HIGH-THROUGHPUT AREA-EFFICIENT INTEGER TRANSFORMS FOR VIDEO CODING

DO THI THU TRANG

(B.Eng. (Hons.), M.Sc., Hanoi University of Technology)

A THESIS SUBMITTED FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

NATIONAL UNIVERSITY OF SINGAPORE

2013

Declaration

I hereby declare that the thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

[illegible]

Do Thi Thu Trang
24th January, 2013

Acknowledgements

The research presented in this dissertation has been carried out during the years 2007-2012 at the Department of Electrical and Computer Engineering, National University of Singapore.

Many people, in one or another way, have helped to make this dissertation a reality. I can only mention a few of them here.

First and foremost, I wish to express my deep gratitude to my supervisor, Dr. HA Yajun, for letting me be one of your students, and guiding me along my Ph.D. study. Without your scientific guidance and your encouragement, this dissertation would not have been possible. Thanks for your understanding, patience and belief in me.

I would like to sincerely thank my co-supervisor, Dr. LE Minh Thinh, who first sparked my research interest in the field of Video codec and ASIC/ASIP design. Thanks for your guidance and unlimited support in the first half of my PhD. journey and thanks for your encouragement even when you have left Singapore.

I would also like to thank the thesis examiners, Prof. Ashraf A. Kassim, A/Prof. Bharadwaj Veeravalli from Department of Electrical and Computer Engineering, National University of Singapore and Prof. Henk Corporaal from Department of Electrical Engineering, Eindhoven University of Technology for their time and valuable comments.

Special thanks to all VLSI lab-mates, especially ZHAO Wenfeng, WU Tong, PAN Rui, LIU Xiayun and JIANG Xi for your supports, suggestions and helpful discussions for my tape-out. Thanks for the friendship and sharing from CHEN Xiaolei, RIZWAN Syed, NG Kian Ann and CHUA Dingjuan. Thanks to TIAN Xiaohua and NGO Tuan Nghia, the colleagues in my research groups for the co-operation and helpful discussions from the early days.

Thanks for the supports of the Signal Processing and VLSI Design Laboratory Supervisor - A/P XU Yong Ping, CAD supporter - Mdm. ZHENG Huan Qun and lab officer - Mr. TEO Seow Miang.

I would like to thank my EE2024 teaching colleagues, A/P THAM Chen Khong and PANICKER Rajesh and NIU Tianfeng, for your understanding and supportiveness. Thanks Rajesh for sharing not only as a colleague but also as a friend.

I would like to acknowledge the financial, academic and technical supports from the ASEAN University Network and the Southeast Asia Engineering Education Development Network Project (AUN/SEED-Net). I believe you will be successful in promoting human resource development in engineering in ASEAN.

I wish to express my appreciation to Dominic ANG, my first and best Singaporean friend, who have helped us since the first day we came to Singapore. I still remember the night you stayed with us in NUH when my son was sick. Thanks for helping and always being with us during our hard times.

Last but not least, I would like to thank all of our family members for their love, encouragement, and support. My husband, NGUYEN Phu Binh, who loves and cares for me, motivates and inspires me with endless support. My precious and cutesy son, NGUYEN Minh Duc, whose lovely smiles, sweet kisses and soft and fragrant hugs always refresh, recharge, bring me joys and give me strengths to overcome all hardships. My parents, parents-in-law and my brother-in-law with their love, encouragement and supports.

This dissertation is dedicated to my parents, DO The Anh and PHAM Thi Minh Nu, who raised me up with their unconditional love. You smile when I am happy. You frown when I am sad. You encourage me when I face challenges. You are always there with endless supports whenever I need. I love you.

Contents

Summary

Integer transform, an approximation of the Discrete Cosine Transform (DCT) to avoid the mismatch problem and reduce the computational complexity, is one of the most important coding tools in the latest video compression standard, H.264/AVC, and the emerging High Efficiency Video Coding (HEVC). The H.264/AVC high profiles and HEVC target high and beyond-high definition videos, respectively. As a result, their integer transform designs are challenging to high throughputs, especially for large transform sizes. On the other hand, area efficiency must be taken into account to reduce costs for hardware designs.

These challenges are addressed at *algorithm* and *architecture* levels in the literature. In H.264, at the *algorithm* level, to achieve high throughput, most of the reported designs follow the fast and low-cost transform algorithms introduced by H.264 developers. At the *architecture* level, some designs input, process and output a data block at once, leading to unpractical wide I/O buses. Some designs deploy pipelining mechanisms among operating processes. This reduces total delays and increases throughputs. However, these designs also have very large bus widths. To achieve area efficiency at the *architecture* level, hardware-sharing can be done among (1) coefficient computations, (2) the forward/inverse transforms, or (3) the forward and inverse transforms. Technique (1) leads to a very low throughput. Technique (3) is not applicable in encoder and decoder scenarios. The reported units deploying technique (2) have yet supported all the transform types.

In HEVC, at the *algorithm* level, Partial Butterfly algorithms are applied in the test models (HMs). These scalar-multiplication-containing algorithms are very complex to be implemented in hardware to achieve high throughput and area efficiency. The reported fast algorithms are based on the H.264 transforms, which are not extendable to larger-than-H.264-transform sizes. At the *architecture* level, techniques to improve design throughput have not reported yet. For area efficiency, hardware-sharing among the coefficient computations is reported. However, like with H.264, this technique leads to a very low throughput.

We address in this dissertation the challenges on transform designs for the H.264/AVC high profiles and HEVC, and present four high-throughput and area-efficient integer transform designs under the condition of reasonable I/O bus widths. In addition, we also introduce techniques for portable integer transform designs.

First, we address the question of how to develop portable integer transform designs. Two H.264/AVC portable inverse transform architectures supporting all the types of the inverse transforms and rescaling function are proposed with preferable high throughputs under the reasonable bus-width constraints. In the first design, a shared-hardware unit among all the inverse transforms is proposed to target area-efficient challenge. For portability, the Wishbone shared bus system is deployed. The second design continues to employ the transform unit from the first design for area efficiency. For a high throughput, the design employs two transform units with a pipelining mechanism among all the input, output and transform processes. The Wishbone crossbar-switch bus system is used for portability and pipeline support.

Simulation results show that the second design can deliver a higher throughput than other reported designs. The resulting circuit area is considerably minimal compared to the designs in its class thanks to the proposed transform unit.

Second, we address the H.264 transform design challenges at the *architecture* level by introducing a very-high-throughput forward and inverse transform architecture under the bus width constraint. The proposed architecture supports all the types of the forward and inverse transforms together with quantization and rescaling functions. For area efficiency, the design introduces a novel shared forward unit and reuses the inverse unit from the first portable design. A novel series of input and output buffers is proposed to balance the data transferring load in I/O bus. This can double the processing speed when integrating two transform units without increasing the bus width. By deploying a pipelining mechanism among all the input, output, transform and additional quantization processes, the architecture can out-perform other reported designs in terms of throughput and area efficiency.

Third, we provide performance-cost analyses and propose a novel performance-cost metric for comprehensive comparisons for H.264 integer transform designs. Some designs deploy extremely wide buses but their designers were still able to claim the area efficiencies. This is because the wire area is not counted in the comparison metrics. The proposed metric is defined as the ratio of data throughput over the design cost, which includes power, area, and delay, and issues associated with interconnections in sub-micron design.

Then, to address the HEVC transform design challenges at the *algorithm* level, we propose a novel optimization method to achieve fast and low-cost implementations for scalar-multiplication-containing algorithms in general and Partial Butterfly algorithms in particular. The method includes a multiplication-to-addition conversion step and three levels of optimization: complexity, timing and resource optimizations. Experimental results show that both the running time and cost of the implementations generated by the proposed method are about 90% less than those of the original algorithms. The method is computationally feasible to be applied to all the Partial Butterfly algorithms from 4×4 to 32×32 in size.

Next, we propose a series of novel hardware-oriented HEVC 4×4 and 8×8 fast and low-cost forward transform algorithms. For a high speed, the algorithms are developed using the implementations obtained by applying the proposed optimization method to the Partial Butterfly algorithms. Simultaneously, for the area efficiency, the strategy of embedding the small transforms into the large transforms is deployed. The running time and cost of the proposed algorithms are about 80% less than those of the Partial Butterfly algorithms. Compared to the reported algorithms, the proposed algorithms are faster and their costs are lower. Similarly, fast and low-cost transform algorithms for larger sizes can be also developed.

Finally, we develop, implement and fabricate a high-throughput and area-efficient transform architecture for HEVC based on the proposed algorithms. To address the design challenges at the *architecture* level for a very tight constraint on I/O pin count, a number of techniques are proposed, including (1) multi-cycle adder design; (2) multi-stage register design; (3) pipelining mechanism among the addition/subtractions; (4) resource binding; (5) micro-code and control signal optimization; and (6) fully pipelining mechanism among input, output and transform processes. Thanks to the proposed techniques, the architecture can support 4×4 and 8×8 transforms for up to quad-full high definition (QFHD) videos at the progressive scan frequency of 30 Hz. This is much larger than the maximum resolution supported by the reported HEVC transform architectures. The power consumed by the proposed architecture is also much less than that of other designs.

List of Tables

List of Figures

List of Acronyms

1

1P6M	1-Poly-6-Metal.

A

AD	Addition Depth.
ALU	Arithmetic Logic Unit.
AMBA	Advanced Micro-controller Bus Architecture by ARM.
ARM	Advanced RISC Machines company.
AS	Addition/Subtraction.
ASIC	Application-Specific Integrated Circuit.
ASIP	Application Specific Instruction-Set Processor.
AVC	Advanced Video Coding.
AVS	Audio Video Standard.

B

BA	Block Adder or Addition Block.
BA	Bit Array.
BFHD	Beyond-Full High Definition.
BO	Best Order.

C

CA	Conversion Array.
CABAC	Context-based Adaptive Binary Arithmetic Coding.
CAVLC	Context-based Adaptive VLC.
CB	Coding Block.
Chroma	Chrominance.
CK	Computing Kernel.
CL	Length array of 1/0 chains in Bit Array.
CMOS	Complementary Metal Oxide Semiconductor.
CO	Complexity Optimization.
COR	Common Operation Region.
CORAS	Common Operation Region label for each Addition/Subtraction AS.
CS	Least Significant bit array of 1/0 chains in Bit Array BA.

CTB	Coding Tree Block.
CTU	Coding Tree Unit.
CU	Coding Unit (in HEVC).
CU	Control Unit (in hardware architecture).

D

DC coefficient	Coefficient with zero frequency in both dimensions.
DCT	Discrete Cosine Transform.
DMA	Direct Memory Access.
DMAC	Direct Memory Access Controller.
DSP	Digital Signal Processing.
DST	Discrete Sine Transform.
DTUA	Data Throughput Per Unit Area.
DVD	Digital Versatile Disc.
DWT	Discrete Wavelet Transform.

E

enas	Enable Adder/Subtractor.

F

FDCT	Forward Discrete Cosine Transform.
FFT	Fast Fourier Transform.
FHD	Full High Definition.
FHT	Forward Hadamard Transform.
FIT	Forward Integer Transform.
FPGA	Field-Programmable Gate Array.
FQ	Forward Quantization.
FRExt	Fidelity Range Extensions.
FSM	Finite-State Machine.

H

HD	High Definition.
HDL	Hardware Description Language.
HDTV	High Definition Television.
HEVC	High Efficiency Video Coding.
HM	HEVC Test Model.
HT	Hadamard Transform.

I

IB	Input Buffer.
IBM	International Business Machines Corporation.
IC	Integrated Circuit.
IDCT	Inverse Discrete Cosine Transform.
IEC	International Electrotechnical Commission.
IEEE	Institute of Electrical and Electronics Engineers.
IHT	Inverse Hadamard Transform.
IIT	Inverse Integer Transform.
I/O	Input/Output.
I/OD	Input/Output Delay.
IP	Intellectual Property.
IR	Instruction Register.
ISO	International Standards Organization.
IT	Integer Transform.
ITU	International Telecommunication Union.

J

JCT-VC	Joint Collaborative Team on Video Coding.
JM	JVT H.264/AVC Test Model (H.264/AVC Reference Software).
JVT	Joint Video Team.

L

LL	Low Leakage.
LR	Long Register.
LRL	Least Significant Part of Long Register LR.
LRM	Most Significant Part of Long Register LR.
Luma	Luminance.
LUT	Lookup Table.

M

MAC	Multiplication-to-Addition Conversion.
MACR	Multiplication-to-Addition Conversion Result.
MB	Macroblock.
MD	Martuza and Wahid (2012)'s design.
MF	Multiplication Factor.

MPEG	Moving Picture Experts Group.
MUX	Multiplexer.

N

NAL	Number of ASs used in levels.
NI	Number of Input of Number of non-zero elements in conversion array CA.
NIL	Number of Inputs to Levels.
NMOS	n-channel MOSFET (metal-oxide-semiconductor field-effect transistor).

O

OB	Output Buffer.

P

PB	Partial Butterfly.
PC	Program Counter.
PCAS	Performance-Cost Analysis Software.
PCB	Printed Circuit Board.
PCM	Performance-Cost Metric.

PCMG	Technology-hidden Peformance-Cost Metric for SoC IP blocks having a small number of pins.
PCMK	Technology-hidden Peformance-Cost Metric for SoC IP blocks having a large number of pins.
PD	Processing Delay.
PF	Position Factor.
PMOS	p-channel MOSFET (metal-oxide-semiconductor field-effect transistor).
ppc	pixel per cycle.
pps	pixel per second.
PS	Positions of non-zero elements in conversion array CA.
PS	Permutation Set.
PSNR	Peak Signal-to-Noise Ratio.

Q

Q	Quantization.
QFHD	Quad-Full High Definition.
QP	Quantization Parameter.

R

RAM	Random-Access Memory.

RLC	Run Length Coding.
RO	Resource Optimization.
RT	Run time.
RTL	Register-Transfer Level.
RVT	Regular Voltage Threshold.

S

SoC	System-on-Chip.
SP	Selected Permutation.
SPM	Stored Program Machine.
SR	Saved Resource.
STOOS	Shortest Timing Operation Order Set.

T

T	Throughput.
TB	Transform Block.
TMuC	Test Model Under Consideration.
TO	Timing Optimization.
TRANSRAM	Transform RAM.
TSMC	Taiwan Semiconductor Manufacturing Company Limited.

U

UHDV	Ultra High Definition Video.
UMC	United Microelectronics Corporation.
URQ	Uniform-Reconstruction Quantization.

V

VCEG	Video Coding Experts Group.
VLC	Variable Length Coding.
VLSI	Very Large Scale Integration.

W

WHXGA	Widescreen Hexadecatuple eXtended Graphics Array (5120×3200).
WVGA	Wide Video Graphics Array (800×480).

CHAPTER 1

Introduction

1.1 Video Coding

Digital video has been becoming an indispensable media to human life with a variety of consumer applications, including digital television broadcasting, Internet video streaming, mobile video streaming, video disc playing and video calling. Due to the fact that raw digital video data are huge while storage capacity and transmission bandwidth for these applications are limited, compression of video data is obligatory. Video compression, or video coding is the process to reduce the redundancy in digital video to achieve a fewer number of bits required to represent the video. In video coding, an encoder is needed to encode or compress a video sequence into a compressed form, and a decoder is needed to decode or convert this compressed form to an approximation of the source video sequence. Video coding scenarios with encoder and decoder for the above video applications are described in Figure 1.1 and Figure 1.2 (Richardson, 2010).

The redundancy in digital video consists of three types: temporal, spatial and statistical redundancies. The temporal redundancy exists due to the high correlations or similarities between video frames that were captured at around the same

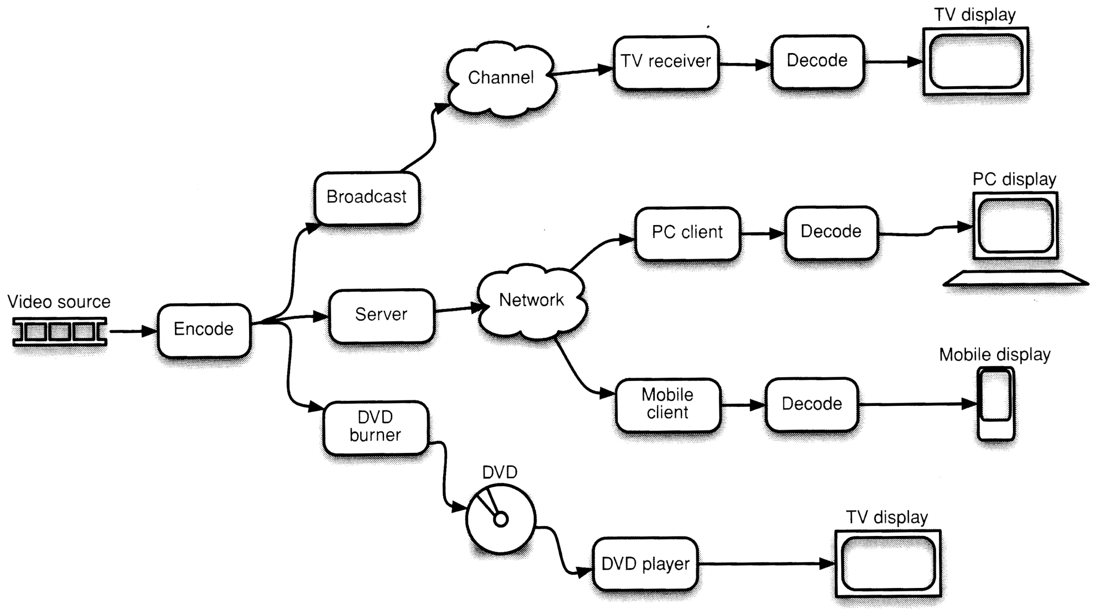


FIGURE 1.1: Video coding scenarios with digital television broadcasting, Internet video streaming, mobile video streaming and video disc playing (Richardson, 2010).

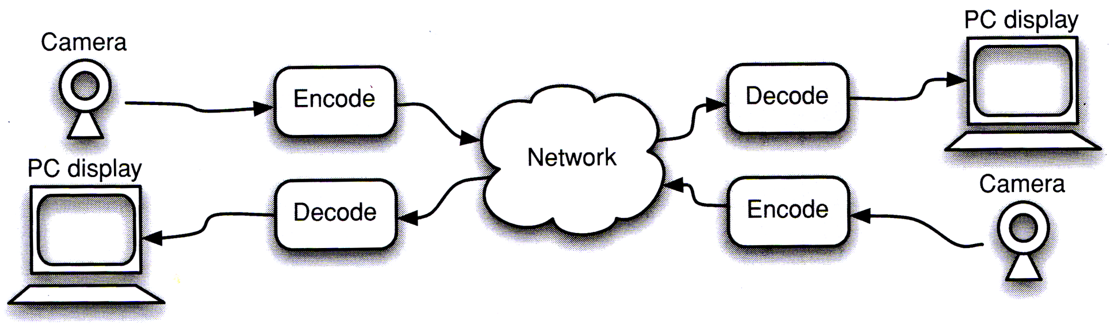


FIGURE 1.2: Video coding scenarios with video calling (Richardson, 2010).

time. The spatial redundancy exists due to the high correlations between pixels (samples) that are close to each other. In video data in which the temporal and spatial redundancy are exploited, the statistical redundancy can be reduced by representing data in a more concise format without losing information. That is the reason why a video encoder normally contains three main function units: a prediction model targeting the temporal and spatial redundancies, a spatial model

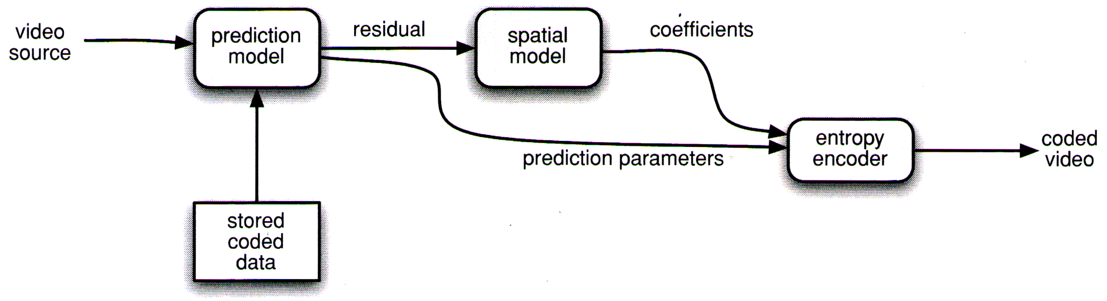


FIGURE 1.3: Video encoder block diagram (Richardson, 2010).

targeting the spatial redundancy and an entropy encoder targeting the statistical redundancy (Figure 1.3).

