the approximate calculation of M×54 whose NN is 13. M×19 is performed using exact constant multiplication. Its result is shifted one bit to the left to obtain M×38 and used in the approximate calculation of M×78 whose NN is 19. M×11 is performed using exact constant multiplication and its result is shifted one bit and three bits to the left to obtain M×22 and M×88, respectively. It is also used in the approximate calculation of M×46 whose NN is 11.

Table 4.2 shows the 29 different constants (N values) used in HEVC 2D DCT matrices. The type of multiplication performed for each constant in the proposed HEVC 2D DCT is given in the table. For the approximate multiplications, NN, y, and z values are determined as shown in equation (4.2) to manipulate the constants. These values and the corresponding approximate constant multiplication equations in the form of equation (4.10) are also shown in Table 4.2. In the table, M is the input variable.

Figure 4.5 shows the average percentage errors for the constants used in the proposed HEVC 2D DCT. The proposed HEVC 2D DCT hardware performs both approximate and exact multiplications. Therefore, it has much less average percentage errors than the approximate constant multiplications in [21].

The proposed constant multiplications are integrated into DCT operations performed by HEVC HM reference software encoder 15.0 [24]. Their impact on rate-distortion performance is determined for several videos [35]. Their first 10 frames are coded with all intra configuration and quantization parameters (QP) 22, 27, 32, 37 using HEVC HM 15.0 [24] with and without the proposed constant multiplications.

The BD-Rate and BD-PSNR values [60] for the HEVC DCT hardware proposed in [12], [21], and the proposed approximate HEVC DCT are given in Table 4.3. The proposed approximate HEVC DCT reduces the computational complexity at the cost of slight reduction in PSNR and slight increase in bitrate, but it has much better rate-distortion performance than the HEVC DCT hardware proposed [12] and [21].

Table 4.2 Constant Multiplications Used in the Proposed Hardware

N	Type of Multiplication (Exact/ Approximate)	NN	y	z	Approximate Multiplication
4	Exact	-	-	-	-
9	Exact	-	-	-	-
13	Exact	-	-	-	-
18	Exact	-	-	-	-
22	Exact	-	-	-	-
25	Approximate	3	0	3	{(M × 3), M[2:0]}
31	Approximate	15	0	1	{(M × 15), M[0:0]}
36	Exact	-	-	-	-
38	Exact	-	-	-	-
43	Exact	-	-	-	-
46	Approximate	11	1	1	{(M × 11), M[0:0]} << 1
50	Approximate	3	1	3	{(M × 3), M[2:0]} << 1
54	Approximate	13	1	1	{(M × 13), M[0:0]} << 1
57	Approximate	7	0	3	{(M × 7), M[2:0]}
61	Approximate	15	0	2	{(M × 15), M[1:0]}
64	Exact	-	-	-	-
67	Approximate	33	0	1	{(M × 33), M[0:0]}
70	Approximate	17	1	1	{(M × 17), M[0:0]} << 1
73	Approximate	9	0	3	{(M × 9), M[2:0]}
75	Approximate	37	0	1	{(M × 37), M[0:0]}
78	Approximate	19	1	1	{(M × 19), M[0:0]} << 1
80	Exact	-	-	-	-
82	Approximate	5	1	3	{(M × 5), M[2:0]} << 1
83	Approximate	41	0	1	{(M × 41), M[0:0]}
85	Approximate	21	0	2	{(M × 21), M[1:0]}
87	Approximate	43	0	1	{(M × 43), M[0:0]}
88	Exact	-	-	-	-
89	Approximate	11	0	3	{(M × 11), M[2:0]}
90	Approximate	11	1	2	{(M × 11), M[1:0]} << 1

The proposed approximate HEVC 2D DCT hardware is implemented using the HEVC 2D DCT hardware architecture proposed in [12]. Each HEVC 1D DCT includes two different 4×4 datapaths, an 8×8 datapath, and a 16×16 datapath. These datapaths support 4×4, 8×8, 16×16, and 32×32 TUs. In the proposed hardware, each datapath first calculates the exact constant multiplications and the results are used either directly or as NN multiplication results to obtain approximate constant multiplications. The exact constant multiplications used in the proposed approximate HEVC 2D DCT hardware are implemented using Hcub multiplierless constant multiplication (MCM) technique [26].

Table 4.3 BD-Rate and BD-PSNR Results

Video		[12]		[21]		Proposed	
		BD-Rate (%)	BD-PSNR (dB)	BD-Rate (%)	BD-PSNR (dB)	BD-Rate (%)	BD-PSNR (dB)
2560×1600	People on Street	1.89	-0.10	2.37	-0.13	0.96	-0.05
2560×1600	Traffic	1.76	-0.09	2.64	-0.14	0.94	-0.05
1920×1080	Tennis	2.32	-0.06	3.15	-0.09	0.75	-0.02
1920×1080	Basketball Drive	4.06	-0.13	1.86	-0.04	0.31	-0.008
1920×1080	Park Scene	2.52	-0.10	2.12	-0.09	0.69	-0.029
1280×720	Vidyo1	2.09	-0.09	1.99	-0.09	0.58	-0.029
1280×720	Vidyo4	2.85	-0.12	1.91	-0.08	0.55	-0.02
1280×720	Kristen and Sara	2.25	-0.11	1.71	-0.08	0.72	-0.03
832×480	Party Scene	0.61	-0.05	0.20	-0.01	0.17	-0.01
832×480	Race Horses	1.58	-0.10	1.01	-0.06	0.36	-0.02
832×480	Basketball Drill	0.44	-0.02	0.56	-0.02	0.08	-0.003
Average		2.03	-0.08	1.77	-0.07	0.55	-0.02

In the first 4×4 datapath, three constant multiplications with 36, 83, and 64 are performed. M×64 is implemented exactly by constant shift operations. M×36 is implemented using exact constant multiplication since its approximate multiplication generates high average error. M×83 is approximately calculated using M×41 as shown in Table 4.2. Figure 4.6 shows the implementation of the exact constant multiplications M×36 and M×41 in the proposed first 4×4 datapath using MCM technique.

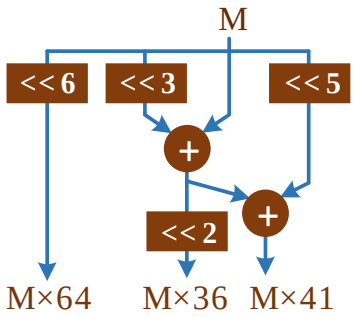

Figure 4.6 Exact multiplications required in the proposed first 4×4 datapath

In the second 4×4 datapath, four constant multiplications with 18, 50, 75, and 89 are performed. In the proposed datapath, M×18 is implemented using exact multiplication since its approximate multiplication generates high average error. M×50, M×75, and M×89 are approximately calculated using M×3, M×37, and M×11, respectively, as shown in Table 4.2. Figure 4.7 shows the implementation of the exact constant multiplications M×18, M×3, M×37, M×11 in the proposed second 4×4 datapath using MCM technique.

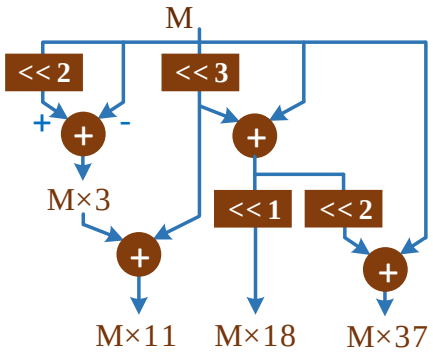

Figure 4.7 Exact multiplications required in the proposed second 4×4 datapath

In the 8×8 datapath, eight constant multiplications with 9, 25, 43, 57, 70, 80, 87, and 90 are performed. In the proposed datapath, M×9 and M×80 are implemented using exact constant multiplications since their approximate multiplications generate high average error. M×43 is also implemented using exact constant multiplication because its exact result is required in the approximate calculation of M×87. M×25, M×57, M×70, M×87, and M×90 are approximately calculated using M×3, M×7, M×17, M×43, and M×11, respectively, as shown in Table 4.2. Figure 4.8 shows the implementation of the exact constant multiplications M×9, M×80, M×43, M×3, M×7, M×11, and M×17 in the proposed 8×8 datapath using MCM technique.

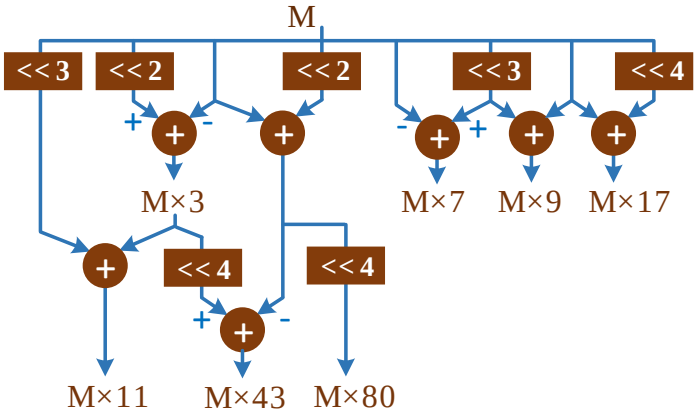

Figure 4.8 Exact multiplications required in the proposed 8×8 datapath

In the 16×16 datapath, 15 constant multiplications with 4, 13, 22, 31, 38, 46, 54, 61, 67, 73, 78, 82, 85, 88, and 90 are performed. M×4 is implemented exactly by constant shift operations. In the proposed datapath, M×13, M×22, M×38, and M×88 are implemented using exact constant multiplications, because their approximate multiplications generate high average error and the shifted values of their exact results

are required in approximate calculations of other constant multiplications. M×38 is obtained by one-bit shift to the left of M×19 result which is exactly calculated for approximate calculation of M×78. M×22 and M×88 are respectively obtained by one-bit and three-bit shifts to the left of the M×11 result which is exactly calculated for approximate calculations of M×46 and M×90. The exact result of M×13 is required in the approximate calculation of M×54. M×31, M×46, M×54, M×61, M×67, M×73, M×78, M×82, M×85, and M×90 are approximately calculated using M×15, M×11, M×13, M×15, M×33, M×9, M×19, M×5, M×21, and M×11, respectively, as shown in Table 4.2. Figure 4.9 shows the implementation of the exact constant multiplications M×13, M×22, M×38, M×88, M×5, M×9, M×11, M×15, M×19, M×21, and M×33 in the proposed 16×16 datapath using MCM algorithm.

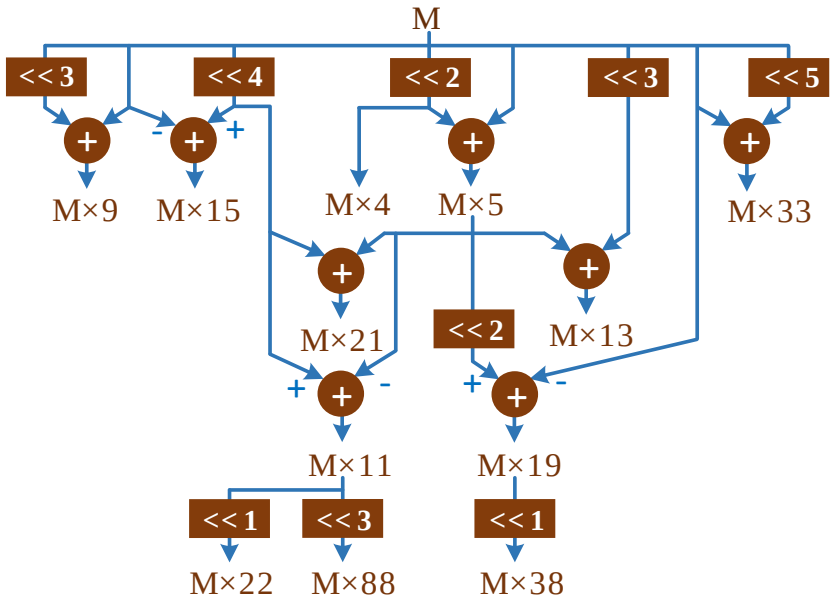

Figure 4.9 Exact multiplications required in the proposed 16×16 datapath

4.3 Implementation Results

We implemented 5 different HEVC 2D DCT hardware using the HEVC 2D DCT hardware architecture proposed in [12]. The only difference between them is the multipliers used to implement constant multiplications in HEVC DCT. The first hardware (Orig_Mult) uses exact multipliers. The second hardware (Orig_MCM) performs the multiplications exactly using the MCM algorithm. The third hardware uses the approximate constant multipliers proposed in [21]. The fourth hardware (21_MCM) uses the MCM algorithm to implement the approximate constant multipliers proposed

in [21]. The fifth hardware, our proposed hardware, uses MCM algorithm to implement the proposed constant multiplications shown in Table 4.2.

All these HEVC DCT hardware support 4×4, 8×8, 16×16, and 32×32 TUs. They perform 2D DCT in the same number of clock cycles. They perform 2D DCT by first applying 1D DCT on the columns of a TU, and then applying 1D DCT on the rows of the TU. Transpose memory is used for storing the coefficients of 1D column DCT. These stored coefficients are used as inputs for 1D row DCT.

All HEVC DCT hardware are implemented using Verilog HDL. Verilog RTL codes are implemented to a 28 nm FPGA. FPGA implementations are verified with post implementation timing simulations. Post place and route simulation results matched the results of HEVC DCT software implementation. Table 4.4 shows the FPGA implementation results.