Many different tools for video coding have been researched and proposed. In the prediction model, inter-frame prediction tools such as motion estimation/motion compensation can be applied to reduce the temporal redundancy, and intra-frame prediction tools can be used for the spatial redundancy reduction. In the spatial model, transform coding tools, such as the Discrete Cosine Transform (DCT), Discrete Wavelet Transform (DWT) and integer transform (IT) together with vector quantization, can be employed targeting the spatial redundancy. In the entropy encoder, Run Length Coding (RLC), Variable Length Coding (VLC), Context-based Adaptive VLC (CAVLC), and Context-based Adaptive Binary Arithmetic Coding (CABAC) can be performed to reduce the statistical redundancy.

Although a variety of video coding tools are available, commercial video coding applications, industrial products and services tend to use tools recommended by video coding standards in order to simplify the inter-operability between encoders and decoders from different manufactures. The video coding standards - chronologically, H.261 (ITU-T, 1993), MPEG-1 (ISO/IEC 11172, 1993), MPEG-2/H.262

(ISO/IEC 13818-2, 1994), H.263 (ITU-T, 1998), MPEG-4 Part 2 (ISO/IEC 14496-2, 1999), and H.264/Advanced Video Coding (H.264/AVC) (Wiegand et al., 2003b,c), hence, have played pivotal roles in spreading technologies and developing video industries to a stage like today, where video applications, products and services are widely used by people in their everyday lives.

Among those video coding standards, the latest one, H.264/AVC, has significantly outperformed and displaced the preceding standards. It provides much better video quality at substantially lower bit-rate by adopting a number of new coding tools (Figure 1.4). H.264/AVC has a very broad range of applications, from low bit-rate Internet streaming applications to HDTV broadcast and Digital Cinema applications. According to Sullivan and Wiegand (1998); Wiegand et al. (2003a); Wiegand and Girod (2001), for video streaming applications, the H.264/AVC main profile allows an average bit-rate saving of about 63% compared to MPEG-2 and about 37% compared to MPEG-4 Part 2. For entertainment quality applications, the bit-rate saving of H.264/AVC compared to MPEG-2 is about 45% on average (Wiegand et al., 2003a). It is able to achieve the same Digital Satellite TV quality as the current MPEG-2 implementations at less than half of the bit-rate. In particular, the current MPEG-2 implementations work at around 3.5 Mbit/s, while H.264 works at only 1.5 Mbit/s (Wenger et al., 2005). H.264 has become widely deployed in numerous applications, products and services, such as internet video streaming like YouTube, Vimeo, iTunes Store; web software like Adobe Flash Player and Microsoft Silverlight; high definition television (HDTV) broadcasting; Blu-ray discs and players; Sony and Panasonic video recorders with AVCHD format; Canon DSLR cameras; and CCTV products, etc.

However, an increasing diversity of services and a large growth in demand for

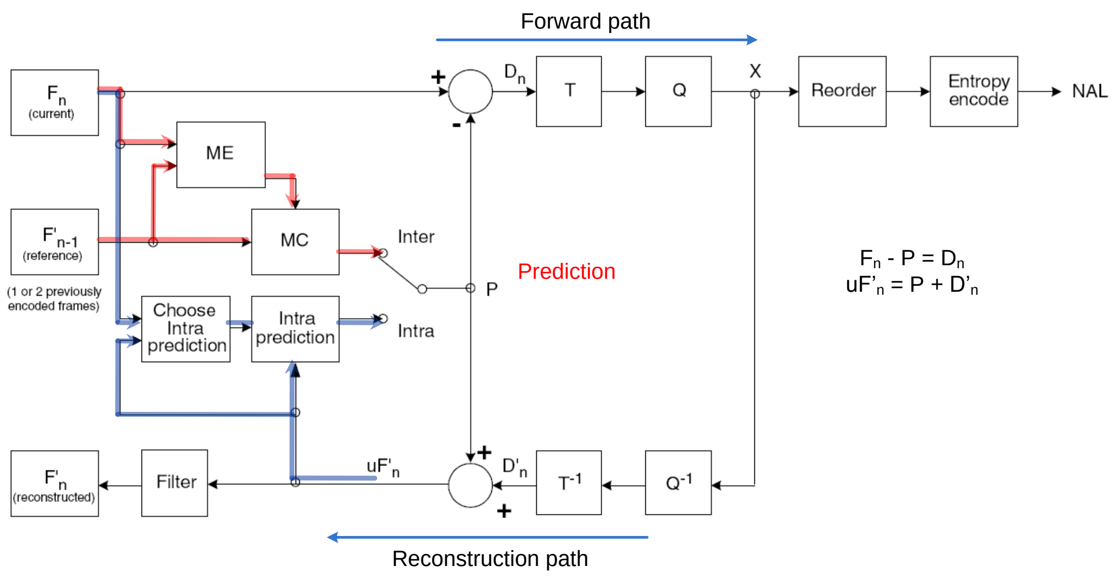


FIGURE 1.4: H.264 encoder and its coding tools (Richardson, 2003).

increasing high resolution and high quality videos (e.g. 4k × 2k or 8k × 4k resolution videos) are creating a very strong need for coding efficiency which is superior to H.264/AVC's capabilities. As a result, motivated by the impressive coding efficiency and phenomenal success of H.264/AVC in industry, the H.264 developers, ISO/IEC Moving Picture Experts Group and ITU-T Visual Coding Experts Group, have been working together again to develop a novel High Efficiency Video Coding (HEVC) standard (Bross et al., 2012). HEVC aims to deliver a dramatic improvement of quality with a 50% bit-rate reduction compared to H.264/AVC. HEVC will be finalized in early 2013.

1.2 Integer Transform

Among the redundancy reduction tools for video coding, transform coding is one of the core tools to reduce the spatial redundancy. It converts input data, i.e., image or motion-compensated residual data, into another domain, i.e., frequency domain (transform domain), to separate the input data into independent components and concentrate most of the information on a few components. The transform coding tools can be classified into two categories: the block-based and image-based transforms. The block-based transforms process video frames in units of blocks, while the image-based transforms operate on entire frames or large sections of frames. Compared to the image-based transforms, the block-based transforms require less memory and are more suitable for video coding since motion estimation and compensation are also based on block units.

Among the existing block-based transform coding tools, the Discrete Cosine Transform (DCT), and in particular the DCT-II, is the most popular tool because it has a strong "energy compaction" property (Ahmed et al. (1974); Rao and Yip (1990)). Most of the signal information tends to be concentrated in a few low-frequency components of the DCT (Figure 1.5). The DCT is used in MPEG-2, MPEG-4 with the transform size of 8×8 pixels.

The transform coding tool of H.264 and HEVC is integer transform. Due to the mismatch problem between encoder and decoder transforms and the expensive floating-point arithmetic computation of the DCT, H.264 and HEVC approximate the DCT to the integer transform. The integer transform carries out all the transforms and quantizations using only integer numbers with a high accuracy. In the H.264 high profiles, the residual data are transformed in blocks sized up to 8×8. In HEVC, a wide range of block sizes, from 4×4 to 32×32, is used to adapt the transform to the varying space-frequency characteristics of residual data.

The H.264/AVC high profiles (Marpe et al., 2005) target higher-fidelity video materials, especially for application areas like professional film production, video post

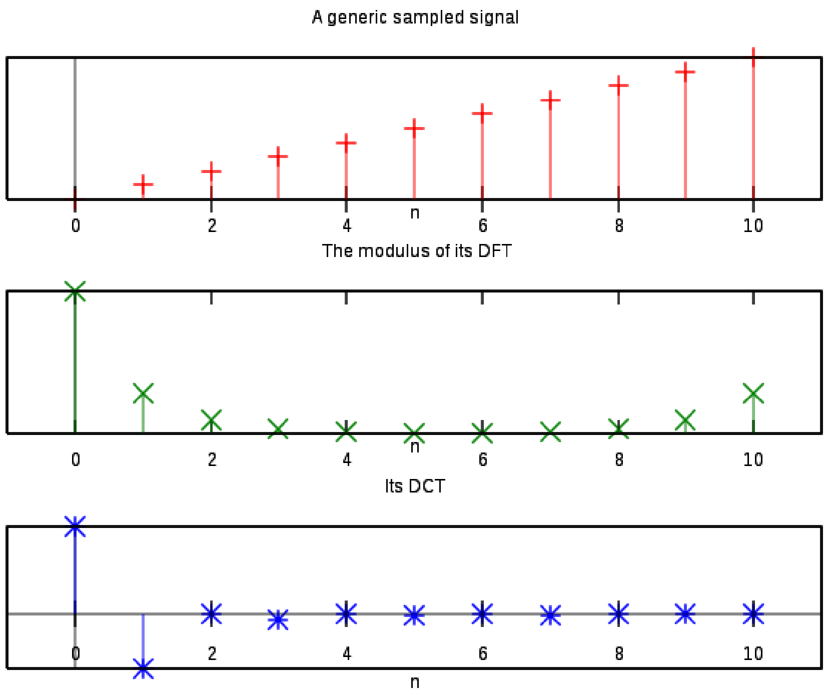


FIGURE 1.5: DCT-II (bottom) compared to the DFT (middle) of an input signal (top).

production, or high-definition TV/DVD, while the new HEVC enables the support for beyond-full high definition videos. In order to support high-definition videos, all the modules of these two standards including the integer transform modules must satisfy the high throughput requirements, where throughput is computed by the number of pixels processed in an unit of time. For hardware designs, in addition to achieving high throughput requirement, area efficiency must be taken into account to reduce hardware cost.

1.3 Challenges of Integer Transform Designs for H.264 and HEVC

High throughput and area efficiency which enable sufficient support for high resolution and beyond-high resolution videos are not only the requirements but also the challenges in H.264 high profile and HEVC designs in general. This is because the improvement of coding efficiency in H.264 and HEVC comes at a price of increased computational complexity. A lot of experiments by Saponara et al. (2003) show that the complexity of the H.264 main profile is more than one order of magnitude compared to that of the MPEG-4 Part 2 simple profile. At the decoder side, the complexity of H.264 is still two times more complex compared to MPEG-4. For the emerging standard, HEVC, the price for the bit-rate reduction by a half with the same quality is expectedly three times more complex than H.264/AVC.

For H.264 high profile and HEVC transform designs in particular, although the integer transforms are actually less complex than the DCT, the requirement of high throughput still requires significant research efforts in the literature. While in the H.264 high profiles, the transforms are up to 8×8 large with the maximum matrix coefficient of 12, in HEVC, they are up to 32×32 large with the maximum matrix coefficient of 90. These increments lead to a large increment in complexity. This complexity increment and the even-higher requirement of high throughput for beyond-full high definition videos, definitely require a lot of research efforts in future.

For hardware designs in general and integer transform hardware designs in particular, high throughput and area efficiency can be achieved at two levels: the *algorithm* level and the *architecture* level.

1.3.1 Reported Techniques Addressing H.264/AVC Transform Design Challenges

In order to achieve high-throughput and area-efficient transform designs for the H.264/AVC standard, at the *algorithm* level, the standard developers have provided fast and low-cost algorithms for all the forward and inverse transforms, including the 4×4, 8×8 integer transforms and the 2×2 and 4×4 Hadamard transforms. Similar to the DCT, these fast algorithms decompose a 2-D transform into two 1-D transforms. In the 1-D transforms, the algorithms transform data row by row or column by column. Most of the reported integer transform designs follow these fast algorithms.

At the *architecture* level, many techniques are used in the literature to increase design throughputs. A large number of reported designs (Amer et al., 2005; Kordasiewicz and Shirani, 2005; Raja et al., 2005) process in parallel all the rows/-columns of a 4×4 or 8×8 data block by using eight or sixteen transform units, respectively. A transform unit is defined as the hardware component corresponding to a 1-D transform for a row or a column. For the 8×8 transforms, these architectures lead to large bus widths of at least 1280 bits. If the architectures are implemented as hard IP cores, it might be impossible to realize in hardware implementation due to a huge number of pins, and consequently, a huge number of I/O pads required. If the architectures are implemented as soft IP cores in a system, these huge data bus widths lead to very large circuit areas due to their huge numbers of wires in the high level metal layers. Wiring areas are normally not reported in the published papers, but the fact is that they are much larger than the cell areas and significantly affect the total circuit areas of the designs. In addition, in order to transform all the rows/columns of a block simultaneously, the architectures themselves require many hardware resources. Therefore, the high throughput challenge for transform architecture designs should be addressed under the condition of having a reasonable bus width or I/O pin count.

When the bus of an architecture is narrower than a transform size, or in other words, several cycles are needed to transfer an entire transform data block to the architecture, the throughput of the architecture must be computed as the transform size divided by total transform delay, which includes input delay, processing delay and output delay. Under the condition of having a reasonable bus width, the input or output delay might be even larger than the processing delay. This significantly affects the throughput and performance of integer transform designs in particular and video codec system design in general. Therefore, the I/O delay should be taken into account when designing integer transform architectures. However, in quite a number of the reported H.264 integer transform designs (Wang et al., 2003; Kordasiewicz and Shirani, 2005; Chao et al., 2007; Choi et al., 2008; Park and Ogunfunmi, 2009; Hu et al., 2009), data are presumed available for processing at the transform modules and performance are measured without the consideration of the delay due to the data transfer.

In the literature, another technique to increase the design throughput is *pipelining*. Among the H.264/AVC main profile transform designs (4×4 in size), some designs (Chen et al., 2006; Shi et al., 2007) applied a pipelining mechanism for input, output and transform processes. In general, pipelining mechanisms definitely reduce the total delays as the modules in the architectures work in parallel. Consequently, they increase the throughput and performance of the systems. Among H.264/AVC high profile transform designs (8×8 in size), some designs (Pastuszak, 2008) applied a pipelining mechanism for the transform and quantization processes. Although the transform and quantization processers are pipelined, these 8×8 transform designs still do not count the I/O delay. In addition, the reported designs supporting these pipelining mechanisms have very large bus widths of thousands of bits, leading to extremely high wiring areas and difficulties in realizing hard IP cores.

Therefore, there is a strong need of having integer transform designs with a reasonable bus width achieving high throughputs, where the throughputs are computed including I/O delays.

In order to have an area-efficient design, the hardware-sharing techniques are used. There are many types of hardware-sharing techniques reported in the literature. In some designs (Kordasiewicz and Shirani, 2005), instead of using the official fast algorithms to input, process and output all data of a row/columns in parallel, the authors implemented the integer transforms, i.e., matrix multiplications, by directly multiplying a row/column by a column/row of the transform matrices to output one element of a row/column at a time. The multiplications are implemented using addition and shifts operations. This type of designs can share the hardware among all the row/column multiplications. However, they have small throughputs of one pixel per cycle or even less. Since for the H.264 high profiles, high throughput is the first priority requirement in order to support high definition videos, this technique is not suitable for high profile designs, but only for main profile designs.

Besides the row/column multiplication hardware-sharing technique, hardware-sharing can be deployed among the different sizes of the transforms (Chao et al., 2007; Pastuszak, 2008; Su and Fan, 2008) or even among the forward and inverse transforms (Shi et al., 2007; Choi et al., 2008). However, the sharing among forward and inverse transform hardwares might not applicable in practice. It is because in H.264 decoders, only the inverse transforms are needed. In encoders, all the forward and inverse transforms are needed and they might work at the same time. When they share the hardware, we still need two separate hardware units. In addition, a shared hardware normally has a higher cost than each individual optimal hardware. If two hardware units are used, the total shared hardware cost is larger than the total hardware cost of the two individual optimal hardwares.

The hardware-sharing among the different sizes of transforms can be among the 1-D inverse 8×8 transform and the 2-D inverse 4×4 and 2-D inverse 2×2 transforms (Chao et al., 2007), or can be among all the inverse 1-D transforms where the shared hardware can perform any 1-D inverse transform (Su and Fan, 2008), or can be among all the forward/inverse 1-D transforms where the shared forward/inverse hardware can perform one 1-D forward/inverse 8×8 transform, two/one 1-D forward/inverse 4×4 transform(s) (Su and Fan, 2008).

Therefore, there is still lack of shared architectures which share hardware among all the forward transforms and among all the inverse transforms including the 4×4 and 2×2 Hadardmard transforms.

Since the integer transform modules are parts of H.264 video encoders and decoders, they need to be integrated into larger systems. However, the transform designs in the literature have different signals and data transfer mechanisms. This makes it difficult to integrate or reuse the reported components. Therefore, a question also can be raised is that how to design portable and flexible architectures which can be easily integrated into other systems and reused in future?

1.3.2 Reported Techniques Addressing HEVC Transform Design Challenges

HEVC targets the beyond-full high resolution videos. Hence, its throughput requirement is much higher than in H.264. In addition, HEVC decoder complexity is about 1.5 times of that of H.264/AVC, while HEVC encoders are expected to be several times more complex than H.264/AVC encoders. Especially, HEVC supports 32×32 transforms with matrix coefficient values of up to 90, while H.264 supports up to 8×8 transforms with matrix coefficient values of up to 12. These large increments in complexity and the even-higher requirement of high throughputs for the beyond-full high definition videos make it more challenging to develop high-throughput and area-efficient designs for HEVC.

To the best of our knowledge, until now, there are not many HEVC transform algorithms and architectures proposed in the literature. In order to achieve high-throughput and area-efficient transform designs for the HEVC standard, at the *algorithm* level, HEVC developers use Partial Butterfly algorithms to implement the transforms using butterfly additions and multiplications in HEVC test model HM (ITU-T and JTC1, 2012). Although this series of algorithms is much less complex than the implementation by multiplications, it is still much more complex than the H.264 fast algorithms when we compare among the equivalent size transforms.

Techniques used in the literature to develop fast and low-complexity transform algorithms for HEVC are manipulating and decomposing the up-to-8×8 transform matrices of HEVC based on the H.264 transform matrices, so that the transforms can be implemented using addition and shift operations. However, these techniques are impossible to extend to larger transform sizes because H.264 only supports the transforms of up to 8×8 in size.

At the *architecture* level, techniques to improve design throughput have not reported yet in the literature. To reduce the area, hardware-sharing techniques are still applied in HEVC designs. Similar to the H.264 designs, some researches implemented the integer transforms by directly multiplying a row/column by a column/row of the transform matrices to output one element of a row/column at a time. However, these designs have small throughputs of one pixel per cycle or even less. On the other hand, some designs also share resources among the large and small sizes of the transforms.

1.4 Dissertation Contributions and Organization

We address in this dissertation the challenges of the transform designs for the H.264/AVC high profiles and the emerging HEVC, and presents high-throughput and area-efficient integer transform designs under the condition of having reasonable I/O bus widths or I/O pin counts. In addition, the dissertation also introduces techniques for portable integer transform designs of H.264/AVC in particular, and for portable hardware designs in general. The design techniques proposed in the dissertation can be applied not only to integer transform designs of the H.264/AVC high profiles and HEVC but also to other hardware designs. The dissertation contributions are chronologically listed as follows.

1. We address the question of how to develop portable integer transform designs for the H.264/AVC high profiles in particular and portable hardware designs in general. In order to have portable hardware designs which can be easily integrated

to larger system or reused in future, system bus standards should be employed. This is because the system bus standards specify the control and data signal names and their functions, as well as required handshaking mechanisms. Some system bus standard can be found such as the Wishbone bus and the Advanced Microcontroller Bus Architecture (AMBA) by ARM.

In this dissertation, we propose two portable inverse transform architectures supporting all types of the inverse transforms and rescaling function for H.264/AVC with preferable high throughputs under reasonable bus-width constraints. Each design employs a Wishbone system bus and is supported by a SoC system with an ASIP and DMAC(s). In the first design, a shared-hardware unit among all the inverse transforms including 4×4 and 2×2 inverse Hadamard transforms is proposed to target area-efficient challenge. This shared transform unit has filled the gap of inverse transform hardware-sharing in the literature. The shared unit includes only one 1-D 8×8 inverse transform unit with three levels of hardware sharing. The first level is that two 4×4 inverse transform units are embedded into the 1-D 8×8 inverse transform unit. The second level is that one 4×4 inverse Hadamard transform shares the same hardware with one 4×4 inverse transform unit. The third level is that the 2-D 2×2 inverse Hadamard transform is embedded into the 1-D 4×4 inverse transform unit. For portability, the Wishbone shared bus system is employed in this architecture.