The proposed approximate HEVC 2D DCT hardware has higher performance, less LUTs, less DFFs, less Slices, and no DSP block, compared to the approximate HEVC 2D DCT hardware proposed in [21]. To make a fair comparison, we compared the proposed hardware with the fourth hardware (21_MCM) using MCM algorithm to implement the approximate constant multipliers proposed in [21]. As it can be seen in Table 4.4, the proposed hardware has higher performance and less area than the 21_MCM hardware.

Power consumptions of all HEVC DCT hardware are estimated using Xilinx Vivado 2020.1 for transforming six 4×4 TUs, four 8×8 TUs, four 16×16 TUs, five 32×32 TUs. To estimate power consumption of an FPGA implementation, timing simulation of its placed and routed netlist is done at 25 MHz using Mentor Graphics QuestaSim. Switching activities in this timing simulation are stored in a switching activity interchange format (SAIF) file. This SAIF file is used by Vivado 2020.1 to estimate power consumption of the FPGA implementation. Table 4.5 shows the power consumption results. The proposed hardware has less power consumption than the one in [21].

The comparison of the proposed approximate HEVC 2D DCT hardware with the exact and approximate HEVC DCT hardware in the literature is shown in Table 4.6. The results shown as "---" have not been reported in the corresponding papers. [57] and [58] are excluded in the comparison because they have not reported FPGA implementation results. In [54] and [55], throughput values are reported as 361 Mpixels/sec and 52 Mpixels/sec, respectively, which are equivalent with 43 and 6 Quad

Full HD (QFHD) frames per second. The proposed approximate HEVC 2D DCT hardware is faster and has less area than the HEVC 2D DCT hardware in the literature. The proposed approximate hardware, in the worst case, can process 76 QFHD (3840×2160) frames per second.

Table 4.4 FPGA Implementation Comparison

	Orig_Mult	Orig_MCM	[21]	21_MCM	Proposed
FPGA	28 nm	28 nm	28 nm	28 nm	28 nm
LUT	28050	32203	27887	27426	27294
DFF	11652	11695	11702	11682	11667
Slice	8397	9058	8163	7755	7746
BRAM	32	32	32	32	32
DSP Block	368	0	108	0	0
Frequency (MHz)	148.3	158.2	156.7	158.2	158.7

Table 4.5 Power Consumption Comparison

	Orig_Mult	Orig_MCM	[21]	21_MCM	Proposed
Clock (mW)	19	16	18	16	17
Signal (mW)	254	362	248	267	266
Logic (mW)	227	340	229	235	246
BRAM (mW)	25	25	25	25	25
DSP (mW)	176	0	48	0	0
Total (mW)	701	743	568	543	554

Table 4.6 Comparison with HEVC DCT Hardware

	[53]	[54]	[55]	[12]	[59]	[14]	[21]	Proposed
Approximate(A) / Exact (E)	E	E	E	A	A	A	A	A
Transform	2D	2D DCT	2D DCT	2D DCT	1D DCT	2D	2D DCT	2D DCT
TU Size	16	4,8,16,32	4,8,16,32	4,8,16,32	32	4,8,16,32	4,8,16,32	4,8,16,32
FPGA	65 nm	28 nm	65 nm	40 nm	45 nm	40 nm	28 nm	28 nm
LUT	16002	5.6 K	54305	35555	1776	30701	27887	27294
DFF	---	---	15607	11230	---	10965	11702	11667
Slice	---	---	---	10080	---	8924	8163	7746
BRAM	---	---	---	32	---	16	32	32
DSP Block	---	128	384	0	---	0	108	0
Frequency (MHz)	27	177	90	100	---	104	156	158
fps	35 QFHD	43 QFHD	6 QFHD	48 QFHD	---	---	75 QFHD	76 QFHD

CHAPTER V

FPGA IMPLEMENTATION OF VIDEO COMPRESSION ALGORITHMS USING HIGH-LEVEL SYNTHESIS

High-level synthesis (HLS) is used to increase productivity [61], [62]. HLS tool takes the behavioral description of the application, such as C++ code, and automatically generates an RTL description [63]-[65].

We propose the first FPGA implementations of VVC FI algorithm using an HLS tool in the literature. In this thesis, we also propose the first FPGA implementations of HEVC fractional motion estimation (FME) algorithm using an HLS tool in the literature. We also propose novel FPGA implementations of HEVC two-dimensional (2D) discrete cosine transform (DCT) algorithm using an HLS tool.

As the first VVC FI HLS implementations [22], three different C++ codes are developed based on the VVC test model software encoder (VTM) [23]. In these C++ codes, called VVC-FI-MUL-HLS, VVC-FI-ASH-HLS, and VVC-FI-MCM-HLS, constant multiplications are implemented with multiplication operations, addition and shift operations, and Hcub multiplierless constant multiplication (MCM) algorithm [26], respectively.

As the first HEVC FME HLS implementations, two different C++ codes are developed based on the HEVC reference software encoder (HM) [24]. In these C++ codes, called HEVC-FME-MUL-HLS and HEVC-FME-DC-HLS, constant

multiplications are implemented with multiplication operations and decomposed coefficients technique [20], respectively.

As novel HEVC 2D DCT HLS implementations, two different C++ codes are developed based on the HEVC reference software encoder (HM) [24]. In these C++ codes, called HEVC-DCT-MUL-HLS and HEVC-DCT-MCM-HLS, constant multiplications are implemented with multiplication operations and Hcub MCM algorithm [26], respectively.

All the C++ codes are synthesized to Verilog RTL using Xilinx Vivado HLS tool. The Verilog RTL codes are implemented to Xilinx Virtex-7 FPGA using Xilinx Vivado tool. The best proposed VVC FI HLS implementation can process 62 full HD (1920×1080) video frames per second (fps). The best proposed HEVC FME HLS implementation supports all the prediction unit (PU) sizes and can process 23 full HD fps. The best HEVC 2D DCT HLS implementation supports all the transform unit (TU) sizes and can process 65 full HD fps.

We proposed the first HLS implementation of VVC FI algorithm in [22]. There is no other HLS implementation of VVC FI algorithm in the literature.

There are several HEVC FME hardware in the literature. But there is no HLS implementation of HEVC FME algorithm in the literature. In [66], a highly parallel HEVC FME hardware is proposed for the 8×8 PU size. In [67], an HEVC FME hardware with a scalable search pattern is proposed. In [68], an HEVC FME hardware is proposed using the sum of absolute differences (SAD) values of neighboring search locations (SLs) to calculate SAD values of fractional SLs. It decreases computational complexity at the cost of quality loss. In [69], low-power and memory-aware approximate hardware is proposed for HEVC FME.

In [53], two instances of one-dimensional (1D) DCT are used to propose a low-cost and high-throughput HEVC 16×16 2D DCT hardware. In [12], a computation and energy reduction technique for HEVC 2D DCT is proposed. This technique decreases the computational complexity of HEVC DCT at the cost of a reduction in peak-signal-to-noise-ratio (PSNR) and increase in bitrate. In [14], an algorithm to compute the minimum number of low-frequency DCT-output/IDCT-input coefficients in HEVC is proposed. It causes a slight reduction in PSNR and increase in bitrate. The HEVC 2D DCT hardware proposed in [55] uses the maximum circuit reuse during computation. In [70], HLS implementations of HEVC 2D IDCT are proposed.

5.1 VVC FI HLS Implementations

Coefficients of the VVC FI FIR filters are given in Table 2.1. In the C++ code of VVC-FI-MUL-HLS, multiplication operations are used to implement constant multiplications. In the VVC-FI-ASH-HLS, addition and shift operations are used to implement constant multiplications. In the VVC-FI-MCM-HLS, constant multiplications are implemented using MCM algorithm [26]. These C++ codes are synthesized to Verilog RTL using Xilinx Vivado HLS tool.

15×15 integer pixels are used for FI of an 8×8 prediction unit (PU). In VVC-FI-MCM-HLS, Hcub MCM algorithm multiplies a single input with multiple constants such that the number of adders and their bit size decrease. VVC-FI-MCM-HLS calculates a common offset for 15 different FIR filter equations to decrease the number of constant multiplications. It also calculates common sub-expressions in several FIR filter equations once and uses the results in corresponding equations.

In the C++ codes, we use two functions called calculation and filter. In all the HLS implementations VVC-FI-MUL-HLS, VVC-FI-ASH-HLS, and VVC-FI-MCM-HLS, filter function is the same. It takes 15 rows of 15 integer pixels as inputs. A for loop with 15 iterations is used in filter function for HHPs interpolation. Figure 5.1 shows a part of this for loop. In each iteration, the input pixels, which are later used for VHPs interpolation, are stored into temp_ver arrays. One row of 15 integer pixels is given to calculation function which interpolates 8×15 HHPs in parallel. In 15 iterations, using 15 rows of 15 integer pixels, 8×15×15 HHPs are interpolated and stored into hpa1[8], …, hpa15[8] arrays.

In filter function, there are 16 for loops with 8 iterations including one for loop used for interpolating 8×8×15 VHPs using integer pixels, and 15 for loops used for interpolating 8×8×225 QPs using HHPs. The QPs interpolated in each iteration are stored into qp1[8], …, qp15[8] arrays. Memories out_mem1[136], …, out_mem15[136] are used for storing all the output fractional pixels. We used a rotating addressing scheme to store the HHPs into transpose memories tr_mem1[15], …, tr_mem15[15]. However, Xilinx Vivado HLS tool did not recognize it. So, there is no difference between the HLS implementations with and without rotating addressing scheme.

```
int c = 0, z = 0;
for (int a = 0;a<15;a++) {
    temp_ver[0][14 - a] = input4[a];
    temp_ver[1][14 - a] = input5[a];
       .                  .
       .                  .
    temp_ver[7][14 - a] = input11[a];
    calculation(input1[a], input2[a], input3[a], input4[a], input5[a], input6[a], input7[a], input8[a],
        input9[a], input10[a], input11[a], input12[a], input13[a], input14[a], input15[a], hpa1, hpa2,
        hpa3, hpa4, hpa5, hpa6, hpa7, hpa8, hpa9, hpa10, hpa11, hpa12, hpa13, hpa14, hpa15);
    if (a>2 && a<11) {
        out_mem1[z] = (hpa1[0], hpa1[1], hpa1[2], hpa1[3], hpa1[4], hpa1[5], hpa1[6], hpa1[7]);
        out_mem2[z] = (hpa2[0], hpa2[1], hpa2[2], hpa2[3], hpa2[4], hpa2[5], hpa2[6], hpa2[7]);
            .                  .                  .                  .
            .                  .                  .                  .
        out_mem15[z] = (hpa15[0], hpa15[1], hpa15[2], hpa15[3], hpa15[4], hpa15[5], hpa15[6], hpa15[7]);
        z++; }
    // Storing the horizontal half pixels in the transpose memory
    if (a == 0) {
        tr_mem15[c] = (hpa1[0], hpa1[1], hpa1[2], hpa1[3], hpa1[4], hpa1[5], hpa1[6], hpa1[7]);
        tr_mem14[c] = (hpa2[0], hpa2[1], hpa2[2], hpa2[3], hpa2[4], hpa2[5], hpa2[6], hpa2[7]);
            .                  .                  .                  .
            .                  .                  .                  .
        tr_mem1[c] = (hpa15[0], hpa15[1], hpa15[2], hpa15[3], hpa15[4], hpa15[5], hpa15[6], hpa15[7]); }
    else if (a == 1) {
        tr_mem14[c] = (hpa1[0], hpa1[1], hpa1[2], hpa1[3], hpa1[4], hpa1[5], hpa1[6], hpa1[7]);
        tr_mem13[c] = (hpa2[0], hpa2[1], hpa2[2], hpa2[3], hpa2[4], hpa2[5], hpa2[6], hpa2[7]);
            .                  .                  .                  .
            .                  .                  .                  .
        tr_mem15[c] = (hpa15[0], hpa15[1], hpa15[2], hpa15[3], hpa15[4], hpa15[5], hpa15[6], hpa15[7]); }

        .        .        .        .        .        .        .        .        .        .        .
        .        .        .        .        .        .        .        .        .        .        .

    else if (a == 14) {
        tr_mem1[c] = (hpa1[0], hpa1[1], hpa1[2], hpa1[3], hpa1[4], hpa1[5], hpa1[6], hpa1[7]);
        tr_mem15[c] = (hpa2[0], hpa2[1], hpa2[2], hpa2[3], hpa2[4], hpa2[5], hpa2[6], hpa2[7]);
            .                  .                  .                  .
            .                  .                  .                  .
        tr_mem2[c] = (hpa15[0], hpa15[1], hpa15[2], hpa15[3], hpa15[4], hpa15[5], hpa15[6], hpa15[7]);  }
    c++;
}
```

Figure 5.1 Part of the C++ codes performing HHPs interpolation

Calculation function takes 15 integer pixels or 15 HHPs as inputs temp14, temp13, …, temp0. It calculates 8 sets of 15 FIR filter (F_1,…,F_{15}) equations. Thus, 8×15 fractional pixels are interpolated in parallel. For each HLS implementations VVC-FI-MUL-HLS, VVC-FI-ASH-HLS, and VVC-FI-MCM-HLS, calculation function is written depending on how constant multiplications are implemented in that HLS implementation. Part of the calculation function in C++ codes of VVC-FI-MCM-HLS implementation, which calculates FIR filter F_5 denoted as hpa5[0], is shown in Figure 5.2. The common sub-expressions and offset value shown in Figure 5.2 are calculated only once and used for calculating other FIR filters as well.

```
temp10_3 = (temp10 << 2) - temp10;
temp12_3 = (temp12 << 2) - temp12;
temp11_5 = (temp11 << 2) + temp11;
X11_X13 = (temp11 << 3) + temp13;
X10_X8 = (temp10 << 3) + temp8;
X12_X14 = (temp12 << 3) + temp14;
X9_X7 = (temp9 << 3) + temp7;
offset0=(ap_uint<15>(X11_X13+X10_X8)<<2)-(X12_X14+X9_X7);
F5_12[0] = offset0 - temp12_3;
hpa5[0] = ((ap_uint<13>(temp11_5) << 2) -
    (ap_uint<11>(temp10_3) << 1) - temp8 + F5_12[0]) >>6;
```

Figure 5.2 Part of the calculation function in C++ codes of VVC-FI-MCM-HLS

HLS tool generates Verilog RTL code based on default behavior, constraints, and optimization directives. We used several optimization directives to improve performance of the proposed HLS implementations.