While the first design targets portability and area efficiency, the second design targets high throughputs together with both area efficiency and portability. This design continues to employ the proposed shared transform unit from the first design for area efficiency. With the constraint of a reasonable I/O bus width (eight pixels for each input or output bus), in order to achieve a high throughput, the second design employs two transform units and deploys a pipelining mechanism among all the input, output and transform processes. For the portability and pipeline support, the Wishbone crossbar-switch bus system is used. The simulation results show that the second design can deliver a throughput of eight ppc and a normalized throughput of sixty-four ppc and 15.6 Gpps at 144 MHz using 0.18 μm technology. This is higher than other reported designs. It can support the transforms for video with the resolution of up to 38.4 Mpels (larger than the resolution of 7680 $\times$ 4800 pels with progressive scan frequency of 30 Hz. The resulting circuit area is considerably minimal compared to the designs in its class thanks to the proposed shared hardware with the embodiment of 4 $\times$ 4 transform circuit in the 8 $\times$ 8 transform circuit.

2. We address the high-throughput and area-efficient challenges on transform designs under a reasonable bus width constraint by introducing a high-throughput and area-efficient forward and inverse transform architecture for H.264/AVC. The proposed architecture supports all types of the forward and inverse transforms together with quantization and rescaling functions. For area efficiency, the design continues to employ the proposed inverse transform unit in the first design and proposed a novel shared forward transform unit with three hardware sharing levels, which is similar to the inverse unit. Based on an observation that the transform units only read input data in half of the processing time, a novel series of input and output buffers is proposed to balance the data transferring load in I/O bus. This is shown to reduce the bus width by half for the same hardware architecture. Thanks to the novel I/O buffer designs, two transform units are integrated into the computing kernel to double the processing speed and utilize the bus width. By deploying a fully pipelining mechanism for input, output, transform and quantization processes, the architecture can out-perform other reported designs in terms of throughput and area efficiency.

3. The next contribution of this research work is a novel fair metric for performance-cost comparison among integer transform designs. The metric development is based on the fact that some designs use a very large bus width but their authors were still able to claim their area efficiency. This is because the wire area or interconnect area is not counted in the current comparison metrics. Therefore, the dissertation provides performance-cost analyses and proposes a novel performance-cost metric for H.264 integer transform designs in order to have a comprehensive comparison among all the reported designs. The proposed metric is defined as the ratio of data throughput over the design cost, which includes power, area, and delay, and issues associated with interconnections in sub-micron design.

4. We propose a novel optimization method to achieve fast and low-cost algorithm implementations for scalar-multiplication-containing algorithms. The optimization method can be applied to algorithms which (a) contain scalar multiplications, (b) enable operations to be performed in parallel, and (c) require a short running time with a low resource consumption. The method (Figure 6.3) includes a multiplication-to-addition conversion step and three levels of optimization: (1) Complexity Optimization, (2) Timing Optimization, and (3) Resource Optimization. The initial step of the method is to convert all multiplications in the original scalar multiplications to addition and shift operations. Next, in the first level of optimization, a complexity optimization algorithm is proposed to minimize the number of addition/subtractions used in the result of the previous multiplication-to-addition conversion step. In the second optimization level, a timing optimization strategy is proposed to structurally arrange the addition/subtractions in the minimized conversion result to achieve the shortest running time. While the first two optimization levels process each multiplication separately, the third level works with all the multiplications. In this level, a resource optimization

algorithm is proposed to find the best operation order of the addition/subtractions for each multiplication, which can provide the best resource-sharing among all the multiplications. Experimental results show that both the running time and the cost of the generated implementations for scalar multiplication algorithms by the proposed method are about 90% less than those of the original implementation by multiplications of the algorithms (Table 6.14 and Table 6.15). The method is computationally feasible to be applied to all the Partial Butterfly algorithms from 4×4 to 32×32 in size.

5. Based on the strategy of designing the small transforms as parts of the large transforms facilitating resource-sharing, and based on the optimized implementations obtained by applying the proposed method to the Partial Butterfly algorithms, we have proposed a series of novel hardware-oriented 4×4 and 8×8 fast and low-cost forward transform algorithms for HEVC (Figure 6.11). The number of addition/subtractions of the proposed implementations for the 4×4 and 8×8 transforms is fourteen and fifty-eight; while their longest paths include four and five addition/subtractions, respectively. The running time and the cost of the proposed fast and low-cost transform algorithms (Table 6.14 and Table 6.15) are 75% and 87% less than that of the original Partial Butterfly algorithms in HMs, respectively. Compared to the reported algorithms for HEVC in the literature, the proposed 8×8 algorithm is faster and its cost is lower (Table 6.17). Fast and low-cost transform algorithms for larger sizes can be also developed using the same method.

6. A high-throughput and area-efficient architecture for the HEVC 4×4 and 8×8 integer transforms is developed, implemented and fabricated based on the proposed algorithms. A number of techniques has also been proposed, including (1) multi-cycle adder design; (2) multi-stage register design; (3) pipelining mechanism

among the addition/subtractions; (4) resource binding; (5) micro-code and control signal optimization; and (6) fully pipelining mechanism among input, output and transform processes. Despite under a very tight constraint on I/O pin count or I/O bus width of half pixel, thanks to the techniques, the proposed architecture can support the transforms for up to Quad-Full High Definition (QFHD) videos at the progressive scan frequency of 30 Hz. This is eight times as large as that of Martuza and Wahid (2012)'s design (Table 6.33). The power consumed by the proposed architecture is only 44% as much as the power consumed by the Martuza and Wahid (2012)'s design.

The dissertation is organized as follows. Chapter 2 provides some background knowledge about the DCT and the integer transforms in H.264 and HEVC and the related works. Chapter 3 describes two portable inverse transform architectures for H.264/AVC with the goal of achieving high throughputs under a reasonable bus width constraint. Chapter 4 introduces a high-throughput and area-efficient forward and inverse tranform architecture for H.264. Chapter 5 is devoted to the performance-cost analyses for H.264 integer transform designs. Chapter 6 introduces an optimization method to achieve fast and low cost algorithm implementation for scalar-multiplication-containing algorithms, and proposes a series of fast and low-cost algorithms for the HEVC integer transforms, followed by a high-throughput and area-efficient architecture for the HEVC integer transforms. Lastly, the conclusions of the dissertation and future work are given in Chapter 7.

Chapter 2

Background and Related Works

While the Discrete Cosine Transform (DCT) is used in the MPEG-2 and MPEG-4 video coding standards, its integer approximation, the integer transform, is used in H.264 and HEVC to avoid the mismatch problem between encoder and decoder transforms and the expensive floating-point arithmetic computation of the DCT. In MPEG-2 and MPEG-4, the DCT size is 8×8. In H.264, the transforms are from 2×2 to 8×8 in size. In HEVC, the transforms are even larger, from 4×4 to 32×32 in size. One reason for these increments of the transform sizes is to adapt the transform to the varying spatial frequency characteristics of residual data. Another reason is that larger transform sizes typically enable better compression than smaller sizes.

The H.264 integer transform is a loose approximation of the DCT for simple transform implementations. The magnitudes of transform matrix coefficients are less than 2 and 12 for 4×4 and 8×8 transform sizes, respectively. However, in HEVC, the sizes of the transforms are much larger. Hence, loose approximations with small matrix coefficients are impossible. Therefore, the HEVC developers use a close approximation with symmetry properties to facilitate faster implementation. The magnitudes of the HEVC transform matrix coefficients are up to 90.

In this chapter, background knowledge about the DCT and the integer transforms in H.264 and HEVC together with their quantizations are introduced in Section 2.1. Section 2.2 presents the related works of the integer transforms in the literature.

2.1 Background

2.1.1 Discrete Cosine Transform and Quantization

2.1.1.1 Discrete Cosine Transform

The Discrete Cosine Transform (DCT) has been adopted as transform coder in the JPEG, H.26x, and MPEG-1 and -2 compression standards due to its excellent energy compaction for highly correlated data, and the availabilities of fast algorithms among the orthogonal transforms.

An 1-D DCT operates on a sequence of N samples, X, to create a sequence of N coefficients, Y. Given a data sample sequence X_i with $i = 0, 1, 2, ..., N-1$, its 1-D N-point forward DCT - coefficient sequence, Y_k, is defined by Equation (2.1) with $k = 0, 1, ..., N-1$.

$$Y_k = C_k \sum_{i=0}^{N-1} X_i \cos[\frac{\pi(2i+1)k}{2N}] \tag{2.1}$$

And X_i with $i = 0, 1, 2, ..., N-1$, the inverse DCT of Y_k with $k = 0, 1, ..., N-1$, is computed as

$$X_i = \sum_{k=0}^{N-1} C_k Y_k \cos[\frac{\pi(2i+1)k}{2N}], \tag{2.2}$$

where

$$C_k = \begin{cases} \sqrt{\frac{1}{N}} & \text{when } k = 0 \\ \sqrt{\frac{2}{N}} & \text{when } 1 \le k \le N-1. \end{cases} \tag{2.3}$$

Equation (2.1) and Equation (2.2) can be rewritten as Equation (2.4) and Equation (2.5), respectively:

$$Y_k = \sum\nolimits_{i=0}^{N-1} X_i A_{k,i} \qquad (2.4) \qquad\qquad X_i = \sum\nolimits_{k=0}^{N-1} Y_k A_{k,i}, \qquad (2.5)$$

where A is an $N \times N$ array with elements' values as follows:

$$A_{k,i} = C_k \cos \frac{(2i+1)k\pi}{2N}. \qquad (2.6)$$

Considering A is a matrix, the 1-D DCT and its inverse, the 1-D IDCT, also can be described in terms of matrix multiplication as in Equation (2.7), Equation (2.8) respectively, where X is a row matrix of samples, Y is a row matrix of coefficients, and A is an $N \times N$ transform matrix as described in Equation (2.9).

$$Y = XA^T \qquad (2.7) \qquad\qquad X = YA \qquad (2.8)$$

$$A_{N\times N} = \frac{1}{\sqrt{N}} \begin{bmatrix} 1 & 1 & \dots & 1 & \dots & 1 \\ \sqrt{2}\cos\frac{\pi}{2N} & \sqrt{2}\cos\frac{3\pi}{2N} & \dots & \sqrt{2}\cos\frac{(2i+1)\pi}{2N} & \dots & \sqrt{2}\cos\frac{(2N-1)\pi}{2N} \\ \sqrt{2}\cos\frac{2\pi}{2N} & \sqrt{2}\cos\frac{6\pi}{2N} & \dots & \sqrt{2}\cos\frac{2(2i+1)\pi}{2N} & \dots & \sqrt{2}\cos\frac{2(2N-1)\pi}{2N} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ \sqrt{2}\cos\frac{k\pi}{2N} & \sqrt{2}\cos\frac{3k\pi}{2N} & \dots & \sqrt{2}\cos\frac{k(2i+1)\pi}{2N} & \dots & \sqrt{2}\cos\frac{k(2N-1)\pi}{2N} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ \sqrt{2}\cos\frac{(N-1)\pi}{2N} & \sqrt{2}\cos\frac{3(N-1)\pi}{2N} & \dots & \sqrt{2}\cos\frac{(N-1)(2i+1)\pi}{2N} & \dots & \sqrt{2}\cos\frac{(N-1)(2N-1)\pi}{2N} \end{bmatrix} \qquad (2.9)$$

When $N = 2$, $N = 4$, we have:

$$A_{2\times 2} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ \sqrt{2}\cos\frac{\pi}{4} & \sqrt{2}\cos\frac{3\pi}{4} \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \qquad (2.10)$$

$$A_{4\times4} = \frac{1}{2}\begin{bmatrix} 1 & 1 & 1 & 1 \\ \sqrt{2}\cos\frac{\pi}{8} & \sqrt{2}\cos\frac{3\pi}{8} & -\sqrt{2}\cos\frac{3\pi}{8} & -\sqrt{2}\cos\frac{\pi}{8} \\ 1 & -1 & -1 & 1 \\ \sqrt{2}\cos\frac{3\pi}{8} & -\sqrt{2}\cos\frac{\pi}{8} & \sqrt{2}\cos\frac{\pi}{8} & -\sqrt{2}\cos\frac{3\pi}{8} \end{bmatrix} \tag{2.11}$$

Or

$$A_{4\times4} = \frac{1}{2}\begin{bmatrix} 1 & 1 & 1 & 1 \\ b & c & -c & b \\ 1 & -1 & -1 & 1 \\ c & -b & b & -c \end{bmatrix} \tag{2.12}$$

where

$$\begin{aligned} b &= \sqrt{2}\cos\tfrac{\pi}{8} \approx 1.307 \\ c &= \sqrt{2}\cos\tfrac{3\pi}{8} \approx 0.541. \end{aligned} \tag{2.13}$$

For image and video compression, the 2-D DCT is required. A 2-D DCT operates on a block of $N \times N$ samples, X, to create a block of $N \times N$ coefficients, Y. Given a data block $X_{i,j}$ with $i, j = 0, 1, 2, ..., N-1$, its 2-D N-point forward DCT - coefficient block, $Y_{k,l}$, is defined by Equation (2.14) with $k, l = 0, 1, ..., N-1$.

$$Y_{k,l} = C_k C_l \sum_{i=0}^{N-1}\sum_{j=0}^{N-1} X_{i,j} \cos\frac{(2i+1)k\pi}{2N} \cos\frac{(2j+1)l\pi}{2N} \tag{2.14}$$

And $X_{i,j}$ with $i, j = 0, 1, 2, ..., N-1$, the inverse DCT of $Y_{k,l}$ with $k, l = 0, 1, ..., N-1$, is computed as in Equation (2.15).

$$X_{i,j} = \sum_{k=0}^{N-1}\sum_{l=0}^{N-1} X_{k,l} C_k C_l \cos\frac{(2i+1)k\pi}{2N} \cos\frac{(2j+1)l\pi}{2N} \tag{2.15}$$

where C, i.e. C_k and C_l, follows Equation (2.3).

Using matrix A described in Equation (2.6) or Equation (2.9) to rewrite Equation (2.14), we have:

$$\begin{aligned} Y_{k,l} &= \sum_{i=0}^{N-1} \left(\sum_{j=0}^{N-1} X_{i,j} C_l \cos \frac{(2j+1)l\pi}{2N} \right) C_k \cos \frac{(2i+1)k\pi}{2N} \\ &= \sum_{i=0}^{N-1} \left(\sum_{j=0}^{N-1} X_{i,j} A_{l,j} \right) A_{k,i} \end{aligned} \tag{2.16}$$

Naming $Y'_{i,l}$ as an $N \times N$ matrix, where its row i is the 1-D DCT of row i in X. For row i of Y' and X, based on Equation (2.4), we have:

$$Y'_{i,l} = \sum_{j=0}^{N-1} X_{i,j} A_{l,j} \tag{2.17}$$

Therefore, Equation (2.16) becomes:

$$Y_{k,l} = \sum_{i=0}^{N-1} Y'_{i,l} A_{k,i} \tag{2.18}$$

Based on Equation (2.4), as can be seen from Equation (2.18), column l of Y is actually the 1-D DCT of row l in Y'^T, or in another word, is the 1-D DCT of column l in Y'. Hence, we can write in terms of matrix multiplication:

$$Y' = XA^T \tag{2.19}$$

$$Y^T = Y'^T A^T \tag{2.20}$$

Thus, we have another formula of the 2-D DCT Y and the 2-D IDCT X, which is often seen as a definition of the 2-D DCT.

$$Y = \left(Y'^T A^T\right)^T = AY' = AXA^T \tag{2.21}$$

$$X = A^T Y A \tag{2.22}$$

All these manipulations lead to a conclusion that the 2-D forward DCT can be

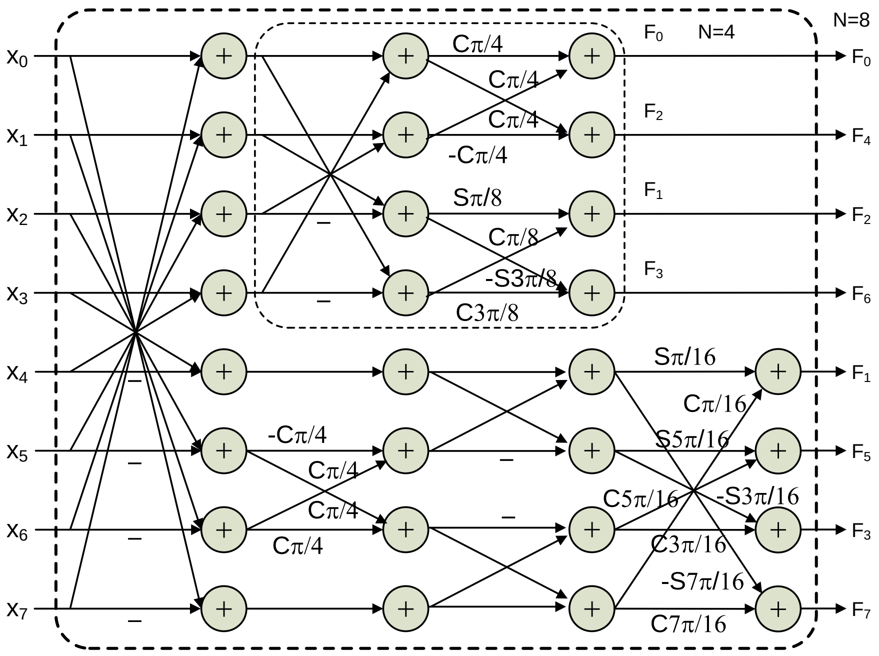


FIGURE 2.1: Forward DCT flow graph for $N = 8$, $Ci = cos(i)$, $Si = sin(i)$ (Chen et al. (1977), Rao and Yip (1990)).

.

implemented using the 1-D forward DCT, by taking a 1-D DCT on the rows of the input block, followed by a 1-D DCT on the columns of the semi-transformed matrix. Similarly, the 2-D IDCT can be implemented using the 1-D IDCT, by taking a 1-D IDCT on the columns of the input block, followed by a 1-D IDCT on the rows of the semi-transformed matrix.

The first DCT algorithm was proposed by Ahmed et al. (1974) where a double-sized Fast Fourier Transform (FFT) algorithm was used with complex arithmetic throughout the computation. Chen et al. (1977) later presented a faster 1-D DCT computation by exploiting the sparseness of the matrices involved. For $N = 8$, the numbers of additions and multiplications are twenty-six and sixteen, respectively (Figure 2.1). Lee (1984) further improved the technique by reducing the number multiplications to 12.

The computational complexity of the 2-D DCT is equal to $2N$ times the numbers of additions and multiplications required for a 1-D DCT. For $N = 8$, the number of additions and multiplications are 416 and 256, respectively, using Chens algorithm (Chen et al., 1977); and 464 and 192, respectively, using Lee's (Lee, 1984). Later, Cho and Lee (1991) proposed a 2-D DCT algorithm which required only 466 additions and 96 multiplications.

2.1.1.2 Quantization

In image and video coding, after transform coding, quantization is usually performed to map the coefficient values with a bigger range to quantized values with a smaller range, so that the quantized values can be represented with fewer bits than the original coefficient values. There are two types of quantization: scalar and vector quantization. A scalar quantizer maps one sample of the input to one quantized output value, while a vector quantizer maps a group of input samples to a group of quantized values. Scalar quantization is commonly used in video coding standards due to its simplicity.

A simple example of scalar quantization is:

$$FQ = round\left(\frac{X}{QP}\right) \tag{2.23}$$

$$Y = FQ.QP \tag{2.24}$$

where X is input sample, QP (Quantization Parameter) is quantization step size, FQ is Forward Quantization output, and Y is quantized output.

Figure 2.2 shows two examples of scalar quantization, a linear and non-linear quantization examples.

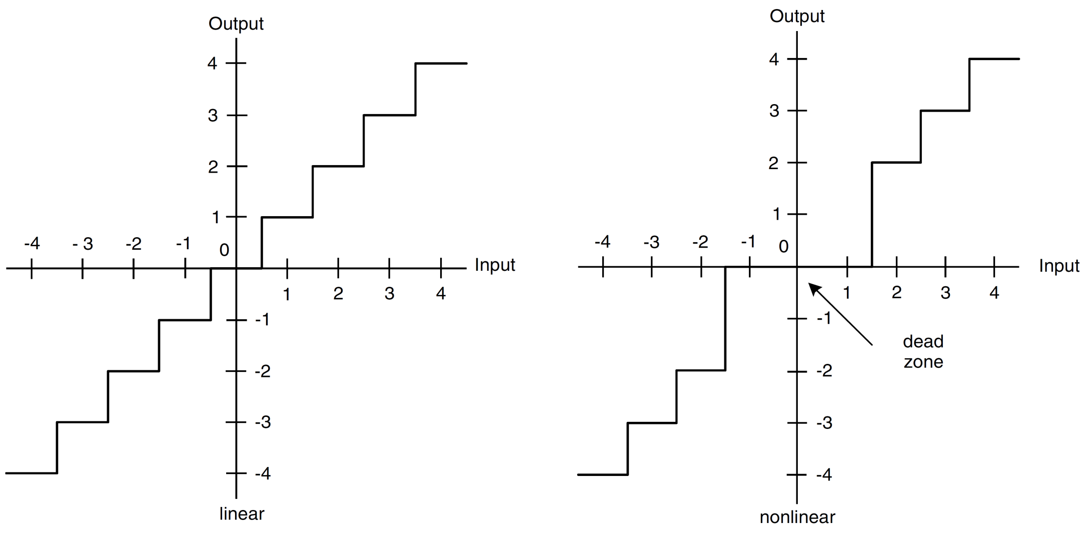


FIGURE 2.2: A linear and non-linear quantization (Richardson, 2003).

In video coding, quantization is usually separated into two parts: forward quantization, whose operation showed in Equation (2.23), in encoders; and inverse quantization or rescaling, whose operation showed in Equation (2.24), in decoders. The step size, QP, is a critical parameter. If it is large, the range of quantized values is small and thus the compression efficiency is high. However, the rescaled values are a crude approximation to the original values. If the step size is small, the rescaled values closely match the original values, but the compression efficiency is low.

2.1.2 Integer Transforms and Quantization in H.264/AVC

2.1.2.1 Forward/Inverse Integer Transforms

The 8 × 8 Discrete Cosine Transform (DCT) is used as the basic transform in MPEG-1, MPEG-2 (ISO/IEC 13818-2, 1994), MPEG-4 (ISO/IEC 14496-2, 1999) and H.263. To avoid the mismatch problem between encoder and decoder transforms and reduce the expensive floating-point arithmetic implementation of the

DCT, H.264 approximates the DCT in transform coding to carry out all transformations using just integer numbers. That is the reason why the H.264 transform is called Integer transform.

In H.264 encoder (Figure 2.3), the output of prediction processes is residual data, which is divided into marcroblocks. A 16×16 residual macroblock, in the usual case of "4:2:0" color sampling, consists of a 16×16 luma and two 8×8 chroma blocks, which are transformed differently (Richardson, 2003). The chroma components, including eight blocks labeled 18-25 in Figure 2.5(b) and Figure 2.5(c), are all 4×4 transformed. The outputs of a forward transform is named transform coefficients. The chroma DC coefficients are grouped (blocks 16, 17) and further 2×2 Hadamard transformed. On the other hand, the luma components are transformed according to prediction modes and sub-partition sizes as showed in Table 2.1. Figure 2.5(a) illustrates the luma transform in 16×16 intra prediction mode. After that, quantization is performed, and quantized transform coefficients are then transmitted in the scanning order shown in Figure 2.5 to the entropy encoder. At decoder side (Figure 2.4), data from entropy decoder, after being reordered, are inverse quantized and inverse transformed. Subsequently, the data is sent to an addition block to compensate the deducted values in the encoders' prediction processes.

Figure 2.6 shows a complete process from input residual block X, through forward integer transform (FIT), quantization, then to inverse quantization (rescaling), inverse integer transform (IIT) and output residual block X'. The core forward/inverse transforms includes 4×4 transform for chroma and luma residual data, and 8×8 transform for luma residual data. The 2×2 and 4×4 DC forward/inverse transforms are the 2×2 Hadamard transforms for DC coefficients of chroma components and the 4×4 Hadamard transforms for DC coefficients of luma components

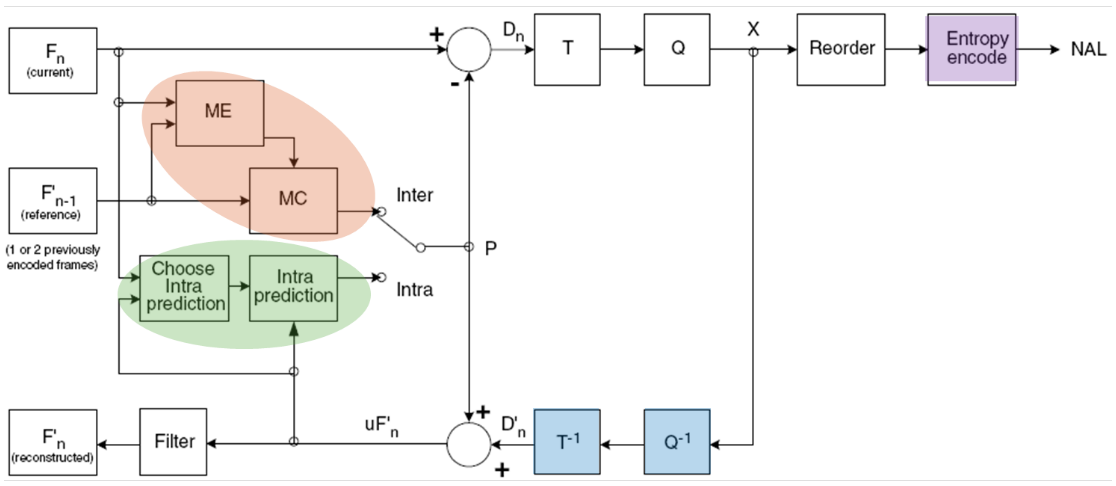


FIGURE 2.3: H.264 encoder block diagram (Richardson, 2003).

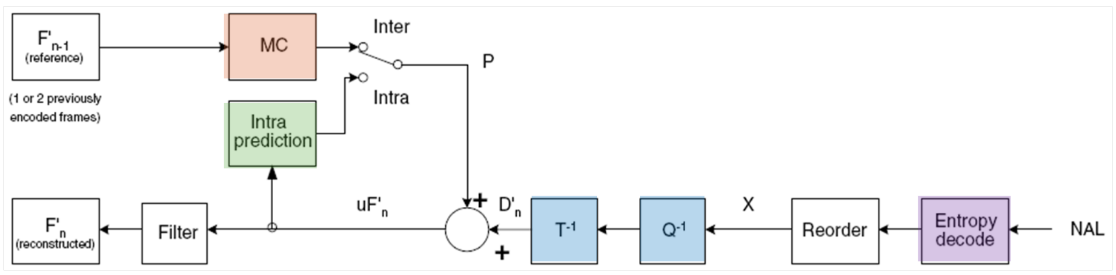


FIGURE 2.4: H.264 decoder block diagram (Richardson, 2003).

TABLE 2.1: Luma forward transforms according to prediction modes and sub-partition sizes. FIT: Forward integer transform; FHT: Forward Hadamard transform.

Sub-partition size	Inter pred. mode	4×4 / 8×8 Intra pred. mode	16×16 Intra pred. mode
$\exists$ a partition $< 8 \times 8$	4×4 FIT	4×4 FIT	4×4 FIT, followed by 4×4 FHT for DC coeff
All partitions $\geq 8 \times 8$	8×8 FIT	8×8 FIT	

in 16×16 intra prediction mode.

All FIT/IITs do the same operation as described in Equation (2.25) with different transform matrix C, except for 4×4 Forward HT, which follows Equation (2.26).

$$W = CXC^T, \tag{2.25}$$

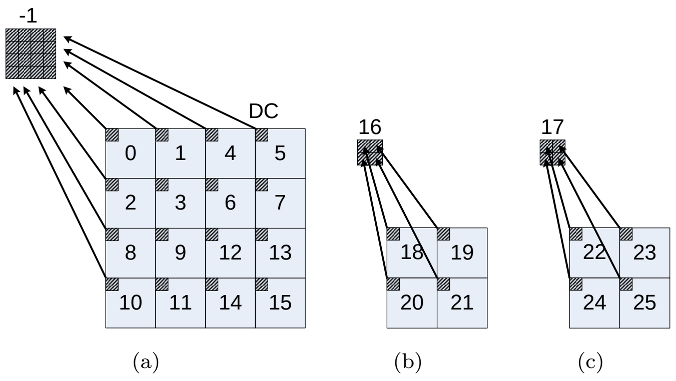


FIGURE 2.5: Scanning order of residual blocks within a 16 × 16 macroblock (Richardson, 2003). (a) Luma, (b) Cb, and (c) Cr.

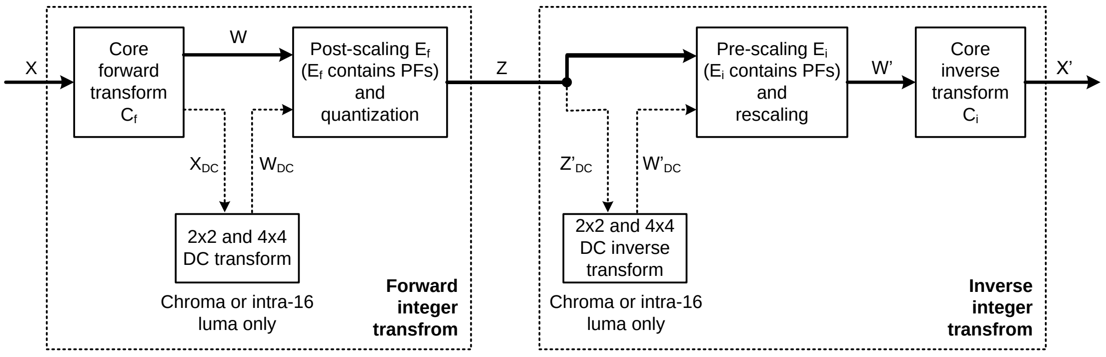

X
Core forward transform C_f
W
Post-scaling E_f (E_f contains PFs) and quantization
Z
X_DC
W_DC
2x2 and 4x4 DC transform
Chroma or intra-16 luma only
Forward integer transfrom
Pre-scaling E_i (E_i contains PFs) and rescaling
W'
Core inverse transform C_i
X'
Z'_DC
W'_DC
2x2 and 4x4 DC inverse transform
Chroma or intra-16 luma only
Inverse integer transfrom

FIGURE 2.6: H.264 transform, quantization, rescale and inverse transform flow (Richardson, 2003).

$$W = CXC^T \frac{1}{2}. \tag{2.26}$$

In particular, 4×4 FIT, 8×8 FIT, 4×4 FHT and 2×2 FHT use C_f (Equation (2.27)), C_{ef} (Equation (2.28)), $H_{f4\times 4}$ (Equation (2.29)) and $H_{f2\times 2}$ (Equation (2.30)), while 4×4 IIT, 8×8 IIT, 4×4 IHT and 2×2 IHT use C_i (Equation (2.31)) and C_{ei} (Equation (2.32)), $H_{i4\times 4}$ (Equation (2.33)) and $H_{i2\times 2}$ (Equation (2.34)), respectively.

$$C_f = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & -2 \\ 1 & -1 & -1 & 1 \\ 1 & -2 & 2 & -1 \end{bmatrix}, \tag{2.27}$$

$$C_{ef} = \frac{1}{8} \begin{bmatrix} 8 & 8 & 8 & 8 & 8 & 8 & 8 & 8 \\ 12 & 10 & 6 & 3 & -3 & -6 & -10 & -12 \\ 8 & 4 & -4 & -8 & -8 & -4 & 4 & 8 \\ 10 & -3 & -12 & -6 & 6 & 12 & 3 & -10 \\ 8 & -8 & -8 & 8 & 8 & -8 & -8 & 8 \\ 6 & -12 & 3 & 10 & -10 & -3 & 12 & -6 \\ 4 & -8 & 8 & -4 & -4 & 8 & -8 & 4 \\ 3 & -6 & 10 & -12 & 12 & -10 & 6 & -3 \end{bmatrix}, \tag{2.28}$$

$$H_{f4\times 4} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix}, \tag{2.29}$$

$$H_{f2\times 2} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \tag{2.30}$$

$$\begin{tikzcd}[column sep=huge] X \arrow[r, "\text{Transform}", "Y = C_f(X \otimes E_f)C_f^T"'] & Y \arrow[r, "\text{Quantization}", "Z_{ij} = round(Y_{ij} / Q_{step})"'] & Z \end{tikzcd}$$

FIGURE 2.7: DCT integer approximation with scaling in H.264 transform coding.

$$C_i = \begin{bmatrix} 1 & 1 & 1 & 0.5 \\ 1 & 0.5 & -1 & -1 \\ 1 & -0.5 & -1 & 1 \\ 1 & -1 & 1 & -0.5 \end{bmatrix}, \tag{2.31}$$

$$C_{ei} = C_{ef}^T, \tag{2.32}$$

$$H_{i4\times4} = H_{f4\times4}, \tag{2.33}$$

$$H_{i2\times2} = H_{f2\times2}. \tag{2.34}$$

2.1.2.2 Quantization and Rescaling

As the H.264 integer transforms are integer approximations of the DCT, in order to ensure the accuracy, scaling is used together with the integer transforms (Figure 2.7). In the figure, X, Y, and Z are the input, transformed, and quantized and transformed matrices, respectively. E_f and C_f are the scaling and forward integer transform matrices, respectively. The symbol $\bigotimes$ indicates the multiplication of the factors having the same positions in the two matrices. Quantizer step size, Q_{step}, is determined by quantization parameter QP (from 0 to 51) (Richardson, 2003). In order to reduce the number of multiplications, E_f multiplication is performed together with quantization as shown in Figure 2.8, where PF_f is position factor in E_f. Consequently, the inverse transforms and rescaling are performed as shown in Figure 2.9, where PF_i is position factor in pre-scaling matrix E_i.

$$X \xrightarrow[W = C_f X C_f^T]{\text{Integer transform}} W \xrightarrow[Z_{ij} = round(W_{ij} PF_f \,/\, Q_{step})]{\text{Quantization}} Z$$

FIGURE 2.8: Integer transform with post-scaling in H.264.

$$Z \xrightarrow[W_{ij}^{'} = Z_{ij} Q_{step} PF_i \, 64]{\text{Rescaling}} W^{'} \xrightarrow[X^{'} = C_i^T W^{'} C_i \,/\, 64]{\text{Inverse IT}} X^{'}$$

FIGURE 2.9: Inverse integer transform with rescaling in H.264.

In order to simplify the computation and avoid division, the factor $\frac{PF}{Q_{step}}$ is implemented as a multiplication by MF and a right shift.

$$\begin{aligned} Z_{ij} = round\left(W_{ij}\tfrac{MF}{2^{qbits}}\right), \quad & \text{where } \tfrac{MF}{2^{qbits}} = \tfrac{PF}{Q_{step}}, \\ & qbits = 15 + \left\lfloor \tfrac{QP}{6} \right\rfloor. \end{aligned} \tag{2.35}$$

Therefore, Z_{ij} can be computed as $|Z_{ij}| = (|W_{ij}| \times MF + f) \ll qbits$ and $sign(Z_{ij}) = sign(W_{ij})$, where $f = \frac{2^{qbits}}{3}$ for intra mode and $f = \frac{2^{qbits}}{6}$ for inter mode.

Table 2.2 shows quantization and rescaling operations corresponding to each type of transforms. It should be noted that H.264 does not specify Q_{step} or PF for the inverse transforms directly. Instead, it specifies parameter $V = Q_{step}PF_i64$ for each $0 \le QP \le 5$ and for each coefficient position. Therefore, $W_{ij}' = Z_{ij}V_{ij}2^{\lfloor \frac{QP}{6} \rfloor}$ (Table 2.2).

2.1.3 Integer Transform in HEVC

Figure 2.10 shows the block diagram of HEVC hybrid video encoder (Sullivan et al., 2012). Although it is similar to H.264, there are some key differences that improve compression efficiency of HEVC over its predecessors, including (1) Coding Tree Units and Coding Tree Block structure; (2) Transform Units and

Table 2.2: Quantization/Rescaling for all FIT/IIT transforms.

Transform	Quantization/Rescaling operations
4×4 FIT	$$\lvert Z_{ij}\rvert = \left(\lvert W_{ij}\rvert .MF + f\right) \gg qbits \quad (2.36a)$$ $$sign\left(Z_{ij}\right) = sign\left(W_{ij}\right) \quad (2.36b)$$
8×8 FIT	$$\lvert Z_{ij}\rvert = \left(\lvert W_{ij}\rvert .MF + 2f\right) \gg (qbits + 1) \quad (2.37a)$$ $$sign\left(Z_{ij}\right) = sign\left(W_{ij}\right) \quad (2.37b)$$
4×4 FHT	$$\lvert Z_{ij}\rvert = \left(\lvert W_{DCij}\rvert .MF_{00} + 2f\right) \gg (qbits + 1) \quad (2.38a)$$ $$sign\left(Z_{ij}\right) = sign\left(W_{DCij}\right) \quad (2.38b)$$
2×2 FHT	$$\lvert Z_{ij}\rvert = \left(\lvert W_{DCij}\rvert .MF_{00} + 2f\right) \gg (qbits + 1) \quad (2.39a)$$ $$sign\left(Z_{ij}\right) = sign\left(W_{DCij}\right) \quad (2.39b)$$
4×4 IIT	$$W'_{ij} = Z_{ij}V_{ij} \ll \left\lfloor \frac{QP}{6} \right\rfloor \quad (2.40)$$
8×8 IIT	$$W'_{ij} = \left(Z_{ij}V_{ij} \ll \left\lfloor \frac{QP}{6} \right\rfloor\right) \gg 2 \quad (2.41)$$
4×4 IHT	$$W'_{ij} = \begin{cases} W'_{DCij}V_{00} \ll \left(\left\lfloor \frac{QP}{6} \right\rfloor - 2\right) & \text{if } QP \geq 12, \\ \left(W'_{DCij}V_{00} + 2^{1-\left\lfloor \frac{QP}{6} \right\rfloor}\right) \gg \left(2 - \left\lfloor \frac{QP}{6} \right\rfloor\right) & \text{otherwise.} \end{cases} \quad (2.42)$$
2×2 IHT	$$W'_{ij} = \begin{cases} W'_{DCij}V_{00} \ll \left(\left\lfloor \frac{QP}{6} \right\rfloor - 1\right) & \text{if } QP \geq 6, \\ W'_{DCij}V_{00} \gg 1 & \text{otherwise.} \end{cases} \quad (2.43)$$

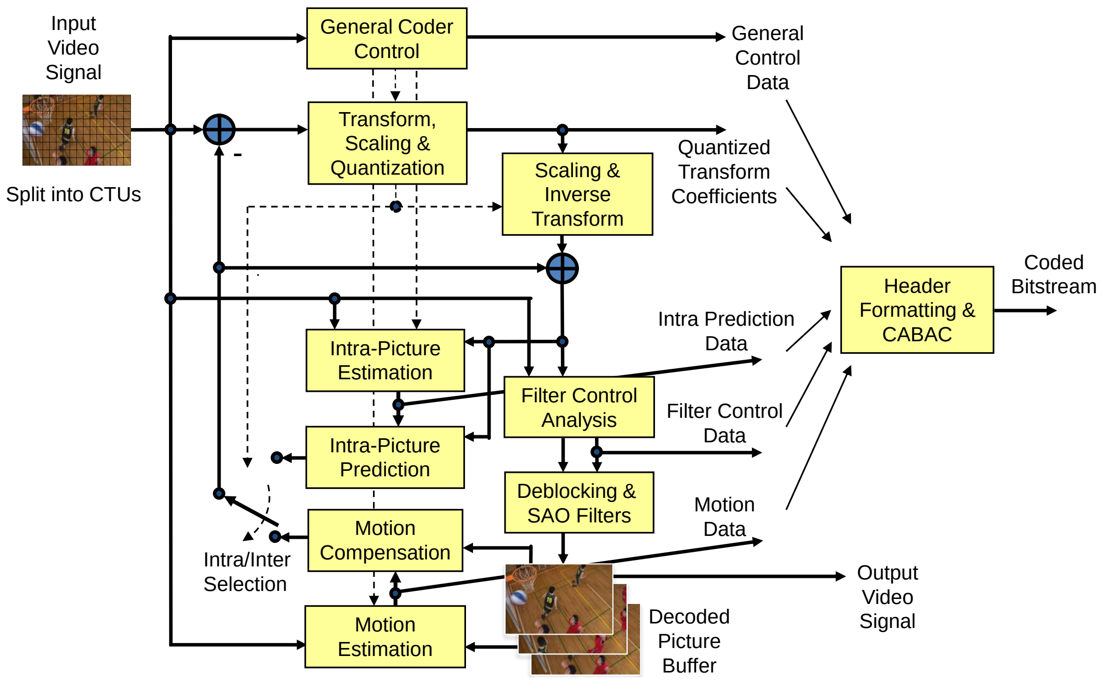


FIGURE 2.10: HEVC encoder block diagram (Sullivan et al., 2012).