In the Xilinx Vivado HLS tool, a library is provided to design bit-accurate models in C++ codes [71]. As shown in Figure 5.2, ap_uint<> bit accurate data type is used in the proposed HLS implementations to reduce adder bit widths and therefore hardware area.

Using array partition (APAR) directive, the large arrays are partitioned into distinct registers to improve access to data and remove block RAM bottleneck. We applied APAR directive to hpa1[8], …, hpa15[8], tr_mem1[15], …, tr_mem15[15], qp1[8], …, qp15[8].

Pipeline (PIPE) directive uses pipelining which improves performance. We apply PIPE directive to the proposed HLS implementations in two ways which are denoted as PIPE(1) and PIPE(2). In PIPE(1), PIPE directive is applied only to the for loops. In PIPE(2), in addition to the for loops, PIPE directive is also applied to the calculation function.

UNROLL directive unrolls the loops so that iterations are implemented in parallel. In the proposed HLS implementations, the for loop with 15 iterations is unrolled 15 times and the for loops with 8 iterations are unrolled 8 times.

The Verilog RTL codes generated by Xilinx Vivado HLS tool for the C++ codes are verified with RTL simulations. The Verilog RTL codes are implemented to Xilinx Virtex-7 FPGA using Xilinx Vivado 2020.1. The FPGA implementations are verified with post place and route simulations.

Table 5.1 shows FPGA implementation results of VVC-FI-MUL-HLS. The multiplication operations are mapped to DSP48 blocks. Array partition (APAR), pipeline (PIPE(1) and PIPE(2)), and UNROLL directives are applied to VVC-FI-MUL-HLS.

Table 5.2 shows FPGA implementation results of VVC-FI-ASH-HLS. Array partition (APAR), pipeline (PIPE(1) and PIPE(2)), and UNROLL directives are applied to VVC-FI-ASH-HLS. It has better performance than VVC-FI-MUL-HLS.

Table 5.3 shows FPGA implementation results of VVC-FI-MCM-HLS. Among the three proposed VVC FI HLS implementations, VVC-FI-MCM-HLS has the best performance with acceptable hardware area because of using Hcub MCM algorithm,

common offset, and common sub-expressions. Array partition (APAR), pipeline (PIPE(1) and PIPE(2)), and UNROLL are applied to VVC-FI-MCM-HLS.

Table 5.1 FPGA Implementation Results of the Proposed VVC-FI-MUL-HLS

	LUTs	FFs	Slices	BRAMs	DSP48	Freq (MHz)	Clock Cycles (8×8 PU)	fps FHD
No optimization	19740	41948	13124	30.5	74	130.7	2159	1
APAR	16334	35909	11166	15.5	73	119.1	811	4
APAR-PIPE(1)	17265	38482	10560	15.5	66	259.7	379	21
APAR-UNROLL	50784	40072	16537	30	292	83	215	11
APAR-UNROLL-PIPE(1)	50784	40072	16537	30	292	83	215	11
APAR-UNROLL-PIPE(2)	53019	36696	17049	30	300	88.5	73	37

Table 5.2 FPGA Implementation Results of the Proposed VVC-FI-ASH-HLS

	LUTs	FFs	Slices	BRAMs	DSP48	Freq (MHz)	Clock Cycles (8×8 PU)	fps FHD
No optimization	19470	41156	14958	30.5	0	135.1	2159	1
APAR	15774	35645	11985	15.5	0	142.9	954	4
APAR-PIPE(1)	15580	37166	11118	15.5	0	183.5	345	16
APAR-UNROLL	48711	45099	19271	30	0	122	214	17
APAR-UNROLL-PIPE(1)	48711	45099	19271	30	0	122	214	17
APAR-UNROLL-PIPE(2)	49687	41380	19094	30	0	124.2	74	51

Table 5.3 FPGA Implementation Results of the Proposed VVC-FI-MCM-HLS

	LUTs	FFs	Slices	BRAMs	DSP48	Freq (MHz)	Clock Cycles (8×8 PU)	fps FHD
No optimization	16549	40264	13614	30.5	0	145.8	2016	2
APAR	12891	33528	10523	15.5	0	166.7	811	6
APAR-PIPE(1)	14071	35299	9965	15.5	0	178.6	345	15
APAR-UNROLL	37242	41911	15933	30	0	157.5	214	22
APAR-UNROLL-PIPE(1)	37242	41911	15933	30	0	157.5	214	22
APAR-UNROLL-PIPE(2)	39047	37450	15319	30	0	150.4	74	62

In the proposed HLS implementations with APAR-UNROLL-PIPE(2), in addition to the for loops, pipelining is also applied to the calculation function. This significantly improved the performance.

In Table 5.4, the best proposed VVC MCM HLS implementation (VVC-FI-MCM-HLS with APAR-UNROLL-PIPE(2)) is compared with manual VVC FI hardware implementations proposed in [29] and [19]. To have a fair comparison, these

handwritten Verilog RTL codes are implemented to Xilinx Virtex-7 FPGA using Xilinx Vivado 2020.1. The proposed VVC-FI-MCM-HLS implementation has higher performance than [29] and [19] at the cost of larger area.

Table 5.4 VVC FI Hardware Comparison

	[29]	[19]	VVC-FI-MCM-HLS
LUT	10569	11125	39047
FF	3591	3521	37450
Slice	3079	3308	15319
BRAM	30	30	30
Frequency (MHz)	225.7	235	150.3
Clock Cycles (8×8 PU)	147	147	74
FHD (1920×1080) fps	47	49	62

5.2 HEVC FME HLS Implementation

FME is done after integer motion estimation. In HEVC reference software encoder (HM) [24], FME is done in two stages. As shown in Figure 5.3, in stage 1, eight fractional SLs around the best integer SL are searched. In stage 2, eight fractional SLs around the best fractional SL found in stage 1 are searched. HEVC FME first interpolates the fractional pixels required for fractional SLs using three different FIR filters. In Figure 5.3, HHPs a, b, c and VHPs d, h, n are interpolated using the nearest integer pixels in horizontal and vertical directions, respectively. QPs e, i, p and f, j, q and g, k, r are interpolated using the nearest a HHPs, b HHPs, and c HHPs, respectively. HEVC FME then calculates the sum of absolute difference (SAD) values for each fractional SL and determines the best fractional SL with the minimum SAD value.

Two HEVC FME HLS implementations HEVC-FME-MUL-HLS and HEVC-FME-DC-HLS are proposed. In the C++ codes, we use four functions called FI, SAD_adders, SAD_8×8, and FME. HEVC-FME-MUL-HLS and HEVC-FME-DC-HLS differ only in FI function and are the same in the other functions. In the C++ code of HEVC-FME-MUL-HLS, multiplication operations are used to implement the constant multiplications in FI function. In the HEVC-FME-DC-HLS, decomposed coefficients technique [20] is used to implement the constant multiplications in FI function. These C++ codes are synthesized to Verilog RTL using Xilinx Vivado HLS tool.

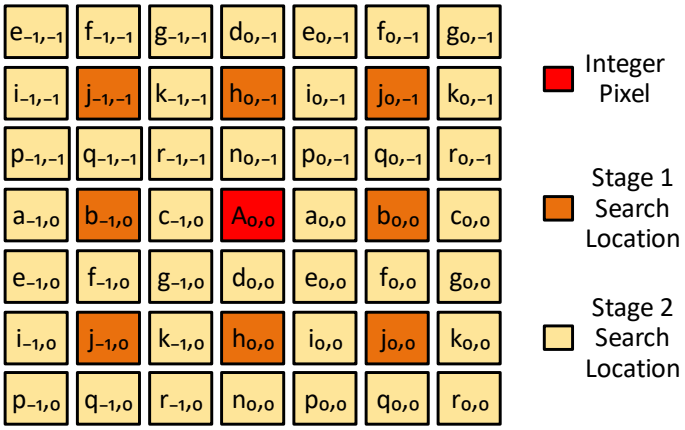

Figure 5.3 Fractional search locations

Figure 5.4 shows the HEVC-FME-DC-HLS. It takes 72 rows of 72 integer pixels as search area, 64 rows of 64 integer pixels as current block, PU size, and best SAD of integer SLs as input. SAD_8×8 function calculates the SADs of eight fractional SLs for an 8×8 PU by calling FI and SAD_adders functions. FI function takes 16 integer pixels as input and calculates 3×9 fractional pixels using 3 FIR filters. FI function in the HEVC-FME-DC-HLS decomposes coefficients of FIR filters to decrease number of additions by using decomposed coefficients technique [20]. SAD_8×8 function calls FI function 51 times (16 for HHPs + 8 for VHPs + 27 for QPs) to calculate all the fractional pixels required for FME of an 8×8 PU. Then, SAD_8×8 calls SAD_adders function to calculate 8 SADs for 8 fractional SLs. In stage 1, eight fractional SLs around the best integer SL are searched. Eight parallel SAD calculation hardware are used to calculate SAD values of these 8 SLs in parallel. Appropriate current block pixels, HHPs, VHPs, and QPs are given to SAD_adders function for the SAD calculations. If the PU size is 8×8, FME function compares the SADs of eight fractional SLs to determine the SL with minimum SAD value in stage 1. In stage 2, eight fractional SLs around the best fractional SL found in stage 1 are searched. The same hardware used in stage 1 is used for FI and SAD calculation in stage 2. Comparison hardware determines the best SAD and its location.

If PU size is larger than 8×8, it is divided to 8×8 PUs. In each stage, FME function adds up the SADs corresponding to the eight SLs in each of the 8×8 PUs to obtain the SADs of eight SLs in the main PU. Then, comparison hardware determines the best SAD and its location. To calculate SADs for 4×8 and 8×4 PUs, zero is assigned to the fractional pixels that do not exist in these small PUs. So, the proposed HLS implementations support FME for all the 24 different PU sizes in HEVC FME.

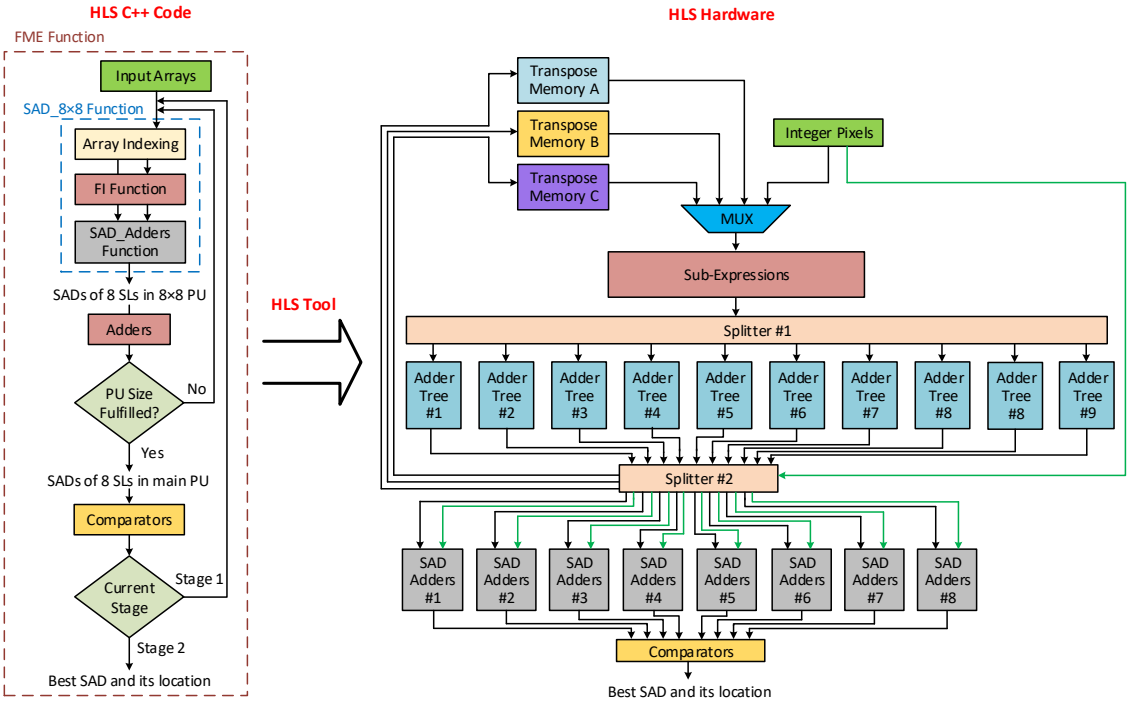

Figure 5.4 HEVC FME HLS implementation HEVC-FME-DC-HLS

To improve performance of the proposed HEVC FME HLS implementations, in addition to bit-accurate models in C++ codes, we apply UNROLL directive to the for loops and pipeline (PIPE) directive in two ways. In PIPE(1), PIPE directive is applied only to the for loops. In PIPE(2), in addition to the for loops, PIPE is also applied to the functions. The for loops are used to divide the large input arrays of a large PU to smaller arrays corresponding to smaller 8×8 PUs.