Transform Blocks; (3) Motion Compensation; (4) Intra-Picture Prediction; (5) Entropy Coding; (6) In-loop Filtering; (7) Slices, Tiles and Wavefront; and (8) High-level Syntax.

In the previous standards, "macroblock" is the core of the coding layer. It consists of a 16×16 block of luma samples and, if the typical "4:2:0" color sampling is used, two corresponding 8×8 blocks of chroma samples and associated syntax elements. On the contrary, the equivalent structure in HEVC is the coding tree unit (CTU), which includes coding tree blocks (CTBs) for luma and chroma. Since a luma CTB has a larger size $L \times L$ with L = 16, 32, or 64 samples, better compression is typically achieved. Based on a quadtree structure, CTBs are partitioned into coding blocks (CBs) (Figure 2.12). The luma and chroma CBs are then split together. A set of one luma CB and the two corresponding chroma CBs with associated syntax elements is termed a coding unit (CU). A CU can be as small

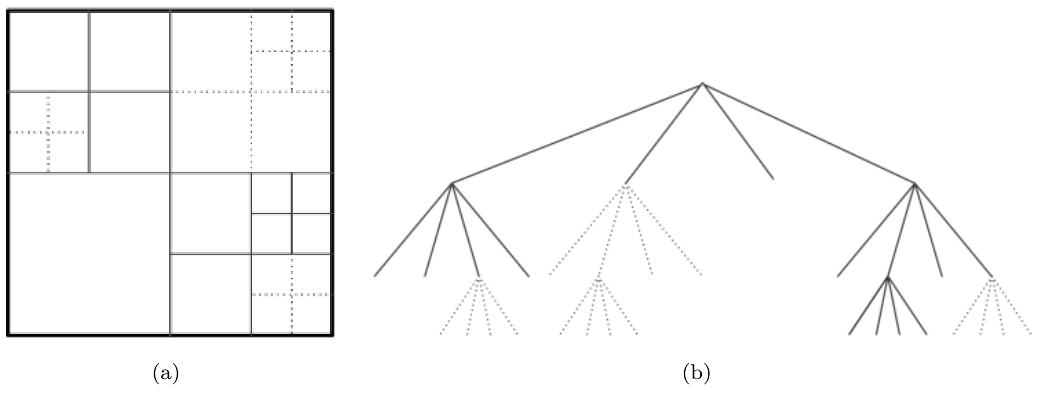


FIGURE 2.11: Subdivision of a 64×64 luma coding tree block (CTB) into coding blocks (CBs) and transform blocks (TBs) (Sullivan et al., 2012). Solid lines indicate CB boundaries and dotted lines indicate TB boundaries. (a) The CTB with its partitioning and (b) the corresponding quadtree. Luma CB can be as small as 8×8 where TB can be minimum 4×4 in size. In this example, the leaf CBs and TBs are 8×8 in size.

as a combination of a 8×8 luma CB and two 4×4 chroma CBs with associated syntax elements.

Block transforms are used to code the prediction residual. The CU is at the root of a transform unit (TU) tree structure, and the CBs may be further split into smaller transform blocks (TBs) of 4×4, 8×8, 16×16, or 32×32.

2.1.3.1 Core Transform

Like the H.264 standard, the 2-D transforms in HEVC are computed by applying the 1-D transforms in both the horizontal and vertical directions. The core transform matrices were derived by approximating scaled DCT basis functions to integer values under considerations maximizing the precision and proximity to orthogonality and limiting the dynamic range for transform computation. HEVC transform matrices can be found in Figure 2.12.

$$\mathrm{H}_{4x4} = \begin{bmatrix} 64 & 64 & 64 & 64 \\ 83 & 36 & -36 & -83 \\ 64 & -64 & -64 & 64 \\ 36 & -83 & 83 & -36 \end{bmatrix}$$

(a)

$$\mathrm{H}_{8x8} = \begin{bmatrix} 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 \\ 89 & 75 & 50 & 18 & -18 & -50 & -75 & -89 \\ 83 & 36 & -36 & -83 & -83 & -36 & 36 & 83 \\ 75 & -18 & -89 & -50 & 50 & 89 & 18 & -75 \\ 64 & -64 & -64 & 64 & 64 & -64 & -64 & 64 \\ 50 & -89 & 18 & 75 & -75 & -18 & 89 & -50 \\ 36 & -83 & 83 & -36 & -36 & 83 & -83 & 36 \\ 18 & -50 & 75 & -89 & 89 & -75 & 50 & -18 \end{bmatrix}$$

(b)

$$\mathrm{H}_{16x16} = \begin{bmatrix} 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 \\ 90 & 87 & 80 & 70 & 57 & 43 & 25 & 9 & -9 & -25 & -43 & -57 & -70 & -80 & -87 & -90 \\ 89 & 75 & 50 & 18 & -18 & -50 & -75 & -89 & -89 & -75 & -50 & -18 & 18 & 50 & 75 & 89 \\ 87 & 57 & 9 & -43 & -80 & -90 & -70 & -25 & 25 & 70 & 90 & 80 & 43 & -9 & -57 & -87 \\ 83 & 36 & -36 & -83 & -83 & -36 & 36 & 83 & 83 & 36 & -36 & -83 & -83 & -36 & 36 & 83 \\ 80 & 9 & -70 & -87 & -25 & 57 & 90 & 43 & -43 & -90 & -57 & 25 & 87 & 70 & -9 & -80 \\ 75 & -18 & -89 & -50 & 50 & 89 & 18 & -75 & -75 & 18 & 89 & 50 & -50 & -89 & -18 & 75 \\ 70 & -43 & -87 & 9 & 90 & 25 & -80 & -57 & 57 & 80 & -25 & -90 & -9 & 87 & 43 & -70 \\ 64 & -64 & -64 & 64 & 64 & -64 & -64 & 64 & 64 & -64 & -64 & 64 & 64 & -64 & -64 & 64 \\ 57 & -80 & -25 & 90 & -9 & -87 & 43 & 70 & -70 & -43 & 87 & 9 & -90 & 25 & 80 & -57 \\ 50 & -89 & 18 & 75 & -75 & -18 & 89 & -50 & -50 & 89 & -18 & -75 & 75 & 18 & -89 & 50 \\ 43 & -90 & 57 & 25 & -87 & 70 & 9 & -80 & 80 & -9 & -70 & 87 & -25 & -57 & 90 & -43 \\ 36 & -83 & 83 & -36 & -36 & 83 & -83 & 36 & 36 & -83 & 83 & -36 & -36 & 83 & -83 & 36 \\ 25 & -70 & 90 & -80 & 43 & 9 & -57 & 87 & -87 & 57 & -9 & -43 & 80 & -90 & 70 & -25 \\ 18 & -50 & 75 & -89 & 89 & -75 & 50 & -18 & -18 & 50 & -75 & 89 & -89 & 75 & -50 & 18 \\ 9 & -25 & 43 & -57 & 70 & -80 & 87 & -90 & 90 & -87 & 80 & -70 & 57 & -43 & 25 & -9 \end{bmatrix}$$

(c)

$$\mathrm{H}_{32x32} = \begin{bmatrix} 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 & 64 \\ 90 & 90 & 88 & 85 & 82 & 78 & 73 & 67 & 61 & 54 & 46 & 38 & 31 & 22 & 13 & 4 & -4 & -13 & -22 & -31 & -38 & -46 & -54 & -61 & -67 & -73 & -78 & -82 & -85 & -88 & -90 & -90 \\ 90 & 87 & 80 & 70 & 57 & 43 & 25 & 9 & -9 & -25 & -43 & -57 & -70 & -80 & -87 & -90 & -90 & -87 & -80 & -70 & -57 & -43 & -25 & -9 & 9 & 25 & 43 & 57 & 70 & 80 & 87 & 90 \\ 90 & 82 & 67 & 46 & 22 & -4 & -31 & -54 & -73 & -85 & -90 & -88 & -78 & -61 & -38 & -13 & 13 & 38 & 61 & 78 & 88 & 90 & 85 & 73 & 54 & 31 & 4 & -22 & -46 & -67 & -82 & -90 \\ 89 & 75 & 50 & 18 & -18 & -50 & -75 & -89 & -89 & -75 & -50 & -18 & 18 & 50 & 75 & 89 & 89 & 75 & 50 & 18 & -18 & -50 & -75 & -89 & -89 & -75 & -50 & -18 & 18 & 50 & 75 & 89 \\ 88 & 67 & 31 & -13 & -54 & -82 & -90 & -78 & -46 & -4 & 38 & 73 & 90 & 85 & 61 & 22 & -22 & -61 & -85 & -90 & -73 & -38 & 4 & 46 & 78 & 90 & 82 & 54 & 13 & -31 & -67 & -88 \\ 87 & 57 & 9 & -43 & -80 & -90 & -70 & -25 & 25 & 70 & 90 & 80 & 43 & -9 & -57 & -87 & -87 & -57 & -9 & 43 & 80 & 90 & 70 & 25 & -25 & -70 & -90 & -80 & -43 & 9 & 57 & 87 \\ 85 & 46 & -13 & -67 & -90 & -73 & -22 & 38 & 82 & 88 & 54 & -4 & -61 & -90 & -78 & -31 & 31 & 78 & 90 & 61 & 4 & -54 & -88 & -82 & -38 & 22 & 73 & 90 & 67 & 13 & -46 & -85 \\ 83 & 36 & -36 & -83 & -83 & -36 & 36 & 83 & 83 & 36 & -36 & -83 & -83 & -36 & 36 & 83 & 83 & 36 & -36 & -83 & -83 & -36 & 36 & 83 & 83 & 36 & -36 & -83 & -83 & -36 & 36 & 83 \\ 82 & 22 & -54 & -90 & -61 & 13 & 78 & 85 & 31 & -46 & -90 & -67 & 4 & 73 & 88 & 38 & -38 & -88 & -73 & -4 & 67 & 90 & 46 & -31 & -85 & -78 & -13 & 61 & 90 & 54 & -22 & -82 \\ 80 & 9 & -70 & -87 & -25 & 57 & 90 & 43 & -43 & -90 & -57 & 25 & 87 & 70 & -9 & -80 & -80 & -9 & 70 & 87 & 25 & -57 & -90 & -43 & 43 & 90 & 57 & -25 & -87 & -70 & 9 & 80 \\ 78 & -4 & -82 & -73 & 13 & 85 & 67 & -22 & -88 & -61 & 31 & 90 & 54 & -38 & -90 & -46 & 46 & 90 & 38 & -54 & -90 & -31 & 61 & 88 & 22 & -67 & -85 & -13 & 73 & 82 & 4 & -78 \\ 75 & -18 & -89 & -50 & 50 & 89 & 18 & -75 & -75 & 18 & 89 & 50 & -50 & -89 & -18 & 75 & 75 & -18 & -89 & -50 & 50 & 89 & 18 & -75 & -75 & 18 & 89 & 50 & -50 & -89 & -18 & 75 \\ 73 & -31 & -90 & -22 & 78 & 67 & -38 & -90 & -13 & 82 & 61 & -46 & -88 & -4 & 85 & 54 & -54 & -85 & 4 & 88 & 46 & -61 & -82 & 13 & 90 & 38 & -67 & -78 & 22 & 90 & 31 & -73 \\ 70 & -43 & -87 & 9 & 90 & 25 & -80 & -57 & 57 & 80 & -25 & -90 & -9 & 87 & 43 & -70 & -70 & 43 & 87 & -9 & -90 & -25 & 80 & 57 & -57 & -80 & 25 & 90 & 9 & -87 & -43 & 70 \\ 67 & -54 & -78 & 38 & 85 & -22 & -90 & 4 & 90 & 13 & -88 & -31 & 82 & 46 & -73 & -61 & 61 & 73 & -46 & -82 & 31 & 88 & -13 & -90 & -4 & 90 & 22 & -85 & -38 & 78 & 54 & -67 \\ 64 & -64 & -64 & 64 & 64 & -64 & -64 & 64 & 64 & -64 & -64 & 64 & 64 & -64 & -64 & 64 & 64 & -64 & -64 & 64 & 64 & -64 & -64 & 64 & 64 & -64 & -64 & 64 & 64 & -64 & -64 & 64 \\ 61 & -73 & -46 & 82 & 31 & -88 & -13 & 90 & -4 & -90 & 22 & 85 & -38 & -78 & 54 & 67 & -67 & -54 & 78 & 38 & -85 & -22 & 90 & 4 & -90 & 13 & 88 & -31 & -82 & 46 & 73 & -61 \\ 57 & -80 & -25 & 90 & -9 & -87 & 43 & 70 & -70 & -43 & 87 & 9 & -90 & 25 & 80 & -57 & -57 & 80 & 25 & -90 & 9 & 87 & -43 & -70 & 70 & 43 & -87 & -9 & 90 & -25 & -80 & 57 \\ 54 & -85 & -4 & 88 & -46 & -61 & 82 & 13 & -90 & 38 & 67 & -78 & -22 & 90 & -31 & -73 & 73 & 31 & -90 & 22 & 78 & -67 & -38 & 90 & -13 & -82 & 61 & 46 & -88 & 4 & 85 & -54 \\ 50 & -89 & 18 & 75 & -75 & -18 & 89 & -50 & -50 & 89 & -18 & -75 & 75 & 18 & -89 & 50 & 50 & -89 & 18 & 75 & -75 & -18 & 89 & -50 & -50 & 89 & -18 & -75 & 75 & 18 & -89 & 50 \\ 46 & -90 & 38 & 54 & -90 & 31 & 61 & -88 & 22 & 67 & -85 & 13 & 73 & -82 & 4 & 78 & -78 & -4 & 82 & -73 & -13 & 85 & -67 & -22 & 88 & -61 & -31 & 90 & -54 & -38 & 90 & -46 \\ 43 & -90 & 57 & 25 & -87 & 70 & 9 & -80 & 80 & -9 & -70 & 87 & -25 & -57 & 90 & -43 & -43 & 90 & -57 & -25 & 87 & -70 & -9 & 80 & -80 & 9 & 70 & -87 & 25 & 57 & -90 & 43 \\ 38 & -88 & 73 & -4 & -67 & 90 & -46 & -31 & 85 & -78 & 13 & 61 & -90 & 54 & 22 & -82 & 82 & -22 & -54 & 90 & -61 & -13 & 78 & -85 & 31 & 46 & -90 & 67 & 4 & -73 & 88 & -38 \\ 36 & -83 & 83 & -36 & -36 & 83 & -83 & 36 & 36 & -83 & 83 & -36 & -36 & 83 & -83 & 36 & 36 & -83 & 83 & -36 & -36 & 83 & -83 & 36 & 36 & -83 & 83 & -36 & -36 & 83 & -83 & 36 \\ 31 & -78 & 90 & -61 & 4 & 54 & -88 & 82 & -38 & -22 & 73 & -90 & 67 & -13 & -46 & 85 & -85 & 46 & 13 & -67 & 90 & -73 & 22 & 38 & -82 & 88 & -54 & -4 & 61 & -90 & 78 & -31 \\ 25 & -70 & 90 & -80 & 43 & 9 & -57 & 87 & -87 & 57 & -9 & -43 & 80 & -90 & 70 & -25 & -25 & 70 & -90 & 80 & -43 & -9 & 57 & -87 & 87 & -57 & 9 & 43 & -80 & 90 & -70 & 25 \\ 22 & -61 & 85 & -90 & 73 & -38 & -4 & 46 & -78 & 90 & -82 & 54 & -13 & -31 & 67 & -88 & 88 & -67 & 31 & 13 & -54 & 82 & -90 & 78 & -46 & 4 & 38 & -73 & 90 & -85 & 61 & -22 \\ 18 & -50 & 75 & -89 & 89 & -75 & 50 & -18 & -18 & 50 & -75 & 89 & -89 & 75 & -50 & 18 & 18 & -50 & 75 & -89 & 89 & -75 & 50 & -18 & -18 & 50 & -75 & 89 & -89 & 75 & -50 & 18 \\ 13 & -38 & 61 & -78 & 88 & -90 & 85 & -73 & 54 & -31 & 4 & 22 & -46 & 67 & -82 & 90 & -90 & 82 & -67 & 46 & -22 & -4 & 31 & -54 & 73 & -85 & 90 & -88 & 78 & -61 & 38 & -13 \\ 9 & -25 & 43 & -57 & 70 & -80 & 87 & -90 & 90 & -87 & 80 & -70 & 57 & -43 & 25 & -9 & -9 & 25 & -43 & 57 & -70 & 80 & -87 & 90 & -90 & 87 & -80 & 70 & -57 & 43 & -25 & 9 \\ 4 & -13 & 22 & -31 & 38 & -46 & 54 & -61 & 67 & -73 & 78 & -82 & 85 & -88 & 90 & -90 & 90 & -90 & 88 & -85 & 82 & -78 & 73 & -67 & 61 & -54 & 46 & -38 & 31 & -22 & 13 & -4 \end{bmatrix}$$

(d)

FIGURE 2.12: Transform matrices in HEVC. (a) $H_{4\times4}$; (b) $H_{8\times8}$; (c) $H_{16\times16}$; and (d) $H_{32\times32}$.

Since the size of the supported transforms is increased, it is important to limit the dynamic range of the intermediate results in transform computation right from the first stage. For 8-bit HEVC video decoding, a 7-bit right shift and 16-bit clipping operation are explicitly inserted after the first 1-D vertical inverse transform stage to guarantee that all intermediate values can be stored in 16-bit memory.

2.1.3.2 Mode-Dependent Alternative Transform

For intra prediction modes, an alternative integer transform derived from a discrete sine transform (DST) is applied to the 4×4 luma residual blocks with the transform matrix

$$H = \begin{bmatrix} 29 & 55 & 74 & 84 \\ 74 & 74 & 0 & -74 \\ 84 & -29 & -74 & 55 \\ 55 & -84 & 74 & -29 \end{bmatrix}. \qquad (2.44)$$

The new transform based on DST is more suitable to model the fact that the residual amplitudes tend to increase as the distance from the boundary samples becomes larger. It is also not more complex than the 4×4 DCT-style transform, and in intra-predictive coding, it provides about 1% bit-rate reduction.

It should be noted that the DST-style transform is used for only 4×4 luma transform blocks, since for other cases there is no significant improvement in efficiency if this additional transform type is used.

2.1.3.3 Scaling and Quantization

Unlike H.264/MPEG-4 AVC, HEVC does not need the pre-scaling operation since the rows of the transform matrix are close approximations of uniformly-scaled basis functions' values of the orthogonal DCT. This helps reduce the intermediate memory size, and especially is useful when the transform is large as 32×32.

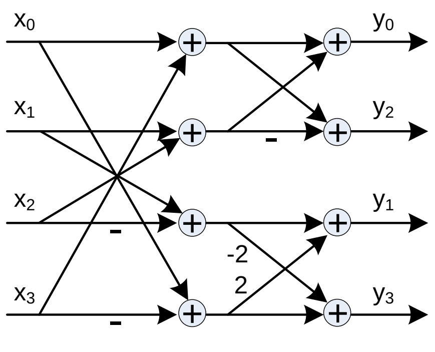
For quantization, as in H.264/MPEG-4 AVC, uniform-reconstruction quantization (URQ) is also used in HEVC. The range of the quantization parameter (QP) is defined from 0 to 51, and an increase by six doubles the quantization step size, such that the mapping of QP values to step sizes is approximately logarithmic.

Quantization scaling matrices are also supported in HEVC. However, to reduce the intermediate memory size, only 4×4 and 8×8 quantization matrices are used. For 16×16 and 32×32 transformations, an 8×8 scaling matrix is sent and is applied by sharing values within 2×2 and 4×4 coefficient groups in frequency sub-spaces except for values at DC positions, for which distinct values are sent and applied.

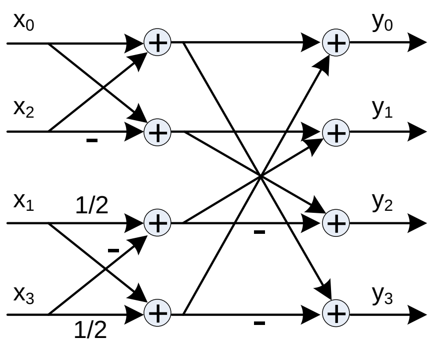
2.2 Related Works

2.2.1 H.264/AVC Integer Transform

In December 2001, ITU-T and MPEG established a Joint Video Team (JVT) to develop the H.264/AVC video coding standard, which was finalized in March 2003 and approved in May 2003. H.264/AVC achieves a compression efficiency of up to 50% of bit-rates with much higher visual quality compared to the previous standards. Its decoder complexity is about four times of that of MPEG-2 and two times of that of the MPEG-4 Visual Simple Profile Ostermann et al. (2004).



(a)



(b)

FIGURE 2.13: Fast implementation of the H.264 1-D 4 × 4 transforms (Malvar et al., 2003). (a) Forward transform algorithm and (b) inverse transform algorithm.

The 4 × 4 forward and inverse integer transforms (Equations (2.27) and (2.31)) were introduced together with their fast multiplication-free implementations (Figure 2.13) by Malvar et al. (2003) in a special issue of the IEEE Transactions on Circuits and Systems for Video Technology, which was dedicated to the newly released H.264/AVC video coding standard. The fast 1-D implementations (Figure 2.13) are to transform all data of a row or column in parallel. The 2-D transforms can be performed by repeatedly applying the corresponding 1-D transform for all rows/columns of the data block, then for all columns/rows. The fast 1-D 4 × 4 forward/inverse transform implementation only needs eight addition/subtractions with two shift operations to transform a 4-input row/column. In addition, it can complete a 1-D transform for a row/column after two addition/subtractions.

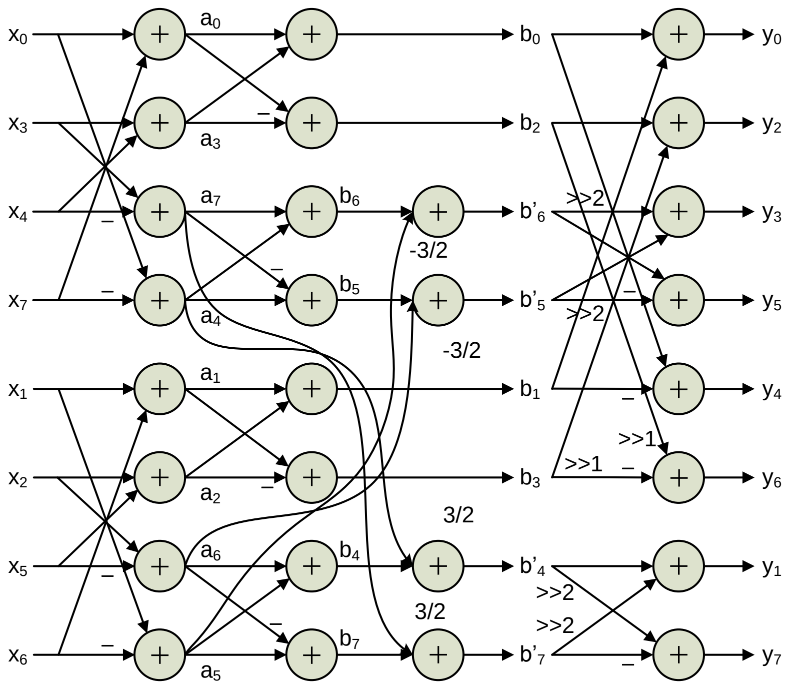
In 2004, Gordon et al. (2004) introduced the 8 × 8 forward and inverse integer transforms (Equations (2.28) and (2.32)) together with their fast multiplication-free implementations for the H.264/AVC high profiles. Similar to the fast 4 × 4 implementations, the fast 8 × 8 1-D implementations transform all data of a row or column in parallel. The 2-D transforms also can be performed by repeatedly applying the corresponding 1-D transform for all rows/columns of the data block, then for all columns/rows. The fast 1-D 8×8 forward/inverse transforms implementation needs thirty-two addition/subtractions with fourteen shift operations to transform a 4-input row/column. In addition, it can complete a 1-D transform for a row/column after five addition/subtractions.

Because the H.264/AVC high profiles (Marpe et al., 2005) target higher-fidelity video materials, their transform designs are challenging to achieve sufficiently high throughput. For hardware designs, not only high throughput but also area efficiency must be taken into account due to hardware cost. High throughput and area efficiency for hardware designs in general and integer transform hardware designs in particular can be achieved through two levels: the *algorithm* level and the *architecture* level.

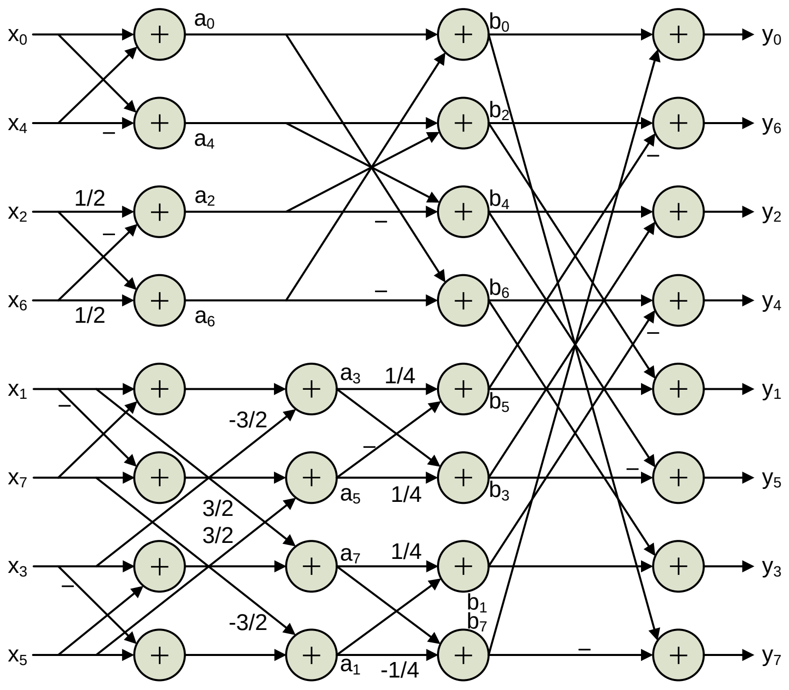
At the *algorithm* level, fast and low-cost algorithms for all the forward and inverse transforms, including the 4×4, 8×8 integer transforms and the 2×2 and 4×4 Hadamard transforms, have been provided by the H.264/AVC developers.

At the *architecture* level, many techniques have been used to increase design throughputs, where most of the architectures are based on those fast algorithms. The fast algorithms can be considered as the “official” transform algorithms because they were proposed by the JVT.

Many designs in the literature (Raja et al., 2005; Kordasiewicz and Shirani, 2005; Amer et al., 2005) process an entire 4×4 or 8×8 data block at one time by using eight or sixteen transform units, respectively. Here, a transform unit is the hardware component corresponding to 1-D transform a row or a column.



(a)



(b)

FIGURE 2.14: Fast 1-D 8 × 8 FIT/IIT algorithms adopted from Gordon et al. (2004). (a) FIT and (b) IIT.

Raja et al. (2005) proposed an architecture supporting the 4×4 forward transform and quantization. The architecture includes eight 1-D transform units, a large quantization module and a RAM to input, process and output sixteen pixels in parallel. It can complete a 2-D transform in one clock cycle.

Kordasiewicz and Shirani (2005) reported two 4×4 forward transform architectures supporting quantization: one is area-optimized and the other is speed-optimized. The speed-optimized design is based on the official fast algorithms. Sixteen inputs are fetched at the same time. Similar to the design by Raja et al. (2005), this architecture includes eight 1-D transform units and process the 2-D 4×4 transform in one cycle. The design quantization takes another cycle. Therefore, the total delay to process a 2-D transform is two cycles.

Amer et al. (2005) proposed an 8×8 forward transform architecture supporting quantization. By using sixteen transform units with a quantization module, the architecture inputs sixty-four pixels in parallel, processes using the fast 8×8 algorithm (Gordon et al., 2004) in one cycle, quantizes in one cycle, and outputs sixty-four coefficients simultaneously. The transform and the quantization block are pipelined. With the 12-bit depth for each pixel, the data bus width is $64 \times 12 \times 2 = 1536$.

The architectures in this class, especially the architectures supporting 8×8 transforms, lead to large bus widths of at least 1280 bits. If the architectures are implemented as hard IP cores, a huge number of pins, and consequently, a huge number of I/O pads is required. Thus, it is impractical in hardware implementation. If the architectures are implemented as soft IP cores, the corresponding circuit areas are very large because of the huge number of wires in the high level metal layers. Although not being reported in the published scripts, wiring areas are actually much larger than the cell areas and significantly affect the total circuit areas of the designs. Moreover, the architectures themselves requires many hardware resources to transform all the rows/columns of a block concurrently. Therefore, this technique is inapplicable for H.264 high profile integer transform designs; and having reasonable bus width or I/O pin count must be taken into account when addressing the high throughput challenge for transform architecture designs.

In the literature, pipelining is another technique to increase the design throughput. For the H.264/AVC main profile transforms (4×4 in size), a pipelining mechanism for input, output and transform processes is applied in some designs (Chen et al., 2006; Shi et al., 2007). The pipelining mechanisms reduce the total delays and increase the throughputs and performances of the systems.

Chen et al. (2006) proposed a direct 2-D 4×4 forward and inverse transform architecture supporting 4×4 forward and inverse Hadamard transforms. The architecture inputs sixteen pixels simultaneously. By using four transform units and a pipelining mechanism among input, output and transform processes, the architecture requires only two cycles to complete a 2-D transform and outputs eight coefficients at one time.

Shi et al. (2007) proposed a transform architecture which can perform the 4×4 forward and inverse integer transform and the 4×4 and 2×2 forward and inverse Hadamard transforms. The shared architecture was developed based on the structural similarities among all the 1-D forward and inverse transforms. By using four 1-D shared transform units, the shared 1-D architecture can process a 2-D 4×4 transform in two cycles. With a pipelining mechanism for input (four cycles), output (four cycles) and transform (two cycles) processes, the architecture can complete a transform within four cycles. By sharing architecture, the design can achieve a relatively small area.

The pipelining mechanism is also applied for transform and quantization processes by some of the H.264/AVC high profile transform designs (8×8 in size). A design by Pastuszak (2008) supports all the forward and inverse transforms with quantization and rescaling, except for the 2×2 Hadamard transforms. The author deployed the similarity of the official fast algorithms for different transform sizes. Thus, one shared 8×8 forward transform unit can perform a 1-D 8×8 transform for a row/column, or two 1-D 4×4 row/column transforms; and one shared 8×8 inverse transform unit can perform a 1-D 8×8 transform, or a 1-D 4×4 for a row/column. The architecture contains eight forward/inverse transform units to process a 2-D 8×8 transform in two cycles. The quantization performs thirty-two coefficients at a time, leading to a delay of two cycles. The transform architecture inputs sixty-four pixels and output sixty-four coefficients at a time with a 16-bit depth for each pixel, while the quantization architecture inputs thirty-two transformed coefficients and output thirty-two quantized transformed coefficients at a time. Therefore, the total data bus width is 128 pixels. By pipelining the transform, quantization and middle-buffer data transferring processes, the architecture can transform an 8×8 block in two cycles. However, the I/O delay is not counted in this 8×8 transform architecture.

In addition, the reported designs supporting these pipelining mechanisms have very large bus widths, leading to extremely high wiring areas and difficulties in realizing hard IP cores.

In the literature, hardware-sharing techniques have been used for area-efficient designs. In some designs, the authors implemented the integer transforms by directly multiplying a row/column of the data blocks by a column/row of the transform matrices to output one element of a row/column at a time. The multiplications are implemented using only addition and shifts operations, and there are shared hardware components among them.

An area-optimized architecture supporting the 4×4 forward transform with quantization was reported by Kordasiewicz and Shirani (2005). The architecture is not based on the official fast algorithms. It performs multiplications and outputs one coefficient among sixteen coefficients of the 1-D transform at once. The multiplication is implemented by addition and shift operations. Because the coefficients are processed separately, the multiplication architecture can be shared among them, leading to a small area. The delay required to output each coefficient is seven cycles, where four cycles are reserved to fetch four inputs. The quantization module needs four cycles to process. Hence, the total delay to process a 2-D transform is $(7 \times 16) \times 2 + 4 = 228$ cycles.

Hardware resource can be shared among all the row/column multiplications in this class of direct-multiplication designs. However, their throughputs are relatively small (less than or equal to one pixel per cycle). Since high throughput is the first priority requirement for the H.264 high profiles in order to support high definition videos, this direct-multiplication technique is not suitable for high profile designs. Nevertheless, it is still useful for main profile designs.

In addition to the application to row/column multiplications, the hardware-sharing technique can be deployed among large and small sizes of the transforms (Chao et al., 2007; Pastuszak, 2008; Su and Fan, 2008).

A design by Chao et al. (2007) supports all types of the inverse transforms without rescaling function. The inverse transform architecture has 16-pixel-wide input and

8-pixel-wide output, leading to a total data bus width of twenty-four pixels. By deploying similarities among the direct 2-D 4×4 inverse transform, the direct 2-D inverse Hadamard transforms and the official fast 1-D 8×8 inverse algorithm, the architecture uses one 8×8 shared transform unit to perform a 1-D 8×8 transform for a 8-input row/column, or a half of 2-D 4×4 transform for a 4×4 data block, or a 2-D 2×2 inverse Hadamard transform for a 2×2 block. The architecture also includes an 8×8 transpose buffer. For each 8×8 block, it requires sixteen cycles to process and three cycles for pipelining.

The architecture reported by Pastuszak (2008) deployed the similarity of the official fast algorithms for different transform sizes. Thus, one shared 8×8 forward transform unit can perform a 1-D 8×8 transform for a row/column, or two 1-D 4×4 row/column transforms; and one shared 8×8 inverse transform unit can perform a 1-D 8×8 transform, or a 1-D 4×4 for a row/column.

A design by Su and Fan (2008) supports all the inverse transforms of H.264 and AVS and without rescaling. Su and Fan (2008) decomposed the transform matrices into common simple matrix components. The shared architecture can perform each 1-D transform in four cycles. With a 12-bit-per-pixel 64-pixel input data bus and a 18-bit-per-pixel 64-pixel output data bus, the 2-D architecture uses two 1-D transform units and it requires sixteen cycles to complete an 8×8 transform.

Hardware-sharing can be deployed not only among different sizes of the transforms, but also among the forward and inverse transforms (Shi et al., 2007; Choi et al., 2008).

The shared architecture proposed by Shi et al. (2007) was developed based on the structural similarities among all the 1-D forward and inverse transforms. By using this shared architecture, the design can achieve a relatively small area.

A design by Choi et al. (2008) supports most all the forward and inverse transforms without quantization and rescaling, except for the 2×2 Hadamard transform. By sharing the architecture among the 4×4 with 8×8 transforms, and among the forward and inverse transforms, the 1-D architecture can perform any 1-D transform of a row or column in one cycle. By using four transform units with a 32-pixel input and a 32-pixel output data bus and a transpose buffer, it can process the 1-D 4×4 and 8×8 transforms in one and two cycles, respectively. The design requires five cycles to complete an 8×8 transform.

In a number of H.264/AVC designs (Wang et al., 2003; Kordasiewicz and Shirani, 2005; Chao et al., 2007; Choi et al., 2008; Park and Ogunfunmi, 2009; Hu et al., 2009), data are presumed available for processing at the transform modules, and delays due to the data transfer are not taken into account when evaluating performances. This leads the fact that the reported throughputs are higher than the actual throughputs of the implementations.

Wang et al. (2003) proposed a 2-D 4×4 forward and inverse transform architecture. The architecture can additionally perform the 4×4 forward Hadamard transform. It contains two 1-D transform units and a transpose register array, where each 1-D transform unit performs the 4×4 fast algorithms for four parallel inputs. The transform units are designed to complete the 1-D transforms of a row or column in one cycle. The architecture inputs four pixels simultaneously and finishes each 2-D transform in four cycles.

Park and Ogunfunmi (2009) proposed an 8×8 forward transform architecture supporting quantization. It is based on the official fast algorithm. The architecture includes two 1-D transform blocks and a transpose logic with a quantization block. It inputs a row/column of eight pixels concurrently and processes its 1-D 8×8

transform in one cycle. However, the 1-D block outputs only one coefficient at once to the transpose logic, leading to the processing delay of sixty-four cycles for the 1-D transform and the pipelined 2-D transform.

Hu et al. (2009) proposed an architecture supporting the 4×4 and 8×8 inverse transforms, the 4×4 inverse Hadamard transform and the rescaling. It processes a transform for each row/column in a cycle and takes seventeen cycles to complete the 8×8 inverse transform including rescaling (one cycle).

The research on the state-of-the-art designs for high throughputs shows that there is a strong need of developing integer transform designs with reasonable bus widths producing high throughputs, in which I/O delays are included.

The research on the state-of-the-art designs for area efficiency shows that there is still lack of shared architectures sharing hardware among all the forward transforms and among all the inverse transforms including the 4×4 and 2×2 Hadardmard transforms.

Integer transform modules eventually need to be integrated into H.264 video encoders and decoders together with other modules. However, the reported transform designs in the literature have different signals and data transfer mechanisms. This makes it difficult to integrate or reuse the reported components. Therefore, another question also can be raised is that how to design portable and flexible architectures which can be easily integrated to other systems and reused in future?

Strongly motivated by the above gaps in the H.264/AVC integer transform designs, this study addresses the high throughput and area-efficient challenges through three novel transform architectures (Section 3.2, Section 3.3 and Section 4) with shared forward/inverse transform units (Section 3.2, Section 3.3 and Section 4), two different pipeline mechanisms (Section 3.3 and Section 4), special I/O buffers (Section 4) and standard bus system deployment (Section 3.2 and Section 3.3).

2.2.2 HEVC Integer Transform

In January 2010, ITU-T VCEG and ISO/IEC MPEG agreed to establish a Joint Collaborative Team on Video Coding (JCT-VC) (Ohm and Sullivan, 2013) and a joint call for proposals on video compression technology (ITU-T and ISO/IEC JTC1, 2010) was issued. The first meeting was held in April 2010 with an agreement on the HEVC project name. In the meeting, submitted proposals in response to the proposal call were studied. The first version of "Test Model Under Consideration" (TMuC) was established from elements of several promising proposals, which were discussed by Wiegand et al. (2010). In October 2010, redundant coding features and low benefit-complexity-rate features were removed from TMuC, producing an HEVC test model version 1 (HM). The HEVC design has been continually updated to Draft 9 of the standard (Bross et al., 2012) and HM 9 (ITU-T and JTC1, 2012). The first version of the HEVC standard is scheduled to be produced in early 2013.

The goal of HEVC is to achieve a reduction by about half of the bit-rate for equivalent visual quality compared to H.264/AVC. The current HMs are roughly meeting or exceeding the targeted goal. From the conducted experiments, the approximate average bit-rate reduction for equivalent subjective quality was 67% for full HD sequences, 49% for WVGA sequences, and 58% overall. The PSNR bit-rate savings of HEVC for the same quality was reported to be around 35-40% compared to the H.264/AVC High Profiles and 70-80% compared to the MPEG-2 Main Profile (Ohm et al., 2012).

Complexity considerations are analyzed in depth by Bossen et al. (2012). Overall, HEVC decoder complexity is about 1.5 times of that of H.264/AVC, while HEVC encoders are expected to be several times more complex than H.264/AVC encoders. It is also predicted that due to the complexity, HEVC encoders is a subject of research for years to be realized.

Two series of algorithms are being used in HMs to implement transform coding for HEVC in four sizes from 4×4 to 32×32. The first series is purely matrix multiplications, while the second one, Partial Butterfly series, includes butterfly additions and scalar multiplications. The latter are developed based on the symmetry properties of the HEVC transform matrices. The former is easier to implement but very computationally complex and slow, while the latter requires more efforts to implement but has far fewer mathematical operations (Sullivan et al., 2012). This is the reason why the Partial Butterfly series is used for all quality and complexity tests of HMs.