The Verilog RTL codes generated by Xilinx Vivado HLS tool are verified with RTL simulations and then implemented to Xilinx Virtex-7 FPGA using Xilinx Vivado 2020.1. The FPGA implementations are verified with post place and route simulations.

Table 5.5 and Table 5.6 show FPGA implementation results of HEVC-FME-MUL-HLS and HEVC-FME-DC-HLS, respectively. UNROLL, PIPE(1), and PIPE(2) directives are applied. The multiplication operations in HEVC-FME-MUL-HLS are mapped to DSP48 blocks.

In Table 5.7, the best proposed HEVC FME HLS implementation (HEVC-FME-DC-HLS with UNROLL-PIPE(2)) is compared with manual HEVC FME hardware implementations proposed in [66], [67] and [68]. The values shown as "---" have not been reported in [67]. Because the hardware proposed in [69] is approximate, it is

excluded in comparison. The hardware proposed in [66] supports only 8×8 PUs. The proposed HEVC-FME-DC-HLS has much better area than [67] at the cost of lower performance. The hardware proposed in [68] has better implementation results at the cost of quality loss. However, FME in our proposed HLS implementations is done without any approximation and quality loss.

Table 5.5 FPGA Implementation Results of the Proposed HEVC-FME-MUL-HLS

	LUTs	FFs	Slices	BRAMs	DSP48	Freq (MHz)	Clock Cycles (8×4 or 4×8 PUs)	FHD fps
No optimization	119381	33697	39281	0	525	114	416	4
PIPE(1)	120724	33768	39555	0	525	119	260	7
UNROLL-PIPE(1)	115684	23486	34486	20	405	66	66	15
UNROLL-PIPE(2)	54990	37043	19998	20	90	99	76	20

Table 5.6 FPGA Implementation Results of the Proposed HEVC-FME-DC-HLS

	LUTs	FFs	Slices	BRAMs	DSP48	Freq (MHz)	Clock Cycles (8×4 or 4×8 PUs)	FHD fps
No optimization	70902	19592	22196	0	0	119	411	4
PIPE(1)	70912	19645	22389	0	0	116	255	7
UNROLL-PIPE(1)	115237	24193	39841	20	0	55	66	13
UNROLL-PIPE(2)	49341	34600	18081	20	0	113	75	23

Table 5.7 HEVC FME Hardware Comparison

	[66]	[67]	[68]	HEVC-FME-DC-HLS
FPGA	28 nm	40 nm	40 nm	28 nm
LUT	17888	130306	5200	49341
FF	17946	---	3794	34600
Slice	5742	---	1814	18081
Frequency (MHz)	97	200	142	113
Supported PU sizes	8×8	All	All	All
FHD (1920×1080) fps	55	128	76	23

5.3 HEVC 2D DCT HLS Implementations

HEVC uses DCT-II for transform operations. It uses 4×4, 8×8, 16×16, and 32×32 TU sizes. HEVC performs 2D transform operation by first performing 1D column transform and then performing 1D row transform. The coefficients in HEVC 1D transform matrices are derived from DCT basis functions. However, integer coefficients are used for simplicity.

Two HEVC 2D DCT HLS implementations HEVC-DCT-MUL-HLS and HEVC-DCT-MCM-HLS are proposed. In the C++ codes, we use three functions called DCT_col, DCT_row, and DCT_2D. In the C++ code of HEVC-DCT-MUL-HLS, multiplication operations are used to implement constant multiplications. In the HEVC-DCT-MCM-HLS, Hcub MCM algorithm [26] is used to implement constant multiplications. These C++ codes are synthesized to Verilog RTL using Xilinx Vivado HLS tool.

In DCT_2D function, we use two for loops with iterations of TU size. The first for loop performs 1D column transform by calling DCT_col function and storing its outputs in a transpose memory in each iteration. The second for loop performs 1D row transform by applying the relevant data from the transpose memory to the DCT_row function in each iteration.

The Verilog RTL codes generated by Xilinx Vivado HLS tool are verified with RTL simulations and then implemented to Xilinx Virtex-7 FPGA using Xilinx Vivado 2020.1. The FPGA implementations are verified with post place and route simulations.

Table 5.8 and Table 5.9 show FPGA implementation results of HEVC-DCT-MUL-HLS and HEVC-DCT-MCM-HLS, respectively. To improve performance of the proposed HEVC DCT HLS implementations, in addition to bit-accurate models in C++ codes, pipeline (PIPE), INLINE, and resource (RES) directives are applied. We apply PIPE to the for loops.

We apply INLINE directive to the functions DCT_col and DCT_row. Function inlining removes the function hierarchy. Inlining a function may improve area by allowing the components within the function to be better shared or optimized with the logic in the calling function [71].

Resource (RES) directive is used to specify which resource will be used to implement a variable such as an array, arithmetic operation or function argument. In HEVC-DCT-MUL-HLS, we apply RES to specify DSP blocks to be used to implement multiplication operations. In both HEVC-DCT-MUL-HLS and HEVC-DCT-MCM-HLS, we apply RES to specify BRAMs to implement the input arrays.

In Table 5.10, the best proposed HEVC DCT HLS implementation (HEVC-DCT-MCM-HLS with INLINE-PIPE-RES) is compared with the manual HEVC DCT hardware implementations proposed in [53], [12], [14], [55] and [70]. The values shown as "---" have not been reported. The proposed HEVC-DCT-MCM-HLS has better area and performance than the HLS implementation proposed in [70] and manual HEVC

DCT hardware proposed in [55]. The hardware proposed in [12] and [14] are approximate hardware. The hardware proposed in [53] performs HEVC 2D DCT for only 16×16 TUs.

Table 5.8 FPGA Implementation Results of the Proposed HEVC-DCT-MUL-HLS

	LUTs	FFs	Slices	BRAMs	DSP48	Freq (MHz)	Clock Cycles (4×4 for TU)	FHD fps
No optimization	20813	23663	7442	0	535	192	71	20
INLINE	20095	21834	7832	0	537	188	59	24
PIPE	20138	23720	7831	0	535	188	51	28
INLINE-PIPE	21515	22730	8064	0	540	172	22	60
INLINE-PIPE-RES	20931	23119	8341	0	540	181	22	63

Table 5.9 FPGA Implementation Results of the Proposed HEVC-DCT-MCM-HLS

	LUTs	FFs	Slices	BRAMs	DSP48	Freq (MHz)	Clock Cycles (4×4 for TU)	FHD fps
No optimization	36247	28491	12108	0	0	181	75	18
INLINE	36667	27777	12073	0	0	187	59	24
PIPE	35652	29020	11871	0	0	187	53	27
INLINE-PIPE	37627	29289	12393	0	0	167	21	61
INLINE-PIPE-RES	37491	29374	12893	0	0	177	21	65

Table 5.10 HEVC DCT Hardware Comparison

	[53]	[12]	[14]	[55]	[70]	HEVC-DCT-MCM-HLS
HLS / Manual	Manual	Manual	Manual	Manual	HLS	HLS
Approximate(A) / Exact (E)	E	A	A	E	E	E
Transform	2D DCT	2D DCT	2D DCT/IDCT	2D DCT	2D IDCT	2D DCT
TU size	16	4,8,16,32	4,8,16,32	4,8,16,32	4,8,16,32	4,8,16,32
FPGA	65 nm	40 nm	40 nm	65 nm	40 nm	28 nm
LUT	16002	35555	30701	54305	50566	37491
FF	---	11230	10965	15607	34955	29374
Slice	---	10080	8924	---	14944	12893
BRAM	---	32	16	---	13	0
DSP blocks	---	0	0	384	0	0
Freq. (MHz)	27	100	104	90	208	177
fps	35 QFHD	48 QFHD	---	6 QFHD	54 FHD	65 FHD

CHAPTER VI

VVC AFFINE MOTION ESTIMATION HARDWARE

Inter prediction is a vital part of video coding, which aims to find a similar block in the reference frames to decrease the temporal redundancy. Motion estimation (ME) and motion compensation are the main tools of inter prediction. The basic motion model of the conventional block-based ME in HEVC is translational motion model. However, the motion of an object may happen in different forms such as rotation and zooming.

In VVC, affine motion estimation (AME) is used which considers rotation, zooming, and shearing of blocks during block matching ME. AME achieves higher compression than translational ME at the cost of much more computational complexity [72], [73].

In this thesis, to reduce the computational complexity of VVC AME, an approximate VVC AME hardware is proposed using a proposed approximate absolute difference (AD) hardware, approximate adder tree, and sub-sampling. The proposed approximate AD hardware reduces the bit length of each AD value from 8 to 5. A new approximate adder tree is proposed to decrease the bit length of the adders. To further reduce the computational complexity of VVC AME, sub-sampling is used.

6.1 VVC Affine Motion Estimation

AME has two modes. 4-parameter AME utilizes two motion vectors and 6-parameter AME utilizes three motion vectors. 4-parameter AME takes zoom and rotation into account. Equations (6.1) and (6.2) show formulas of 4-parameter AME. 6-parameter AME takes zoom, rotation, and shear into account. Equations (6.3) and (6.4) show the formulas of 6-parameter AME. Figure 6.1 shows 6-parameter AME model with three motion vectors.

Translational ME has the most computational complexity in video coding. AME is more computationally complex than translational ME. To decrease computational complexity, VVC performs AME on 4×4 sub-blocks instead of pixels. For higher spatial video resolutions, the importance of each 4×4 sub-block is reduced, which lets us apply AME on sub-blocks instead of pixels with negligible quality loss.

$$mv_x = \frac{mv_{1_x} - mv_{0_x}}{w} x - \frac{mv_{1_y} - mv_{0_y}}{w} y + mv_{0_x} \quad (6.1)$$

$$mv_y = \frac{mv_{1_y} - mv_{0_y}}{w} x - \frac{mv_{1_x} - mv_{0_x}}{w} y + mv_{0_y} \quad (6.2)$$

$$mv_x = \frac{mv_{1_x} - mv_{0_x}}{w} x - \frac{mv_{2_x} - mv_{0_x}}{w} y + mv_{0_x} \quad (6.3)$$

$$mv_y = \frac{mv_{1_y} - mv_{0_y}}{w} x - \frac{mv_{2_y} - mv_{0_y}}{w} y + mv_{0_y} \quad (6.4)$$

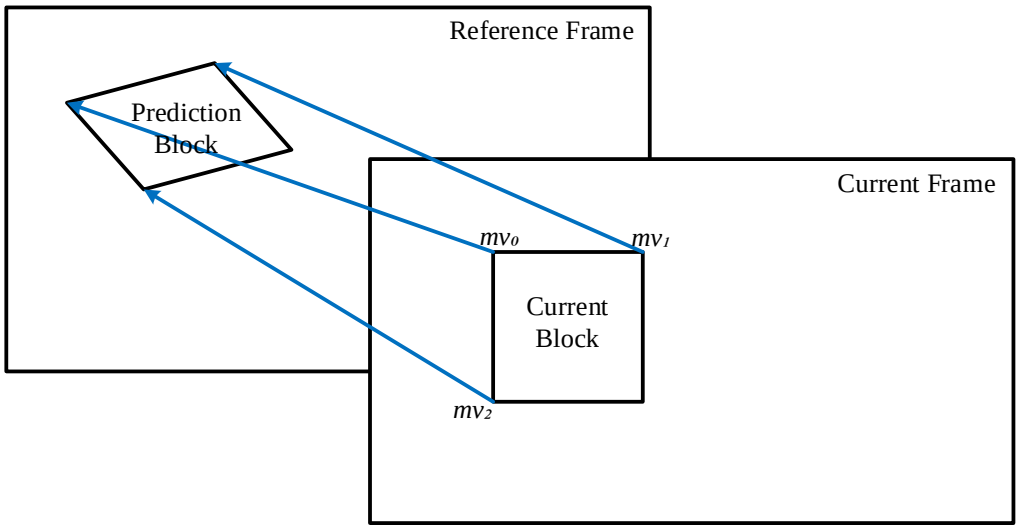

Figure 6.1 The 6-parameter affine model with three motion vectors.

In translational ME, a motion vector is calculated for every block, then all the pixels within that block are shifted to the location indicated by that motion vector regardless of their positions within the block. Whereas, in AME, new locations of pixels are calculated based on their locations within the block.

Figure 6.2 shows AME of 4×4 sub-blocks in a 16×16 block where the affine motion vectors are shown with the black arrows, and four of 16 calculated translational motion vectors are shown with the colored arrows. In AME, affine motion vectors are used to calculate translational motion vectors for the center points of the sub-blocks.

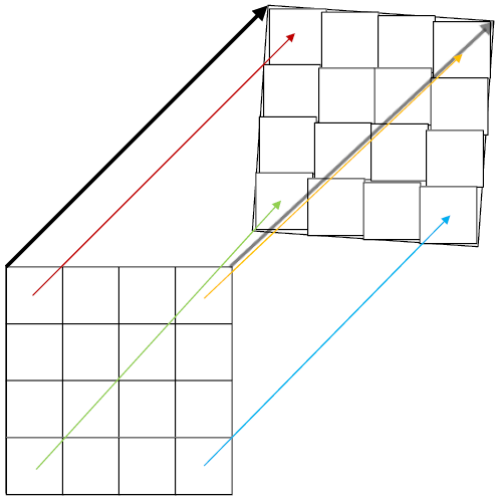

Figure 6.2 AME of 4×4 sub-blocks in a 16×16 block.

Since AME uses 2 or 3 motion vectors per block, the number of search locations increases exponentially. For instance, for a 128×128 search window, there are 16384 mv_0 search locations; while for each mv_0, 16384 mv_1 search locations exist, and for each mv_1, 16384 mv_2 search locations exist. Thus, for a 6-parameter full-search AME, 4.3×10^{12} SAD values need to be calculated. It is not feasible to calculate them. Hence, approximate AME algorithms are required such as using full search for mv_0 while searching just pre-determined search locations for mv_1 and mv_2 [72].

In [74], a VVC AME hardware is proposed to perform 4-parameter AME. It is the first VVC AME FPGA implementation in the literature. The hardware proposed in [74] uses a new pixel storage method that considerably decreases the computational complexity and the number of BRAM read operations. In the hardware proposed in [74], fixed search window size is 128×128 and block sizes are 16×16, 32×32 or 64×64. The block size is given to the hardware as input. The user determines the trade-off

between compression and speed by selecting the block size [75]. The hardware proposed in [74] searches all the mv_0 search locations in the 128×128 search window, i.e., 4096 mv_0 search locations for 64×64 block size, 9216 mv_0 search locations for 32×32 block size, and 12544 mv_0 search locations for 16×16 block size. It searches just eight pre-determined mv_1 search locations. Thus, the hardware proposed in [74] searches 32768, 73728, or 100352 search locations.

The hardware proposed in [74] has two copies of the VVC AME hardware shown in Figure 6.3, which work in parallel. Each copy consists of a translational motion vector calculation component in the control module, multiplexers to select pixels utilizing the motion vectors, 64×64 processing units for absolute difference (AD) calculation and an adder tree. After start signal, the search window pixels are read from off-chip memory and written to BRAMs. 64 pixels are read in a clock cycle. The 64 pixels are concatenated and written to a single location in BRAMs. Because the search window size is 128×128, 128 rows are stored to every BRAM. Thus, 256 clock cycles are required to store the search window pixels in BRAMs. Then, the current block pixels are read in 64, 16, or 4 clock cycles based on the block size. Next, in one clock cycle, one row of search window (128 pixels) is read from BRAMs and stored to registers. Previous and next rows are also required for SAD calculations because the affine motion vectors can point upwards and downwards. After reading the required pixels, SAD calculation begins.

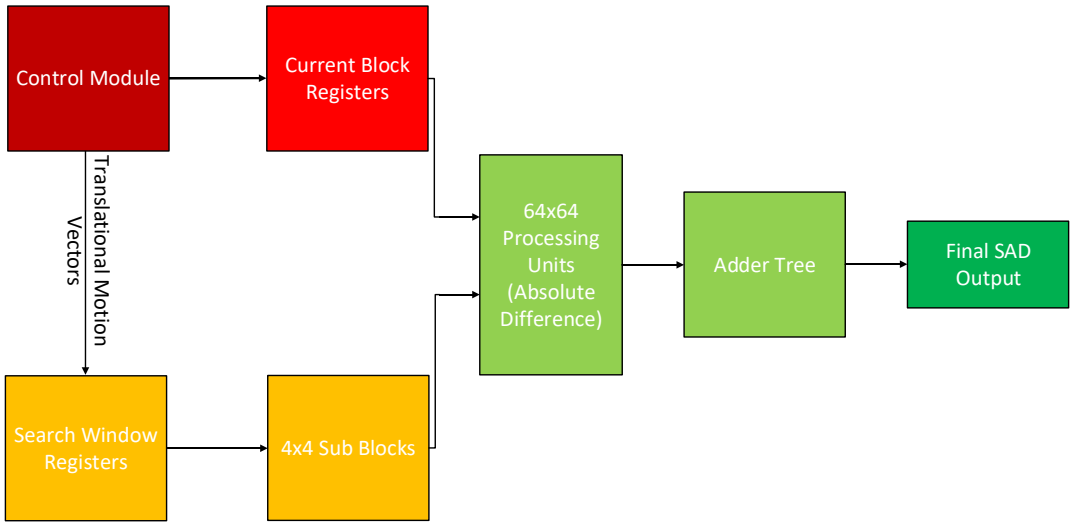

Figure 6.3 VVC affine motion estimation hardware proposed in [74].

In the hardware proposed in [74], based on the translational motion vectors calculated by the control module, proper pixels are sent to the 64×64 processing units

for AD calculation and AD results are added up by the adder tree. Eight mv_1 search locations are searched for each mv_0. Figure 6.4 shows seven mv_1 search locations. The 8th mv_1 search location is the upper right corner of the block. Each of the copies of the hardware shown in Figure 6.3 performs motion vector and SAD calculations four times. The smallest SAD and the corresponding motion vector are stored in the registers to be compared with the SAD values that will be calculated later.

The hardware proposed in [74] uses a new pixel storage method. After SAD calculations for a mv_0 search location and eight mv_1 search locations are completed, instead of incrementing mv_{0x}, the search window pixels in the registers are shifted by one to the left. Also, after SAD calculations for a row is completed, instead of incrementing mv_{0y}, the search window pixels in the registers are shifted up and a new row is read from the BRAMs. After all the SAD calculations are completed, the smallest SAD and the related motion vector are sent to the output. This process is repeated for each block.

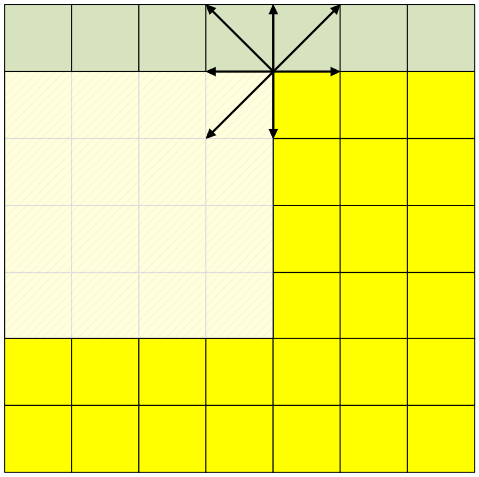

Figure 6.4 MV_1 locations in the VVC AME hardware proposed in [74].

This pixel storage method uses a large number of registers. However, it has three advantages. First, it considerably decreases the number of BRAM read operations. Second, it removes complex address generation for BRAMs. Third, it notably simplifies calculation of translational motion vectors from affine motion vectors. This method performs left and up shifts that can be considered as moving the search window instead of the current block. Thus, mv_0 value does not change. Since the first mv_0 is considered as (0,0), AME formulae are converted to equations (6.5) and (6.6) for 4-parameter AME and equations (6.7) and (6.8) for 6-parameter AME, which decreases the hardware area.

$$mv_x = \frac{mv_{1_x}}{w} x - \frac{mv_{1_y}}{w} y \quad (6.5)$$

$$mv_y = \frac{mv_{1_y}}{w} x - \frac{mv_{1_x}}{w} y \quad (6.6)$$

$$mv_x = \frac{mv_{1_x}}{w} x - \frac{mv_{2_x}}{w} y \quad (6.7)$$

$$mv_y = \frac{mv_{1_y}}{w} x - \frac{mv_{2_y}}{w} y \quad (6.8)$$

Hardware friendly interweaved prediction for affine motion compensation is presented in [76]. In [77], a hardware architecture for the VVC affine motion compensation (MC) is proposed. In [78], two new hardware architectures for the VVC affine MC are developed. These proposed architectures process the 4×4 subblocks, generating the interpolated samples for the affine MC process of the VVC standard and generating four interpolated samples in parallel. In the hardware proposed in [79], four 4×4 subblocks are reconstructed in parallel, where the MCM technique is used to replace the multipliers with sum and shifts in the SMV generator and interpolation filters. In [80], a simplified AME algorithm and its ASIC hardware implementation are proposed.

In [74], low error approximate absolute difference (LAD_X) hardware is proposed. LAD_2 hardware and its two least significant bits (LSBs) are shown in Figure 6.5. LAD_X hardware comprises a subtractor, XOR gates, an adder, and OR gates. First, the difference (D) is obtained by subtracting inputs A and B. The sign bit of the difference (D[8]) is XOR'ed with the other eight bits of the difference (D[7:0]). Then, the sign bit (D[8]) is added to the least significant *X* bits. Therefore, instead of calculating 2's complement of the entire difference, 2's complement of its least significant *X* bits is calculated. This restricts carry propagation in the addition operation. Lastly, the MSB of the addition result (ECN[2]) is OR'ed with the other bits of the addition result.

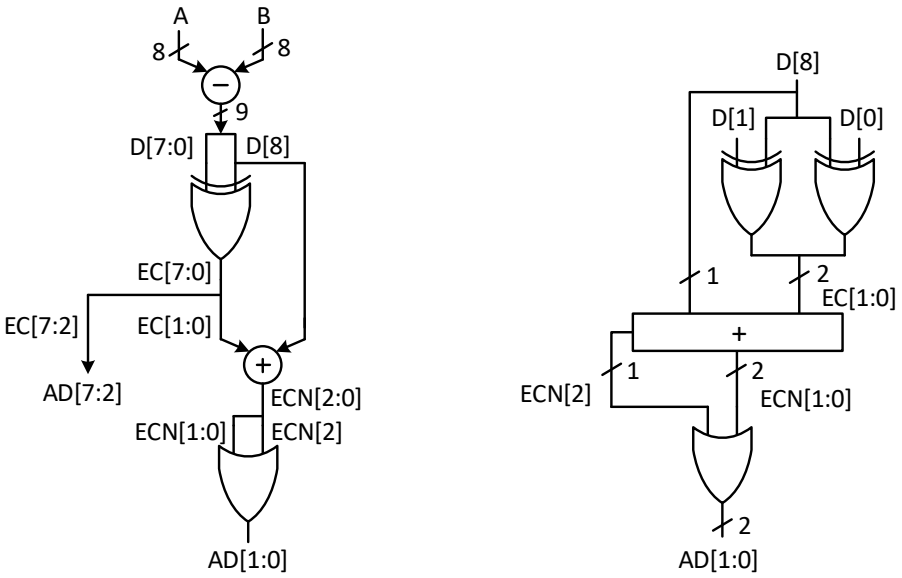

Figure 6.5 (a) LAD_2 hardware, (b) Two least significant bits of absolute difference in the LAD_2 hardware.

6.2 Proposed VVC Affine Motion Estimation Hardware

An approximate 4-parameter VVC AME hardware is proposed using a new approximate absolute difference (AD) hardware, approximate adder tree, and sub-sampling.

6.2.1 Proposed Approximate AD hardware

The pixels in a video are 8-bit integers in the range [0-255]. Sum of absolute differences (SAD) is a distortion metric which is commonly used in block matching ME and AME. The search location with the minimum SAD is selected as the best search location. To design the proposed approximate AD hardware, we assume that the AD value of the best search location is smaller than 32.

The proposed approximate AD hardware comprises a subtractor and some logic gates. The inputs of the hardware are two 8-bit unsigned integers shown as A[7:0] and B[7:0], and its output is a 5-bit unsigned integer shown as AD[4:0]. First, the inputs A and B are subtracted and the result is shown as D[8:0]. If $|D| < 32$, AD[4:2] is equal to XOR of D[8] (sign bit of the difference) with D[4:2] as shown in Table 6.1. If $|D| > 32$,

we make AD[4:0] as large as possible so AD[4:2] is as shown in Table 6.1. The cases corresponding to |D| > 32 are not expected to be the best search location in ME and AME. Therefore, this approximation does not cause much quality loss.

Figure 6.6 shows the Karnaugh maps for AD[4]. Accordingly, AD[4] is obtained as shown in equation (6.9) which is simplified as shown in equation (6.10).