HEVC aims to support beyond-full high resolution videos. Hence, HEVC designs are required to achieve much higher throughputs than those of H.264 designs. In addition, the HEVC transforms are much more complex than the H.264 transforms due to its much larger transform sizes (up to 32×32) and much larger matrix coefficient values (up to 90). Both the large increment in the supported resolutions and the large increment in complexity make it more challenging to develop high-throughput and area-efficient designs for HEVC than H.264/AVC.

Similar to H.264/AVC, for HEVC integer transform hardware designs, two design levels can be considered to achieve high throughput and area efficiency: the *algorithm* level and the *architecture* level.

At the *algorithm* level, Partial Butterfly algorithms have been deployed for the HEVC transforms in HEVC test model HM (ITU-T and JTC1, 2012). The Partial Butterfly algorithms are implemented using butterfly additions and multiplications. Compared to the transform implementation by multiplications, this series of algorithms is much less complex. However, it is still far more complex than the equivalent-size transform algorithms of H.264. This is because the H.264/AVC transform algorithms contain simple sequences of shift and addition operations, while the Partial Butterfly algorithms for HEVC contain a number of multiplications with large multipliers. Therefore, less complex and faster transform algorithms using only additions and shift operations are obviously needed.

To the best of our knowledge, until now (December 2012), there are not many HEVC transform algorithms proposed in the literature. The techniques of manipulating and decomposing the up-to-8×8 transform matrices of HEVC based on the H.264 transform matrices have been reported to develop fast and low-complexity transform algorithms for HEVC. As a result, the transforms can be implemented using addition and shift operations.

In November 2012, Rithe et al. (2012), based on the structural similarity and symmetry of the HEVC, H.264 and VC-1 transform matrices, proposed a shared algorithm and architecture to perform 4×4 and 8×8 transforms for the three standards. The algorithm and architecture was developed based on decompositions of the 4×4 and 8×8 transform matrices into small-size simple matrix components, where VC-1 transforms can share several components with those of H.264 and those of HEVC can share several components with those of the other two standards. These decompositions, at the same time, also enable the implementation of the transforms by using only addition and shift operations. The proposed algorithm and architecture of HEVC requires eighteen and sixty additions with the longest operation path of four and six additions for the 4×4 and 8×8 transforms, respectively. However, this decomposition method cannot guarantee to be extendable to larger sizes of the HEVC transforms due to difficulties in finding small-size simple matrix components for decompositions. In addition, the authors did not implement their HEVC design in hardware.

Martuza and Wahid (2012) approached the problem solution in a different way. Instead of decomposing the HEVC transform matrices into simple matrix components, the authors directly used the H.264 8×8 inverse transform matrix for HEVC 8×8 transform implementation with a compensation by the difference between the two transform matrices. Both H.264 matrix multiplication and the matrix difference are then implemented using addition and shift operations. The H.264 matrix multiplication implementation is not based on the recommended simplified algorithms for H.264/AVC (Gordon et al., 2004), which 1-D transforms eight inputs in parallel to produce eight outputs simultaneously. It directly multiplies eight inputs by each column of the transform matrix to produce an output at a time. The sequenced outputs of the algorithm facilitate hardware sharing among the column multiplications. Therefore, although the algorithm consumes seventy-two addition/subtractions with the longest path of at least seven addtion/subtractions, the shared architecture can achieve a relative small resource consumption of thirty-two adders. However, the architecture requires eight cycles to 1-D transform eight inputs, where in each cycle, at least seven additions are required to be performed. This leads to a low throughput as a trade-off for the cost reduction. Since the HEVC standard enables capability to compress ultra high definition videos, high throughput should be the highest priority among all the aspects. In addition, this design also cannot be extendable to larger sizes of the HEVC transforms as H.264 does not have larger-than-8×8 transforms.

Both of the above designs are based on H.264 transform matrices to manipulate HEVC matrices so that the HEVC matrix multiplications can be implemented using addition and shift operations. At the same time, due to the common operations, the HEVC and H.264 transforms can share some architecture components. However, this H.264-shared approach may lead to a longer running time and larger resource consumption for HEVC transform designs because the shared algorithms and architectures are normally not as optimized as the individual optimal algorithms and architecture designs.

At the *architecture* level, techniques to improve design throughput have not reported yet in the literature. Similar to H.264/AVC, in order to reduce the area, hardware-sharing techniques are reported. Martuza and Wahid (2012) implemented the integer transforms using direct multiplications to share the hardware resource among all the coefficient computations. However, this direct-multiplication technique normally produces small throughputs of one pixel per cycle or even less. In addition, hardware-sharing among large and small sizes of the transforms is reported (Rithe et al., 2012).

Therefore, together with HEVC finalization, there is an urgent need of developing fast algorithms and high-throughput architectures for HEVC transforms. As the size and complexity of the 32×32 transform in HEVC are large, it is difficult to manually develop such algorithms. This study researches methods to develop fast transform algorithms for all the sizes of the HEVC transforms, which facilitate high throughput designs. In this study, we propose a novel method to automatically find fast algorithms even for the 32×32 transforms. Based on the proposed method, a series of 4×4 and 8×8 hardware-oriented fast and low-cost transform algorithms for HEVC is developed. Fast and low-cost larger size transform algorithms also can be developed using the same method. A high-throughput and area-efficient architecture with hardware implementation is finally developed, implemented and fabricated based on the proposed algorithms.

Chapter 3

Portable Inverse Transform Architectures for H.264/AVC

3.1 Introduction

H.264/AVC (Wiegand et al., 2003b,c) provides better video quality at substantially lower bit-rate than the previous standards by the adoption of a number of new coding tools. One of the coding tools, the integer transform, is based on the Discrete Cosine Transform of the previous standards, e.g. MPEG-4 (ISO/IEC 14496-2, 1999), with the avoidance of the mismatch problem between encoder and decoder transforms and the reduction of the computational complexity. Four transforms are used in total, depending on each type of residual data: a Hadamard transform (HT) for the 4×4 array of luminance DC coefficients in intra macroblocks predicted in the 16×16 mode, a HT for the 2×2 array of chrominance DC coefficients in every macroblock, and 4×4 and 8×8 integer transforms (ITs) for all blocks in the residual data in fidelity range extensions (FRExt) (Gordon et al., 2004). Both 4×4 and 8×8 transforms can be chosen dynamically based on the prediction modes and the sub-partition sizes to improve encoder performance and flexibility.

Integer transform modules need to be integrated with other modules to form the entire encoders or decoders or even larger systems. They might also need to be integrated into testing systems. In order to enable the integration to a larger system, each module needs to have a proper interface with a hand-shaking protocol and signals. The component modules in the same system should share the same handshaking protocol to alleviate integration problems. However, the state-of-the-art transform designs (Wang et al., 2003; Raja et al., 2005; Amer et al., 2005; Kordasiewicz and Shirani, 2005; Chen et al., 2006; Shi et al., 2007; Chao et al., 2007; Pastuszak, 2008) have not addressed this portability and flexibility issue. Data is assumed available on data bus without any effort to communicate and fetch data, which is not practical.

In this chapter, we address the portability for integer transform designs in particular and IP designs in general by deploying an interconnection standard (bus standard). The standard specifies the bus architectures, signal names and handshaking protocols so that the IP modules which conform to the same interconnection standard can be easily integrated. Several on-chip bus standards can be found in the literature such as the Wishbone Interconnection Architecture by Silicore (OpenCores, 2002), Advanced Micro-controller Bus Architecture (AMBA) by ARM (ARM, 1999), CoreConnect by IBM (IBM, 1999) and Avalon by Altera (ALTERA, 2003). Some of them are open standards and some are commercial. Among the bus standards, the Wishbone standard out-performs others for its open property, simplicity, flexibility and portability (Sharma and Kumar, 2012). Therefore, it has been selected to be deployed in our two portable designs to achieve portability.

On the other hand, the forward integer transforms are used in H.264 video encoders, while the inverse integer transforms are used in both H.264 encoder and decoders. This motivates the study in this chapter to concentrate on the inverse integer transforms. The H.264/AVC high profiles (Marpe et al., 2005) target higher-fidelity videos for applications like professional film production, video post production, or high-definition TV/DVD. Therefore, inverse integer transform designs are challenging to achieve high throughput, which is computed as the number of pixels processed in an unit of time. In addition, the hardware designs also need high area efficiency to reduce costs.

These challenges can be addressed through two design levels: the *algorithm* level and the *architecture* level. In H.264/AVC, at the *algorithm* level, fast and low-cost transform algorithms have been introduced by the H.264/AVC developers, and most of the reported designs follow these algorithms. At the *architecture* level, two techniques for high throughput were reported in the literature. The first technique (Amer et al., 2005; Kordasiewicz and Shirani, 2005; Raja et al., 2005) is to input a entire coefficient block, process and output all inverse-transformed (and rescaled) data in parallel. This leads to unpractical wide I/O bus. The second technique (Amer et al., 2005; Kordasiewicz and Shirani, 2005; Raja et al., 2005; Pastuszak, 2008) is to deploy pipelining mechanisms among operating processes. This reduces total delays of all the processes, and consequently, increase throughputs of the architectures. However, the reported designs employing this technique have very large bus widths, which produces the unpractically large wiring areas for their hardware implementations. Therefore, there is a strong need of having the integer transform designs with a reasonable bus width but still achieving high throughputs.

For area-efficient architectures, different hardware-sharing techniques are reported. Hardware-sharing can be done among (1) the coefficient computations (Kordasiewicz and Shirani, 2005) or (2) all the inverse transforms (Chao et al., 2007; Pastuszak, 2008). Technique (1) leads to a very low throughput as only one coefficient is outputted at one time. The architectures using technique (2) have not included the Hadamard transforms in the shared hardware unit yet.

In this study, besides addressing the portability for inverse integer transform designs, we propose a shared hardware unit among all the inverse integer transforms including the 4×4 and 8×8 inverse transforms and the 2×2 and 4×4 inverse Hadamard transforms of DC coefficients. This shared hardware unit, named computing kernel, not only targets area efficiency by fully embedding the smaller-size transforms into the larger-size ones, but also facilitates high-throughput designs by duplicating the 4×4 inverse transform module. This utilizes the bus width for the 8×8 transform and enables two 4×4 transforms to be performed simultaneously.

Based on the shared unit, we also propose two portable inverse integer transform (IIT) architecture designs for H.264/AVC in this chapter, under the condition of reasonable bus widths. The first design includes an IIT IP block with an instance of the proposed shared hardware unit and rescaling module. Thanks to the hardware unit, the IP block achieves a very reasonable area. The IIT block, which conforms to the Wishbone shared bus specification for portability, is integrated in a System-on-Chip controlled by an Application-Specific Instruction Set Processor (ASIP), allowing functional testability and design flexibility.

While the first portable design focuses on portability and area efficiency, the second design targets portability and high throughput. Thanks to an efficient pipelining mechanism, the second architecture can achieve a high throughput with a reasonable area compared to other designs in the literature. The architecture includes an ASIP, two DMA controllers and two instances of the proposed IIT IP block with rescheduled timing. For the portability and pipeline support, all the modules in this architecture conform to the Wishbone crossbar switch bus specification.

The chapter is organized as follows. Section 3.2 describes the first portable design targeting area efficiency. Section 3.3 describes the second portable design targeting high throughput. The chapter ends with a summary in Section 3.4.

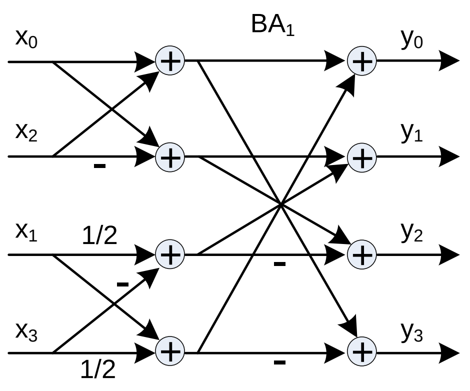
3.2 Portable Area-Efficient Inverse Integer Transform Architecture Design

3.2.1 Proposed Inverse Integer Transform Unit

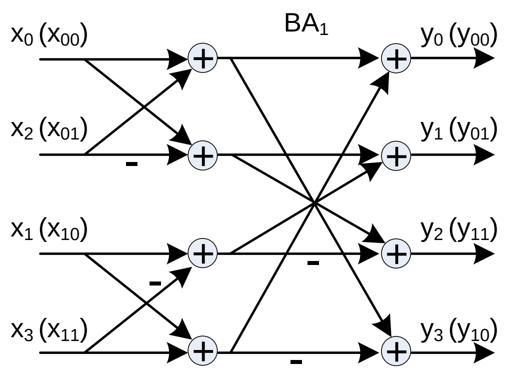
In the H.264/AVC standard, the 2-D forward transforms are computed using the row-column decomposition technique. That means a 2-D transform is performed as a 1-D vertical (column) transform followed by a 1-D horizontal (row) transform. Since the column and row transformations use the same algorithm, the identical 1-D core can be used.

In 2003, a series of fast and low-cost algorithms for the 4×4 forward and inverse integer transforms of H.264/AVC (Equations (2.27) and (2.31)) were introduced by Malvar et al. (2003). In 2004, Gordon et al. (2004) introduced the 8×8 forward and inverse integer transforms (Equations (2.28) and (2.32)) together with their fast and low-cost algorithms for the H.264/AVC high profiles. These fast and low-cost inverse algorithms are illustrated in Figure 3.1.

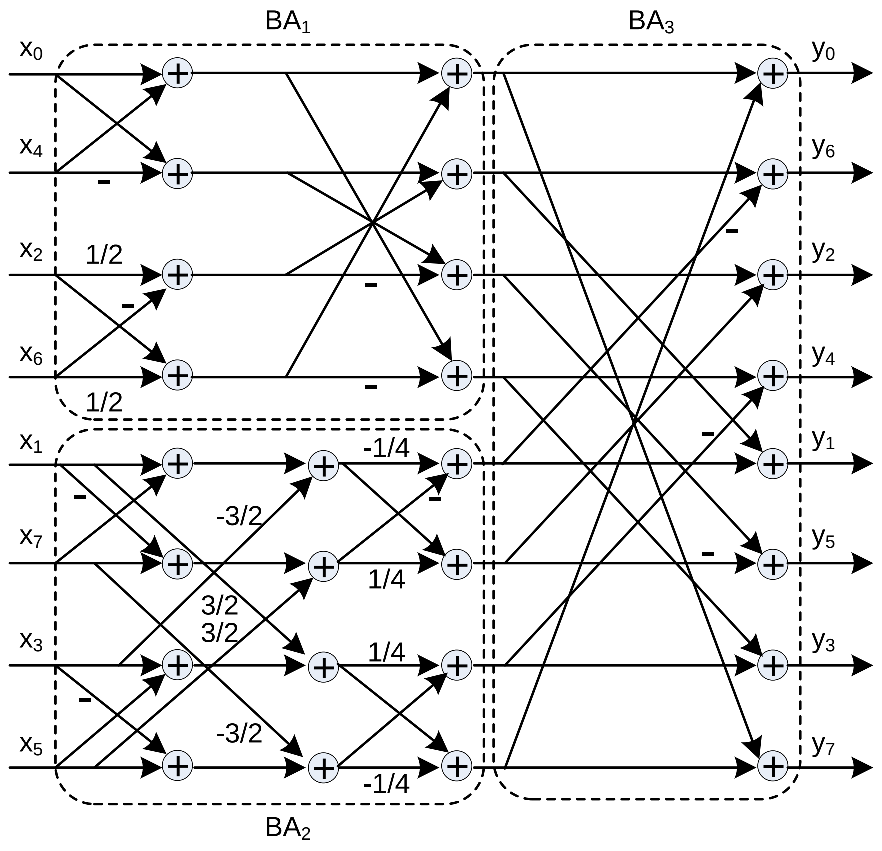
As can be seen from the figure, the 1-D 4×4 IIT (Figure 3.1(a)) and Hadamard transform (Figure 3.1(b)) algorithms are similar, except for two different coefficients. In addition, the 1-D 8×8 IIT algorithm (Figure 3.1(c)) has sub-block BA_1, which is similar to the 1-D 4×4 IIT. Especially, after some manipulations, the 2-D



(a)



(b)



(c)

FIGURE 3.1: Dataflow diagrams of H.264 inverse integer transforms. (a) 1-D 4×4 inverse integer transform; (b) 1-D 4×4 inverse Hadamard transform (2-D 2×2 inverse Hadamard transform); and (c) 8×8 1-D inverse integer transform.

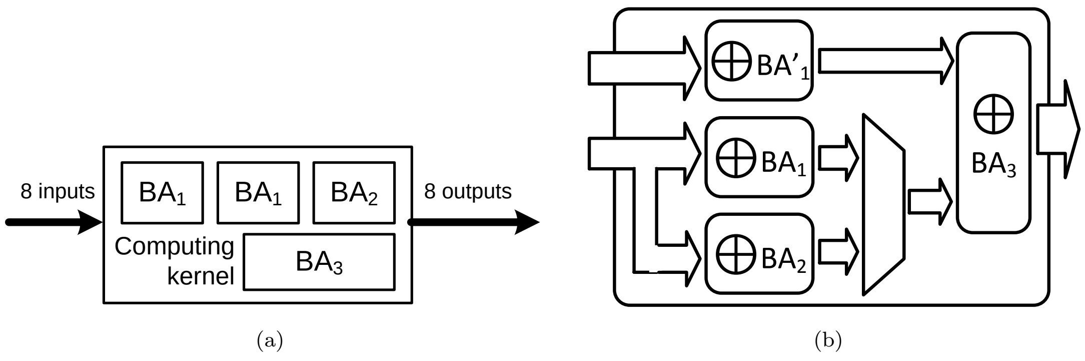


FIGURE 3.2: The proposed shared inverse integer transform hardware unit (computing kernel). (a) Block diagram; and (b) Dataflow diagram.

2×2 inverse Hadamard transform can be performed using the same architecture as the architecture of the 1-D 4×4 inverse Hadamard transform (Figure 3.1(b)).

Based on this analysis, a shared inverse integer transform unit (Figure 3.2(a)) is proposed. The unit contains a hardware resource to compute the 1-D 8×8 IIT for an 8-input row/column. It includes four adder blocks: two BA$_1$s, one BA$_2$, and one BA$_3$. These BA$_1$, BA$_2$, and BA$_3$ are corresponding to the BA$_1$, BA$_2$, and BA$_3$ in Figure 3.1(c), respectively. Adder block BA$_1$ is used to compute all the 1-D 4×4 transform types, including the 4×4 inverse transform and the 4×4 inverse Hadamard transform, and the 2-D 2×2 inverse Hadamard transform. Depending on the transform mode, the proper coefficients ($\frac{1}{2}$ for integer and 1 for Hadamard transform) are selected accordingly. To utilize the 8-value I/O bus for the 4×4 cases and facilitate high-throughput, BA$_1$ is duplicated. Hence, with two BA$_1$s, two row/columns of $2 \times 2/4 \times 4$ blocks then can be transformed simultaneously.

Figure 3.2(b) shows the dataflow diagram of the proposed inverse integer transform hardware unit. In the 4×4 mode, eight inputs are divided into two parts: four inputs are fed to adder block BA$_1$, while the others are fed to adder block BA'$_1$ (identical to adder block BA$_1$). Then, two 4×4 matrices are computed in parallel by adder blocks BA_1 and BA'_1. Afterwards, results from adder block BA_1 are passed through a multiplexer. Together with the results from adder block BA'_1, they are bypassed adder block BA_3 to output.

In the 8×8 transform mode, adder blocks BA_1, BA_2 and BA_3 are all enabled where adder blocks BA_2 and BA_3 are specifically designed for the 8×8 transform. Adder block BA_1, which was previously used to compute the 4×4 transform, now is reused to compute four pixels of even rows. The other four pixels of odd rows are processed in adder block BA_2. Adder block BA_3 receives the data from adder blocks BA_1 and BA_2, computes the final step and sends data out.

In total, eight multipliers, forty adders, and twenty shifters are required by the proposed inverse transform unit.