Table 6.1 D[4:2] in the Proposed Approximate AD Hardware

	D[8]	D[7]	D[6]	D[5]	AD[4]	AD[3]	AD[2]
|D| < 32	0	0	0	0	D[4]	D[3]	D[2]
|D| > 32	0	0	0	1	1	1	1
	.	.	.	.	1	1	1
	.	.	.	.	1	1	1
	.	.	.	.	1	1	1
	1	1	1	0	1	1	1
|D| < 32	1	1	1	1	$\overline{D[4]}$	$\overline{D[3]}$	$\overline{D[2]}$

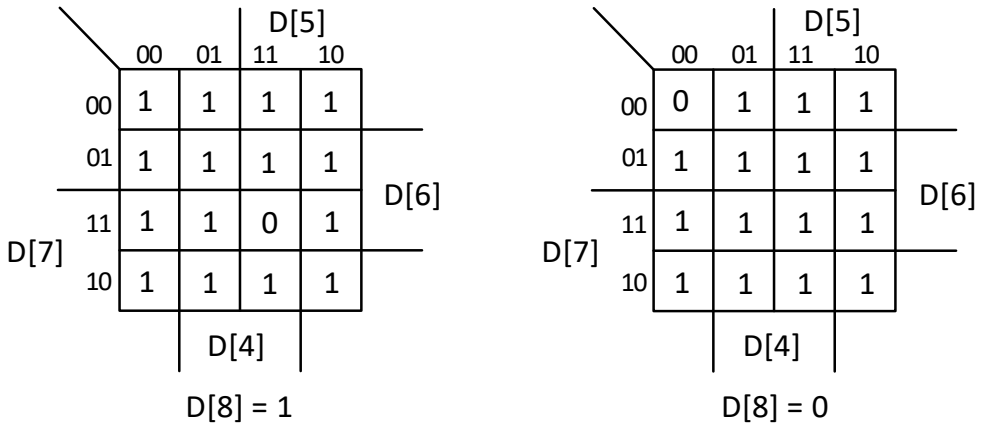

Figure 6.6 Karnaugh maps for AD[4] in the proposed approximate AD hardware.

$$AD[4] = \overline{D[8].D[7].D[6].D[5].D[4] + \overline{D[8]}.\overline{D[7]}.\overline{D[6]}.\overline{D[5]}.\overline{D[4]}} \quad (6.9)$$

$$AD[4] = \overline{D[8].D[7].D[6].D[5].D[4]}\,.(D[8] + D[7] + D[6] + D[5] + D[4]) \quad (6.10)$$

Similarly, AD[3] and AD[2] are obtained as shown in equations (6.11) and (6.12), respectively.

$$AD[3] = \overline{D[8].D[7].D[6].D[5].D[3]}\,.(D[8] + D[7] + D[6] + D[5] + D[3]) \quad (6.11)$$

$$AD[2] = \overline{D[8].D[7].D[6].D[5].D[2]}\,.(D[8] + D[7] + D[6] + D[5] + D[2]) \quad (6.12)$$

In the proposed approximate AD hardware, the two least significant bits of the absolute difference (AD[1:0]) are implemented similar to the LAD_2 hardware shown in Figure 6.5 so that the carry propagation of addition with 1 is restricted to only two bits. The hardware shown in Figure 6.5 is simplified using the truth table shown in Table 6.2.

Table 6.2 Truth Table for AD[1:0] in the LAD_2 Hardware

D[8]	D[1:0]	EC[1:0]	ECN[2:0]	AD[1:0]
0	00	00	000	00
0	01	01	001	01
0	10	10	010	10
0	11	11	011	11
1	00	11	100	11
1	01	10	011	11
1	10	01	010	10
1	11	00	001	01

Figure 6.7 shows the Karnaugh maps for AD[1]. Accordingly, AD[1] is obtained as shown in equation (6.13) which is simplified as shown in equation (6.14). Similarly, AD[0] is obtained as shown in equation (6.15).

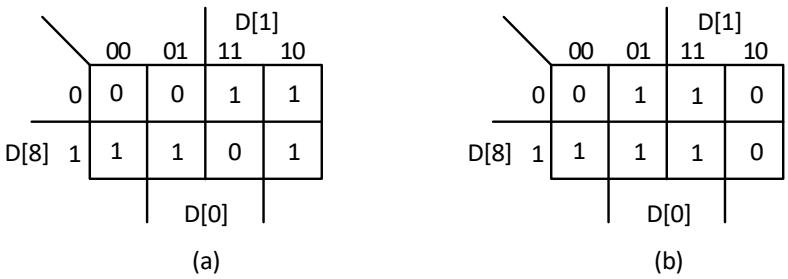

Figure 6.7 Karnaugh maps for (a) AD[1] and (b) AD[0] in the proposed approximate AD hardware.

$$AD[1] = D[8].\overline{D[1]} + \overline{D[8]}.D[1] + D[1].\overline{D[0]} \quad (6.13)$$

$$AD[1] = (D[8] \oplus D[1]) + D[1].\overline{D[0]} \quad (6.14)$$

$$AD[0] = D[0] + D[8].\overline{D[1]} \quad (6.15)$$

Figure 6.8 shows the proposed approximate AD hardware using equations (6.10), (6.11), (6.12), (6.14), and (6.15). In the cases that the absolute difference of two pixels A and B in the best search location is smaller than 32 (|A-B| < 32), the proposed approximate AD hardware has very low error. In the cases that the absolute difference

of two pixels A and B in the best search location is larger than 32 (|A-B| > 32), although the AD for these two pixels may not have low error, the SAD value can still have low error.

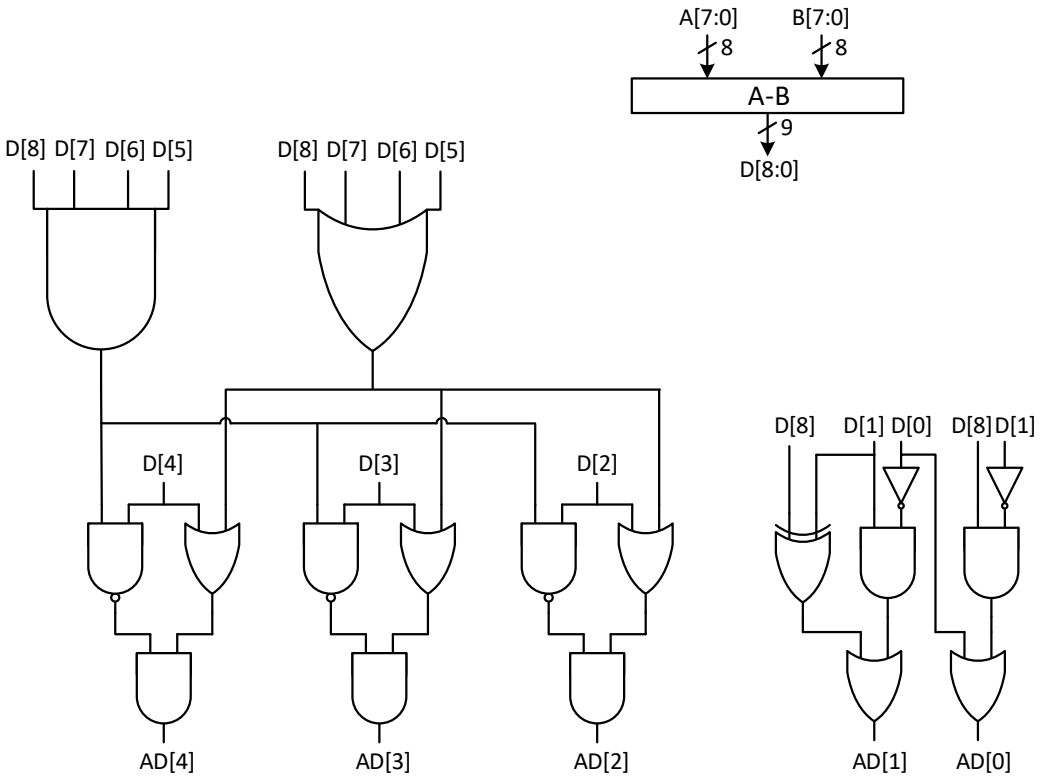

Figure 6.8 The proposed approximate AD hardware.

In ME and AME, the best search location is found using a distortion metric such as SAD. If some of the AD results have large errors, the SAD values can still have low error especially because large number of AD results are added in the adder tree to obtain the SAD value. For example, for the 64×64 block size, 4096 AD results are added to obtain the SAD value. Even though some of the AD results are calculated inaccurately, the best search location can still be found. Therefore, the proposed approximate AD hardware can be used for ME and AME. In the proposed approximate AD hardware, bit length of each AD is reduced from 8 to 5 which reduces area of both AD hardware and adder tree.

6.2.2 Approximate Adder Tree

In the VVC AME hardware proposed in [74], each AD is an 8-bit value and there are 12 stages in the adder tree, therefore the SAD bit length is 20.

In the proposed approximate VVC AME hardware, approximate adders are used in some of the stages in the adder tree to further decrease the bit length of the adders. Figure 6.9 shows the proposed approximate adder that is used in stage 4 of the adder tree, in which A[7:0] and B[7:0] are inputs and AS[7:0] is output. In the proposed approximate adder, the most significant bit (MSB) is removed. But before its removal, it is OR'ed with the three bits which have less significance than the MSB. Therefore, if the addition result is large such that MSB is one, before removing the MSB, we make the three less significant bits "111" and keep the rest of the bits as they are. This keeps the addition result large enough so that it does not affect the AME predictions much. In ME and AME, accuracy in calculation of the minimum SAD is much more important than the large SAD values. AD values used to calculate the minimum SAD corresponding to the best search location are not very large so that removal of MSB in some stages of adder tree does not cause much error.

In the proposed approximate VVC AME hardware, using the proposed approximate AD hardware, each AD is a 5-bit value and there are 12 stages in the adder tree. The proposed approximate adders are used in stages 4, 7, and 10 as shown with red color in Figure 6.10. In addition to the 3-bit reduction in bit length by using the proposed approximate AD hardware, there are three 1-bit reductions in bit length by using the proposed adder tree. Therefore, the SAD bit length in the proposed approximate VVC AME hardware is reduced from 20 to 14.

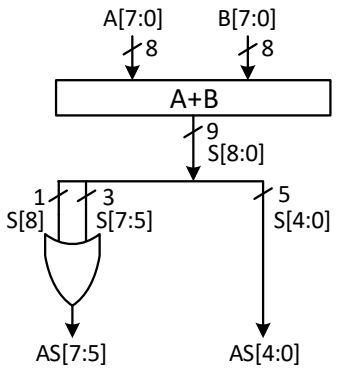

Figure 6.9 The approximate adder used in stage 4 of the proposed adder tree

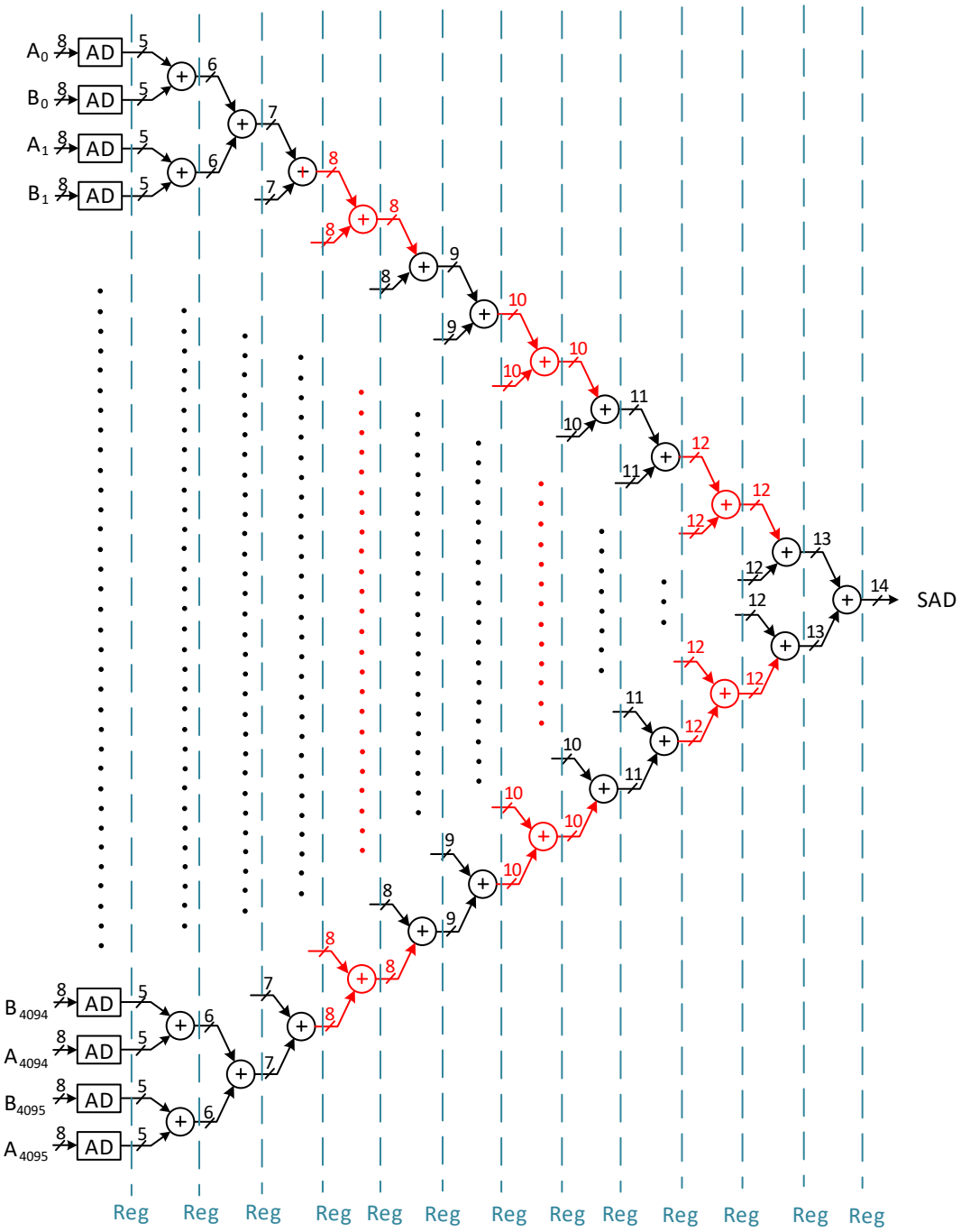

Figure 6.10 The proposed approximate adder tree

6.2.3 Sub-Sampling

Sub-sampling method is used in the second proposed approximate VVC AME hardware (proposed VVC AME (2)) to improve the performance and reduce the hardware area. In the proposed approximate hardware, only four of the pixels are used to calculate the SAD of a 4×4 sub-block instead of all the 16 pixels in the sub-block. In

Figure 6.11, the pixels that are used for SAD calculation in a 4×4 sub-block are shown as red squares. Using this sub-sampling, the bit length of the sub-block SAD and the output SAD become 7 and 12, respectively. The number of stages in the adder tree is reduced to 10. The proposed approximate adders are used in stages 4, 7, and 9 of the adder tree.

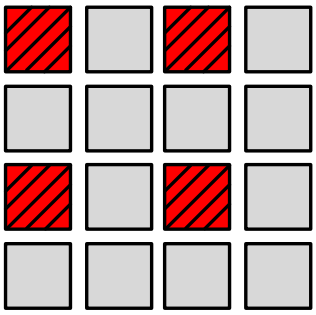

Figure 6.11 Sub-sampling in a 4×4 sub-block used in proposed VVC AME (2)

6.3 Implementation Results

We proposed two approximate VVC AME hardware (1) and (2). In both hardware, the pixel storage method proposed in [74] is used. Eight AME search locations are searched as in [74]. In proposed hardware (1), the proposed approximate AD hardware and proposed adder tree are used. In proposed hardware (2), in addition to the proposed approximate AD hardware and proposed adder tree, the sub-sampling method is also used.

Both proposed approximate VVC AME hardware are implemented using Verilog HDL. Verilog RTL codes are synthesized, placed and routed to a Virtex 7 FPGA using Xilinx Vivado 2020.1. In Table 6.3, the implementation results of the proposed hardware are compared to [74] which is the only FPGA implementation of the VVC AME in the literature. Proposed approximate VVC AME hardware (1) has 5% higher frequency and 7%, 20%, and 7% less LUT, FF, and BRAM, respectively, than the one in [74]. Proposed approximate VVC AME hardware (2) has 79% higher frequency and 71%, 57%, and 7% less LUT, FF, and BRAM, respectively, than the one in [74]. No DSP blocks are used in proposed hardware (2), because the multiplications in equations (6.5) and (6.6) are implemented with only adders and shifters.

Table 6.3 Implementation Results

	Frequency (MHz)	LUT	FF	BRAM	DSP
[74]	125.786	655741	252982	16	1920
Proposed VVC AME hardware (1)	131.579	610268	203680	15	1920
Proposed VVC AME hardware (2)	225.734	186392	106733	15	0

VVC AME in the proposed hardware is performed in 50188, 36876, and 16396 clock cycles for the blocks with 16×16, 32×32, and 64×64 sizes, respectively. Table 6.4 shows the frames per second (fps) of the proposed hardware for HD and FHD video resolutions. In the table, in hybrid case, 40% of the frame is processed with 64×64 block size, 35% of the frame is processed with 32×32 block size and 25% of the frame is processed with 16×16 block size. Both proposed hardware have higher performance than the one in [74].

Table 6.4 Number of frames per second (fps)

Block Size	[74]		Proposed hardware (1)		Proposed hardware (2)	
	HD	FHD	HD	FHD	HD	FHD
64×64	32.5 fps	15 fps	35.6 fps	15.8 fps	61.1 fps	27.1 fps
32×32	3.7 fps	1.6 fps	3.9 fps	1.7 fps	6.8 fps	3.0 fps
16×16	0.65 fps	0.29 fps	0.72 fps	0.32 fps	1.25 fps	0.55 fps
Hybrid	2 fps	0.91 fps	2.25 fps	1.76 fps	3.87 fps	3.02 fps

CHAPTER VII

CONCLUSIONS AND FUTURE WORK

In this thesis, approximate F2 VVC FI filters and VVC FI hardware implementing approximate F1 and F2 filters (MCMF1, BF2, MCMF2) are proposed. The proposed approximate VVC FI filters reduce computational complexity of VVC FI at the expense of very small quality loss. F2 filter causes slightly more quality loss than F1 filter. The proposed approximate VVC FI hardware have higher speed, smaller area, and up to 51% lower power consumption than the exact VVC FI hardware. Since VVC FI has higher computational complexity than HEVC FI, implementation results of the HEVC FI hardware are better than implementation results of the proposed approximate VVC FI hardware at the expense of lower quality. BF2 and MCMF2 hardware have higher speed, smaller area and lower power consumption than BF1 and MCMF1 hardware. However, they have slightly worse rate-distortion performance than BF1 and MCMF1 hardware. Therefore, MCMF1 hardware can be used in consumer electronics devices requiring high speed, small area, low power consumption and high quality. MCMF2 hardware can be used in consumer electronics devices requiring higher speed, smaller area, lower power consumption and slightly lower quality. Moreover, a novel VVC FI hardware using memory based constant multiplication for all PU sizes is proposed. Several optimizations are proposed to reduce memory size. The proposed VVC FI hardware can process 49 full HD (1920×1080) video frames per second. It has up to 9.4% less power consumption than VVC FI hardware in the literature.

In this thesis, decomposed coefficients technique is proposed for implementing HFI and VFI. Exact HFI hardware, exact VFI hardware, and approximate VFI hardware DCF1 and DCF2 are designed and implemented using the proposed technique. The proposed exact HFI and exact VFI hardware have higher performance, less area, and less power consumption than the best exact HFI and exact VFI hardware, respectively. The proposed approximate VFI hardware have the same performance, less area, and less power consumption than the best approximate VFI hardware. Therefore, the proposed hardware can be used in consumer electronics products which require real-time HEVC and VVC video encoder and decoder.

In this thesis, a new approximate constant multiplication technique is used to propose a new approximate HEVC 2D DCT for all transform unit (TU) sizes. In the proposed hardware, the approximate constant multiplication is used for multiplications with only the DCT coefficients that do not cause high average percentage error. So, it has less quality loss than the existing approximate HEVC 2D DCT hardware. In the proposed hardware, there are some common constant multiplications that are calculated once so that the number of multiplications is reduced. The proposed approximate HEVC 2D DCT hardware has less area, less power consumption, and higher performance than the existing HEVC 2D DCT hardware.

In this thesis, the first FPGA implementations of VVC FI and HEVC FME algorithms using an HLS tool in the literature are proposed. Novel FPGA implementations of HEVC 2D DCT algorithm using an HLS tool are proposed. The best proposed VVC FI HLS implementation can process 62 full HD video fps. It has higher performance than the manual VVC FI hardware implementations at the cost of larger area. The best proposed HEVC FME HLS implementation supports all the PU sizes, and in the worst case, can process 23 full HD video fps. The best proposed HEVC 2D DCT HLS implementation, in the worst case, can process 65 full HD fps.

In this thesis, we proposed an approximate VVC AME hardware using proposed approximate AD hardware, approximate adder tree, and sub-sampling. Using the proposed approximate hardware reduces VVC AME hardware area and improves its performance. Sub-sampling is used to further reduce the area and improve the performance. The proposed approximate VVC AME hardware has higher performance and smaller area than the best VVC AME hardware in the literature.

As future work, new VVC 2D DCT hardware can be proposed using the approximate constant multiplication method that is used in the proposed HEVC 2D DCT hardware. The proposed decomposed coefficients technique can be applied to VVC AME. Instead of full search algorithm, fast search algorithms can be used for VVC AME to achieve higher performance with smaller area.

BIBLIOGRAPHY

[1] V. Sze, M. Budagavi, and G. J. Sullivan, “High efficiency video coding (HEVC),” in *Integrated circuit and systems, algorithms and architectures*, vol. 39, p. 40. Berlin, Germany: Springer, 2014.

[2] “IMT traffic estimates for the years 2020 to 2030,” ITU, Geneva, Switzerland, Rep. M.2370-0, Jul. 2015. [Online]. Available: http:// www.itu.int/dms_pub/itu-r/opb/rep/R-REP-M.2370-2015-PDF-E.pdf

[3] B. Bross, J. Chen, S. Liu, and Y. K. Wang, “Versatile Video Coding (Draft 10),” *JVET-S2001*, Jul. 2020.

[4] J. Chen, Y. Ye, and S. Kim, “Algorithm Description for Versatile Video Coding and Test Model 5,” *JVET-N1002*, Jun. 2019.

[5] A. C. Mert, E. Kalali, and I. Hamzaoglu, “High performance 2D transform hardware for future video coding,” *IEEE Trans. Consum. Electron.*, vol. 63, no. 2., pp. 117-125, May 2017.

[6] H. Azgin, A. C. Mert, E. Kalali, and I. Hamzaoglu, “Reconfigurable intra prediction hardware for future video coding,” *IEEE Trans. Consum. Electron.*, vol. 63, no. 4., pp. 419-425, Nov. 2017.

[7] M. J. Garrido, F. Pescador, M. Chavarrías, P. J. Lobo, and C. Sanz, “A 2-D Multiple Transform Processor for the Versatile Video Coding Standard,” *IEEE Trans. Consum. Electron.*, vol. 65, no. 3, pp. 274-283, Aug. 2019.

[8] A. Henkel, I. Zupancic, B. Bross, M. Winken, H. Schwarz, D. Marpe, and T. Wiegand, “Alternative Half-sample Interpolation Filters for Versatile Video Coding,” in *Proc. IEEE Int. Conference on Acoustics, Speech and Signal Processing*, May 2020, pp. 2053-2057.

[9] High Efficiency Video Coding, ITU-T Rec. H.265 and ISO/IEC 23008-2 (HEVC), *ITU-T and ISO/IEC*, Apr. 2013.

[10] F. Pescador, M. Chavarrias, M. J. Garrido, E. Juarez, and C. Sanz, "Complexity analysis of an HEVC decoder based on a digital signal processor," *IEEE Trans. Consum. Electron.*, vol. 59, no. 2, pp. 391- 399, May 2013.

[11] E. Kalali, E. Ozcan, O. M. Yalcinkaya, and I. Hamzaoglu, "A low energy HEVC inverse transform," *IEEE Trans. Consum. Electron.*, vol. 60, no. 4, pp. 754-761, Nov. 2014.

[12] E. Kalali, A. C. Mert, and I Hamzaoglu, "A computation and energy reduction technique for HEVC discrete cosine transform," *IEEE Trans. Consum. Electron.*, vol. 62, no. 2, pp. 166-174, May 2016.

[13] H. Azgin, E. Kalali, and I. Hamzaoglu, "A computation and energy reduction technique for HEVC intra prediction," *IEEE Trans. Consum. Electron.*, vol. 63, no. 1, pp. 36-43, Feb. 2017.

[14] A. Singhadia, M. Mamillapalli, and I. Chakrabarti, "Hardware-Efficient 2D-DCT/IDCT Architecture for Portable HEVC-Compliant Devices," *IEEE Trans. Consum. Electron.*, vol. 66, no. 3, pp. 203-212, Aug. 2020.

[15] M. T. Pourazad, C. Doutre, M. Azimi, and P. Nasiopoulos, "HEVC: The new gold standard for video compression: How does HEVC compare with H. 264/AVC?," *IEEE consum. Electron. magazine*, vol. 1, no. 3, pp. 36-46, 2012.

[16] J. Lainema, F. Bossen, W.J. Han, J. Min and K. Ugur, "Intra Coding of the HEVC Standard", *IEEE Trans. on Circuits and Systems for Video Technology*, vol.22, no.12, pp.1792-1801, Dec. 2012.

[17] T. Biatek, V. Lorcy, P. Castel, P. Philippe, "Low-Complexity Adaptive Multiple Transform for Video Coding", *Proc. Data Compression Conference*, April 2016.

[18] H. Mahdavi, H. Azgin, and I. Hamzaoglu, "Approximate versatile video coding fractional interpolation filters and their hardware implementations," *IEEE Trans. on Consum. Electron.*, vol. 67, no. 3, pp. 186-194, 2021, Aug. 2021, doi: 10.1109/TCE.2021.3107460.

[19] H. Mahdavi and I. Hamzaoglu, "A VVC fractional interpolation hardware using memory based constant multiplication," in *IEEE Int. Conf. on Consum. Electron. (ICCE)*, pp. 1-5, Jan. 2021.

[20] H. Mahdavi and I. Hamzaoglu, "An efficient HEVC fractional interpolation hardware," in *IEEE Int. Conf. on Consum. Electron. (ICCE)*, pp. 1-4, Jan. 2021.

[21] H. Azgin, E. Kalali, and I. Hamzaoglu, “A Novel Approximate Constant Multiplier and HEVC Discrete Cosine Transform Case Study,” in *IEEE 10th Int. Conf. on Consumer Electronics (ICCE-Berlin)*, pp. 1-6, Nov. 2020.

[22] I. Hamzaoglu, H. Mahdavi, and E. Taskin, “FPGA Implementations of VVC Fractional Interpolation Using High-Level Synthesis,” in *IEEE Int. Conf. on Consum. Electron. (ICCE)*, pp. 1-6, 2022, doi: 10.1109/ICCE53296.2022.9730363.

[23] J. Chen, Y. Ye, and S. Kim, “Algorithm description for versatile video coding and test model 5,” *Joint Video Experts Team (JVET) of ITU-T SG*, 2019.

[24] K. McCann, B. Bross, W. J. Han, I. K. Kim, K. Sugimoto, and G. J. Sullivan, “High Efficiency Video Coding (HEVC) Test Model 15 (HM 15) Encoder Description,” *JCTVC-Q1002*, June 2014.

[25] H. Azgin, E. Kalali, and I. Hamzaoglu, “An Approximate Versatile Video Coding Fractional Interpolation Hardware,” in *Proc. IEEE Int. Conference on Consumer Electronics*, Jan. 2020, pp. 1-4, doi: 10.1109/ICCE46568.2020.9042986.

[26] Y. Voronenko and M. Püschel, “Multiplierless constant multiplication,” *ACM Trans. Algorithms*, vol. 3, no. 2, May 2007.

[27] A. Henkel, I. Zupancic, B. Bross, M. Winken, H. Schwarz, D. Marpe, and T. Wiegand, “Alternative Half-sample Interpolation Filters for Versatile Video Coding,” in *Proc. IEEE Int. Conference on Acoustics, Speech and Signal Processing*, May 2020, pp. 2053-2057.

[28] H. Azgin, A. C. Mert, E. Kalali, and I. Hamzaoglu, “A Reconfigurable Fractional Interpolation Hardware for VVC Motion Compensation,” in *Proc. Euromicro Conference on Digital System Design*, Aug. 2018, pp. 99-103, doi: 10.1109/DSD.2018.00030.

[29] A. C. Mert, E. Kalali, and I. Hamzaoglu, “A Low Power Versatile Video Coding (VVC) Fractional Interpolation Hardware,” in *Proc. Int. Conference on Design and Architectures for Signal and Image Processing*, Oct. 2018, pp. 43-47, doi: 10.1109/DASIP.2018.8597040.

[30] E. Kalali and I. Hamzaoglu, “A low energy HEVC sub-pixel interpolation hardware,” in *Proc. IEEE Int. Conference on Image Processing*, Oct. 2014, pp. 1218-1222, doi: 10.1109/ICIP.2014.7025243.

[31] C. Y. Lung and C. A. Shen, “A high-throughput interpolator for fractional motion estimation in high efficient video coding (HEVC) systems,” in *Proc. IEEE Asia Pacific Conference on CAS*, Nov. 2014, pp. 268-271, doi: 10.1109/APCCAS.2014.7032771.

[32] G. Pastuszak and M. Trochimiuk, “Algorithm and architecture design of the motion estimation for the H.265/HEVC 4K-UHD encoder,” *Journal of Real-Time Image Process.*, vol. 12, no. 2, pp. 517-529, Aug. 2016, doi: https://doi.org/10.1007/s11554-015-0516-4.

[33] C. M. Diniz, M. Shafique, S. Bampi, and J. Henkel, “A reconfigurable hardware architecture for fractional pixel interpolation in high efficiency video coding,” *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 34, no. 2, pp. 238-251, Feb. 2015, doi: 10.1109/TCAD.2014.2384517.

[34] E. Kalali, and I. Hamzaoglu, “Approximate HEVC Fractional Interpolation Filters and Their Hardware Implementations,” *IEEE Trans. Consum. Electron.*, vol. 64, no. 3, pp. 285-291, Aug. 2018.

[35] F. Bossen, “Common test conditions and software reference configurations,” *JCTVC-I1100*, May 2012.

[36] P. K. Meher, “LUT Optimization for Memory-Based Computation”, *IEEE Transactions on Circuits and Systems-II: Express Briefs*, vol. 57, no. 4, pp. 285-289, Apr. 2010.

[37] P. K. Meher, “New Approach to Look-Up-Table Design and Memory- Based Realization of FIR Digital Filter”, *IEEE Transactions on Circuits and Systems-I: Regular Papers* , vol. 57, no. 3, pp. 592-603, Mar. 2010.

[38] A. C. Mert, E. Kalali, and I. Hamzaoglu, “An HEVC fractional interpolation hardware using memory based constant multiplication,” *IEEE Int. Conference on Consumer Electronics*, pp. 1-5, Jan. 2018, doi: 10.1109/ICCE.2018.8326312.

[39] P. Sjövall, M. Rasinen, A. Lemmetti, and J. Vanne, “High-Level Synthesis Implementation of an Accurate HEVC Interpolation Filter on an FPGA,” in *IEEE Nordic Circuits and Syst. Conf. (NorCAS)*, Oslo, Norway, 2021, pp. 1-7, doi: 10.1109/NorCAS53631.2021.9599653.

[40] S. Mittal, “A survey of techniques for approximate computing,” *ACM Computing Surveys*, vol. 48, no. 4, May 2016.

[41] Q. Xu, T. Mytkowicz, and N. S. Kim, “Approximate computing: A survey,” *IEEE Design & Test*, vol. 33, no. 1, pp. 8-22, Feb. 2016.

[42] T. Nomani, M. Mohsin, Z. Pervaiz, and M. Shafique, “xUAVs: Towards efficient approximate computing for UAVs—Low power approximate adders with single LUT Delay for FPGA-based aerial imaging optimization,” *IEEE Access*, vol. 8, pp. 102982-102996, June 2020.

[43] E. Kalali and I. Hamzaoglu, “An approximate HEVC intra angular prediction hardware,” *IEEE Access*, vol. 8, pp. 2599-2607, 2020.

[44] H. Mahdavi and S. Timarchi, “Improving architectures of binary signed-digit CORDIC with generic/specific initial angles,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 7, pp. 2297-2304, 2020.

[45] A. C. Mert, H. Azgin, E. Kalali, and I. Hamzaoglu, “Novel approximate absolute difference hardware,” in *Euromicro Conf. on Digital System Design*, Aug. 2019.

[46] G. Zervakis, H. Amrouch, and J. Henkel, “Design automation of approximate circuits with runtime reconfigurable accuracy,” *IEEE Access*, vol. 8, pp. 53522-53538, Mar. 2020.

[47] S. Xu and B. C. Schafer, “Toward self-tunable approximate computing,” *IEEE Trans. on VLSI Systems*, vol. 27, no. 4, pp. 778-789, Apr. 2019.

[48] Y. Kim, Y. Zhang, and P. Li, “Energy efficient approximate arithmetic for error resilient neuromorphic computing,” *IEEE Trans. on VLSI System*s, vol. 23, no. 11, pp. 2733-2737, Nov. 2015.

[49] A. Raha, H. Javakumar, and V. Raghunathan, “Input-based dynamic reconfiguration of approximate arithmetic units for video encoding,” *IEEE Trans. on VLSI Systems*, vol. 24, no. 3, pp. 846-857, Mar. 2016.

[50] W. Ahmad, B. Ayrancioglu, and I. Hamzaoglu, “Low Error Efficient Approximate Adders for FPGAs,” *IEEE Access*, vol. 9, pp. 117232-117243, Aug. 2021.

[51] A. Momeni, J. Han, P. Montuschi, and F. Lombardi, “Design and analysis of approximate compressors for multiplication,” *IEEE Trans. on Computers*, vol. 64, no. 4, pp. 984-994, Apr. 2015.

[52] I. Hammad, L. Li, K. El-Sankary, and W. M. Snelgrove, “CNN inference using a preprocessing precision controller and approximate multipliers with various precisions,” *IEEE Access*, vol. 9, pp. 7220-7232, Jan. 2021.

[53] R. Conceição, J. C. de Souza Jr, R. Jeske, B. Zatt, M. Porto, and L. Agostini, “Low-cost and high-throughput hardware design for the hevc 16x16 2-d dct transform,” *Journal of Integrated Circuits and Systems*, vol. 9, no. 1, pp. 25-35, 2014.

[54] M. Chen, Y. Zhang, and C. Lu, “Efficient architecture of variable size HEVC 2D-DCT for FPGA platforms,” *AEU-International Journal of Electronics and Communications*, vol. 73, pp. 1-8, Mar. 2017.

[55] H. Loukil and N. Masmoudi, “A novel architecture design for VLSI implementation of integer DCT in HEVC standard,” *Multimedia Tools and Applications*, vol. 79, no. 33, pp. 23977-23993, Sep. 2020.

[56] A. Shabani, S. Timarchi, and H. Mahdavi, “Power and area efficient CORDIC-Based DCT using direct realization of decomposed matrix,” *Microelectronics Journal*, vol. 91, pp. 11-21, 2019.

[57] Z. Vasicek and V. Mrazek, “Towards low power approximate DCT architecture for HEVC standard,” in *IEEE Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1576-1581, 2017.

[58] M. Masera, M. Martina, and G. Masera, “Adaptive approximated DCT architectures for HEVC,” *IEEE Trans. on Circuits and Systems for Video Technology*, vol. 27, no. 12, pp. 2714-2725, July 2016.

[59] M. Jridi, A. Alfalou, and P. K. Meher, “Efficient approximate core transform and its reconfigurable architectures for HEVC,” *Journal of Real-Time Image Processing*, vol. 17, no. 2, pp. 329-339, Apr. 2020.

[60] G. Bjontegaard, “Calculation of average PSNR differences between RD-curves,” *13th Video Coding Experts Group Meeting*, 2001.

[61] R. Saha, P. P. Banik, and K. D. Kim, “Conversion of LDR image to HDR-like image through high-level synthesis tool for FPGA implementation,” in *IEEE Int. Conf. on Consumer Electronics (ICCE)*, pp. 1-2, Jan. 2018.

[62] H. S. Lee and H. W. Jeon, “Comparison between HLS and HDL image processing in FPGAs,” in *IEEE Int. Conf. on Consumer Electronics-Asia (ICCE-Asia)*, pp. 1-2, Nov. 2020.

[63] S. Lahti, P. Sjövall, J. Vanne, and T. D. Hämäläinen, “Are we there yet? A study on the state of high-level synthesis,” *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 5, pp. 898-911, 2018.

[64] M. Kammoun, A. B. Atitallah, K. M. A Ali, and R. B. Atitallah, “Case study of an HEVC decoder application using high-level synthesis: intraprediction, dequantization, and inverse transform blocks,” *Journal of Electronic Imaging*, vol. 28, no. 3, pp. 033010, 2019.

[65] A. Sengupta, “Evolution of the IP design process in the semiconductor/EDA industry [hardware matters],” *IEEE Consum. Electron. Magazine*, vol. 5, no. 2, pp. 123-126, 2016.

[66] J. S. León, C. S. Cardenas, and E. V. Castillo, “A high parallel HEVC Fractional Motion Estimation architecture,” in *IEEE ANDESCON*, pp. 1-4, 2016.

[67] D. Ding, X. Ye, and S. Wang, “1/2 and 1/4 pixel paralleled FME with a scalable search pattern for HEVC ultra-HD encoding,” in *IEEE 16th Int. Conf. on Communication Technology (ICCT)*, pp. 278-281, 2015.

[68] A. C. Mert, E. Kalali, and I. Hamzaoglu, “Low complexity HEVC sub-pixel motion estimation technique and its hardware implementation,” in *IEEE 6th Int. Conf. on Consum. Electron.-Berlin (ICCE-Berlin)*, pp. 159-162, 2016.

[69] W. Penny, G. Correa, L. Agostini, D. Palomino, M. Porto, G. Nazar, and B. Zatt, “Low-power and memory-aware approximate hardware architecture for fractional motion estimation interpolation on HEVC,” in *IEEE Int. Symposium on Circuits and Systems (ISCAS)*, pp. 1-5, 2020.

[70] E. Kalali and I. Hamzaoglu, “FPGA implementations of HEVC inverse DCT using high-level synthesis,” in *Conf. on Design and Architectures for Signal and Image Processing (DASIP)*, pp. 1-6, 2015.

[71] UG902, “Vivado Design Suite User Guide: High-Level Synthesis,” Oct. 2019.

[72] L. Li, H. Li, Z. Lv, and H. Yang, “An affine motion compensation framework for high efficiency video coding,” in *IEEE Int. Symp. on Circuits and Systems (ISCAS)*, pp. 525-528, 2015.

[73] Y. J. Choi, D. S. Jun, W. S. Cheong, and B. G. Kim, “Design of efficient perspective affine motion estimation/compensation for versatile video coding (VVC) standard,” *Electronics*, vol. 8, no. 9, pp. 993, 2019.

[74] B. Ayrancıoğlu, Approximate computing based video compression hardware. Diss. 2022.

[75] B. Bross, J. Chen, J. R. Ohm, G. J. Sullivan, and Y. K. Wang, “Developments in international video coding standardization after avc, with an overview of versatile video coding (vvc),” *Proceedings of the IEEE*, vol. 109, no. 9, pp. 1463-1493, 2021.

[76] T. Fu, K. Zhang, L. Zhang, S. Wang, and S. Ma, “Hardware friendly interweaved prediction for affine motion compensation,” in *IEEE Picture Coding Symposium (PCS)*, pp. 1-5, 2021.

[77] M. M. Muñoz, D. Maass, M. Perleberg, L. Agostini, and M. Porto, “Hardware Design for the Affine Motion Compensation of the VVC Standard,” in *IEEE 14th Latin America Symposium on Circuits and Systems (LASCAS)*, pp. 1-4, 2023.

[78] M. M. Muñoz, D. Maass, M. Perleberg, L. Agostini, and M. Porto, “Efficient Hardware Design for the VVC Affine Motion Compensation Exploiting Multiple Constant

Multiplication," in *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pp. 1-6, 2023.

[79] M. M. Muñoz, D. Maass, M. Perleberg, L. Agostini, G. Correa, and M. Porto, "4K UHD@ 60fps Design For The VVC Affine Motion Estimation Reconstructor," in *36th SBC/SBMicro/IEEE/ACM Symposium on Integrated Circuits and Systems Design (SBCCI)*, pp. 1-6, 2023.

[80] C. Taranto, "Simplified affine motion estimation algorithm and architecture for the versatile video coding standard," PhD diss., Politecnico di Torino, 2022.