3.2.2 Proposed Inverse Integer Transform Architecture

The proposed shared transform unit (Section 3.2.1), is employed in this IIT architecture for area efficiency. Since the bit-depth requirement for the H.264/AVC standard (Wiegand et al., 2003b) is twelve bits to the input of the inverse integer transforms, implementation of a 128-bit data bus allows data transferring at maximum eight coefficients per clock cycle (if sixteen bits are assigned to each coefficient). This means one column of an 8×8 matrix or two columns of a 4×4 matrix can be processed at a time. Compared to the 64-bit data bus, the 128-bit data bus helps to minimize the bottleneck and provides nearly two times increase in I/O throughput, but with a slightly larger gate count.

In the proposed inverse transform architecture (Figure 3.3), there are one 8-input multiplier block, one shifter, one inverse transform unit, one rounding and one

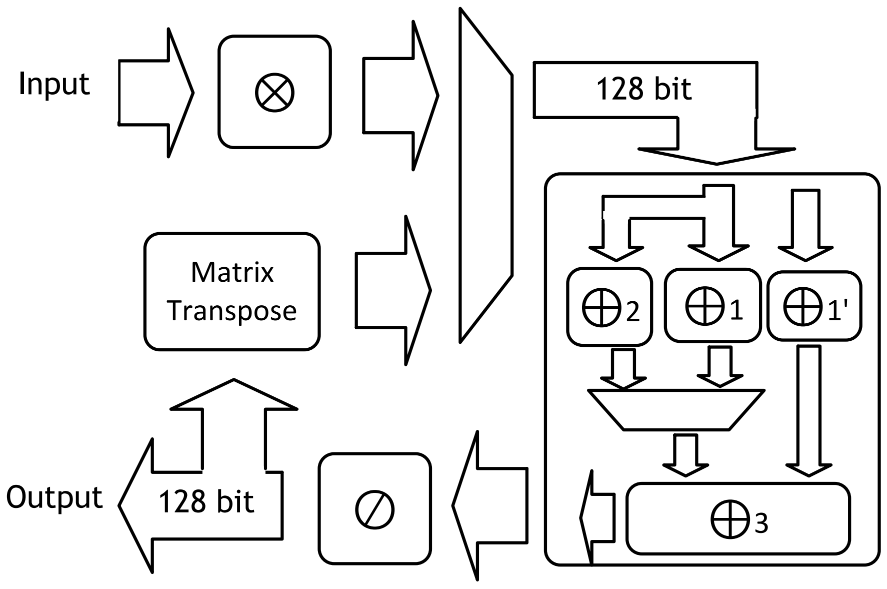


FIGURE 3.3: The proposed inverse integer transform architecture.

matrix transposer. A 2-D inverse transform usually needs to be performed by two 1-D transforms (column and row transforms) at two stages. When the proposed inverse transform block is operating, at the first 1-D transform stage, inputs are fetched to the rescaling multipliers. Based on the value of QP, multiplicative factors are generated and used inside the multiplier for rescaling data. Next, data are passed through a multiplexer to the inverse transform unit for the first 1-D matrix multiplication. Data are then bypassed in the rounding block and stored into the matrix transposer. At this point, rows and columns are exchanged for the second 1-D transform stage. At the second stage, a multiplexer selects the data coming from the transposer and passes them through processing steps including core transforming (transform unit), rounding, and sending results to the output.

The detailed operations of the proposed architecture are described as follows. The input data is first rescaled. This is an important step for all transform modes where the input matrix must be multiplied by the scaling factor in the same position in matrix E_f. This multiplier block contains eight parallel multipliers so as it can take full advantage of the 128-bit wide data bus. Pipelined multi-stage multipliers are employed to decrease the combinational delay in one clock cycle. This block is also responsible for QP DIV 6 and QP MOD 6 operations, and the corresponding multiplicative factors for the transform. In the inverse transforms, the input data is scaled by a factor of less than or equal to 58. Therefore, the multiplier requires only six bits. This helps to minimize the hardware cost.

After being rescaled in the multiplication block, results are passed into the transform unit. The results are then moved to the rounding block for a division. It divides the values by 64, which can be realized with a arithmetic right shifting by 6. This block is only enabled in the 4×4 mode. A simple multiplexer can be used to bypass the division in other transform configurations. From here we get the final outputs of the inverse integer transforms.

The transposer requires a total of 64×16-bit registers in order to transpose a maximum of sixty-four pixels of an 8×8 matrix. This block is shared between all the 4×4 and 8×8 transforms to minimize the number of registers. With some additional multiplexers, the row and column pixels are exchanged. Since the second 1-D transform stage requires one row to operate, four clock cycles delay are expected to perform the 4×4 transform. Eight cycles for the 8×8 transform and the same number of cycles are required to completely send out the data. To utilize the system and reduce the number of cycles for these transforms, pipelining is used to increase the throughput of the whole system.

Table 3.1 shows the number of clock cycles required for each transform. Since pipeline technique is being used, only the first block takes a long time to setup, subsequent continuous blocks are faster in process.

TABLE 3.1: First transform design clock cycle report.

Transform	First block	Next block
$2 \times (2 \times 2$ Hadamard transform$)$	4	1
$2 \times (4 \times 4$ Hadamard transform$)$	13	11
$2 \times (4 \times 4$ integer transform$)$	13	11
8×8 integer transform	21	19

3.2.3 Proposed IIT System Architecture

3.2.3.1 Top-Level System Architecture

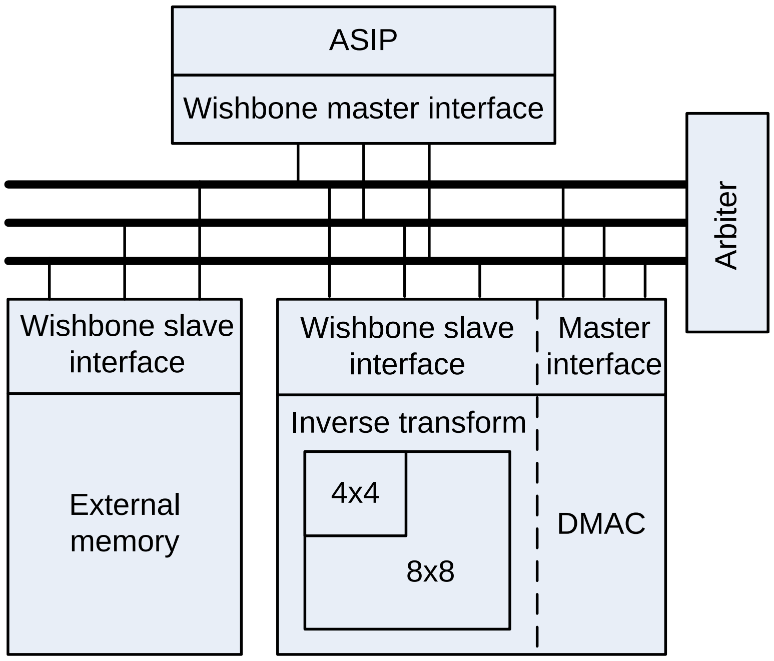
For portability, all the modules in the proposed system should conform to a bus standard. Among the bus standards, the Wishbone standards surpasses others thanks to its open property, simplicity, flexibility and portability (Sharma and Kumar, 2012). As a result, a Wishbone shared bus system is deployed in this architecture.

The top-level system architecture - depicted in Figure 3.4(a) - includes an ASIP, an external memory, and the proposed inverse transform block with its DMA controller. The data, address, and control buses are regulated by an Arbiter. The Wishbone system bus signals at the master and slave interfaces are shown in Figure 3.4(b).

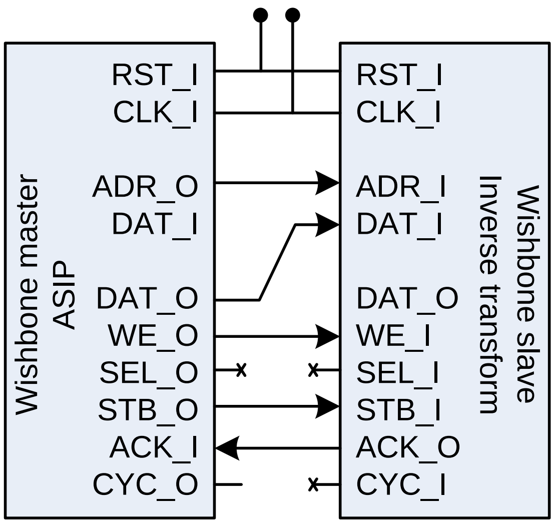
3.2.3.2 ASIP and Instruction Set

The inverse integer transform block is controlled by an ASIP. The ASIP architecture is based on a RISC stored program machine (SPM) (Ciletti, 2003) with a specific instruction set to control up to four IP blocks.

Figure 3.5 shows the architecture of the supporting ASIP. The ASIP consists of three parts: a controller, a datapath and an internal memory. The controller



(a)



(b)

FIGURE 3.4: (a) The proposed top-level architecture with shared bus system and (b) Wishbone interface signals.

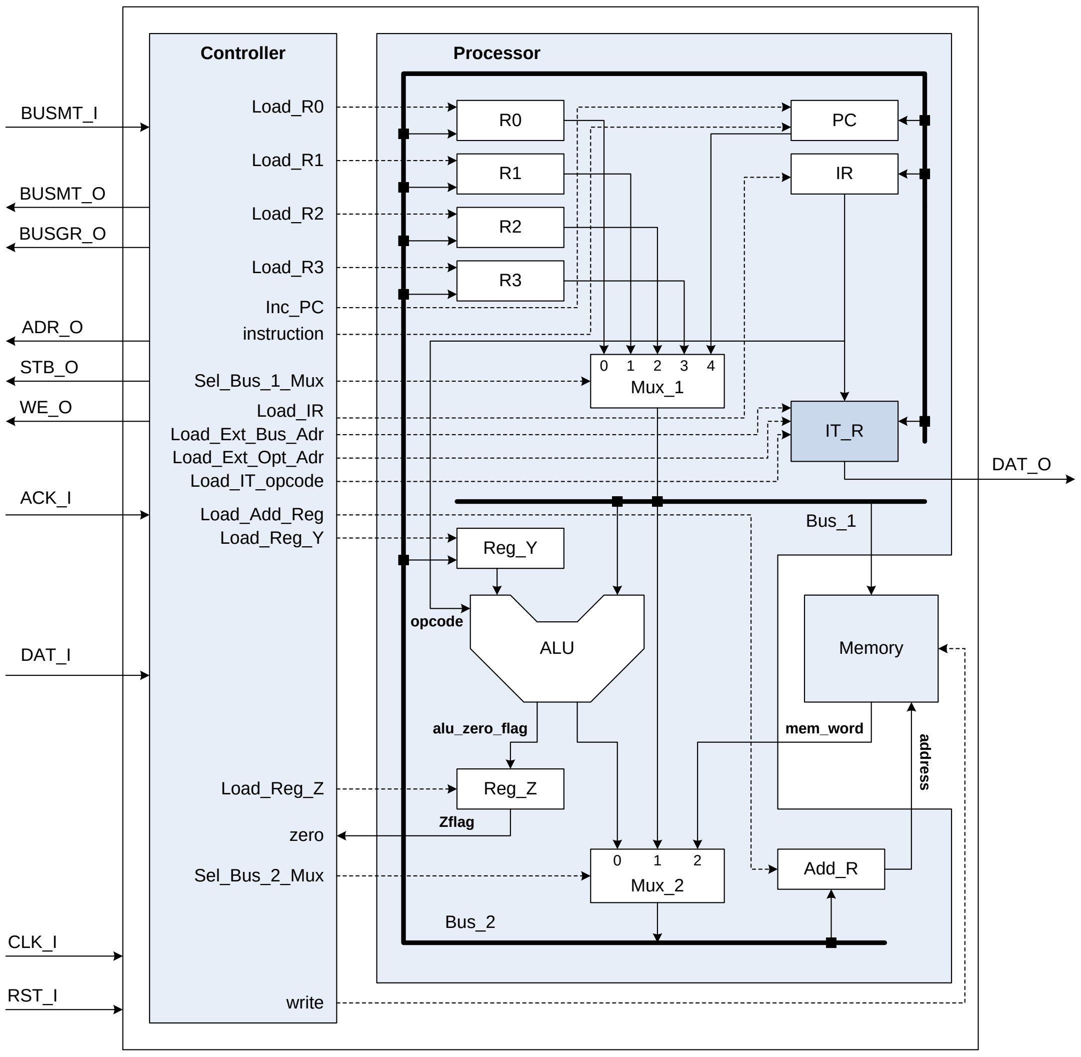


FIGURE 3.5: The proposed ASIP architecture.

generates signals to orchestrate the operation of the ASIP. The datapath comprises an ALU, ten registers, and two multiplexers. The ten registers include four general-purpose registers (R_0, R_1, R_2, R_3), a Zero flag (Reg_Z), a temporary register (Reg_Y), a program counter (PC), an instruction register (IR), and an address register (Add_R). Add_R is used to store memory address information prior to memory write or memory read.

Table 3.2: The proposed 24-bit IIT instruction in RISC SPM.

Opcode				Src		Dst		Ext. mem. src. addr.								Ext. mem, dst. addr							
0	0	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0

Table 3.3: Newly defined instructions for the inverse transforms.

Instruction	Machine byte 1			Byte 2	Byte 3	Action
	Opcode	Src	Dst	E-src	E-dest	
IIT	1010	src	???	E-src	E-dest	Exec inverse integer transform

The modified ASIP has ten instructions, in which the newly define instruction IIT is used to invoke the inverse integer transform function.

The ASIP operates through three phases: fetch, decode, and execute. In the fetch phase, it retrieves an instruction from memory in two clock cycles. In the decode phase, it decodes the instruction, manipulates the datapath, and loads registers using one cycle. In the execution phase, it executes and generates the results within zero to five cycles.

The RISC SPM has three types of instructions: 8-bit, 16-bit, and 24-bit long instructions. The 8-bit instructions are intended for basic arithmetic operations. The 16-bit instructions are for accessing the internal memory, and the 24-bit instructions are for accessing the external memory. A 24-bit instruction is illustrated in Table 3.2.

In this instruction, Src is used to indicate the four types of the inverse integer transforms: the 4×4 integer transform, the 4×4 Hadamard transform, the 2×2 Hadamard transform and the 8×8 integer transform.

3.2.3.3 The Inverse Integer Transform Block

The inverse integer transform block includes an IIT component with a Wishbone (OpenCores, 2002) slave interface to communicate with its Wishbone master interface of the ASIP, and a DMAC with a Wishbone master interface to directly transfer image data from/to the external memory.

3.2.3.4 System-on-Chip Shared Bus

The ASIP, the external memory, and the IT blocks are connected using a SoC bus, which is controlled by an arbiter. The arbiter is to decide which master will transfer data at a certain time.

In the proposed system, the ASIP is a bus master while the external memory and the IP block are bus slaves. However, this special IP block is designed with a built-in DMAC which helps to improve the transfer speed of image data from the external memory to IP block, and vice versa. The DMAC itself also is a bus master of its own block. As a result, the system has two bus masters and two bus slaves in total.

3.2.3.5 ASIP and Hand-Shaking Issues with IP block(s)

The ASIP and the IT block are connected via a SoC-based Wishbone system bus, whose master and slave interfaces are shown in Figure 3.4(b).

In the interface, ASIP sends strobe signal STB_O, write enable signal WE_O, IT slave address through ADR_R and command through DAT_O to all the slaves. The IT block then reads the slave address in the address bus and decides if it is the target block. This slave immediately sends back an acknowledge signal to ACK_I of the ASIP to indicate that it is ready, and read all the necessary data in the bus. After receiving the acknowledge signal, the ASIP sends BUSMT_O signal with the new bus master code, which represents the above IT block, to the arbiter. Because the IT block has a Wishbone master interface and a Wishbone slave interface, so at the next cycle, the above IT block will become the bus master of the system and has full right to use the system bus. And the ASIP continues its operation on its internal memory and registers.

Upon completion, the master interface of the IT block will send CYC_O to the arbiter to signal its completion. After receiving this signal, the arbiter will re-assign the ASIP as the default system bus master.

3.2.4 Experimental Results and Discussion

3.2.4.1 Functionality, Testability and Portability

The proposed design has been implemented with the 4×4 inverse transform for the residual data; the 4×4 and 2×2 inverse Hadamard transforms for DC coefficients; and the newly added 8×8 inverse transform.

A quantization block is also integrated into our design. The hardware sharing methodology of different types of transform reduces the system area at the cost of a few extra clock cycles. Based on the instruction analysis, this cost is more affordable and timing requirement can be met at a slightly higher operating frequency.

Some video test sequences are converted into bit streams and the complete tests are conducted on the ASIP. Because the proposed IP block of the inverse integer transforms is designed on an SoC environment deploying the Wishbone system bus, it can be ported to other SoC platforms of similar system bus characteristics.

3.2.4.2 Performance and Discussion

The proposed inverse integer transform and inverse quantization architecture with the ASIP, embedded DMAC, RAM and Wishbone shared-bus system were implemented in Verilog, and verified with RTL simulations using Mentor Graphics ModelSim. System-level functional verification was also performed by passing different input patterns to the architecture and comparing the output with the results obtained by Matlab. It was pre-layout synthesized using AMS 0.35μm technology library by Synopsys Design Compiler. The gate count is 21.5k for the inverse integer transforms in combination with inverse quantization at 150 MHz. Table 3.4 lists the reported designs with their performances.

In Table 3.4, the reported forward and inverse integer transform designs are listed in chronological order, and if two designs are reported on the same year, the FPGA design is listed first, followed by the ASIC design. We try to list the reported designs, but not all designs can be compared due to different reported parameters.

It should be noted that among all the listed designs, only the proposed design conforms to a bus standard for portability, testability and flexibility.

As can be seen from the table, among the designs supporting the 8×8 transforms, an implementation of the 8×8 integer transform followed by quantization was

TABLE 3.4: Performance comparisons of pre-layout synthesis results.

Design	Type of transform	Tech. (μm)	Gate [a] (10^3)	Speed (MHz)	I/O Throughput/ Throughput (ppc)	Quant.
Wang et al. (2003)	4×4 [fi]	0.35	6.5	80	4/4	No
Raja et al. (2005)	4×4 [f]	Virtex II	3.2K slices + 4.1K ff	127	16/16	Yes
Amer et al. (2005)	8×8 [f]	Virtex II	29K LUT	68	64/64	Yes
Kordasiewicz and Shirani (2005)	4×4 [f]	Virtex II	1.6K slcs + 0.5K ff	97	16/8	Yes
Kordasiewicz and Shirani (2005)	4×4 [f]	0.18	51.6	68 [c]	16/8	Yes
Chen et al. (2006)	4×4 [fih4]	0.18	6.4	100	16(I)/ 8(O)/8	No
Shi et al. (2007)	4×4 [ih42]	0.18	5.0	166	8/4	No
Chao et al. (2007)	All [d]	0.18	9.5	125	8/3.4	No
Proposed	All [d]	Virtex 4	2.0K slcs + 1.2K ff	145	8/3.4	Yes
Proposed	All [d]	0.35	21.5	150	8/3.4	Yes
Proposed	All [d]	0.35	8.3	150	8/3.4	No

[a] Gate count is computed as the total combinational area normalized by the area of a 2-input NAND gate.

[c] Speed is computed based on the critical delay of speed-optimized Quantization circuit. This is the slower of the two DCT and Quantization circuits.

[d] 'All' means all 4×4, 8×8 integer transforms, and 4×4, 2×2 Hadamard transform are supported.

[fi] Forward and inverse transform.

[f] Forward transform.

[fih4] Forward and inverse transform with additional 4×4 Hadarmard transforms.

[hih42] Forward and inverse transform with additional 4×4 and 2×2 Hadarmard transforms.

reported by Amer et al. (2005). However, the support for complete coding of residual data was not implemented.

In the design by Chao et al. (2007) and this design, complete support for coding of residual data is implemented. In terms of circuit area, the proposed design outperforms the design by Chao et al. (2007) in its relatively smaller area. The proposed design without quantization consumes 8.3 Kgates, which is 87.4% of the gate count of the design by Chao et al. (2007).

For real-time requirement, the proposed architecture can process all existed frame sizes (Wiegand et al., 2003b,c). For example, if the clock frequency is setup at 146 MHz, it can transform video with the resolution up to 16.4 Mpels, e.g., 4096×2304 (4K) pels or 5120×3200 (WHXGA) pels, and progressive scan frequency of 30 Hz (30 frames per second). "pel" stands for pixel element.

3.3 Portable High-Throughput Inverse Transform Architecture

While the first portable design focuses on the portability and area efficiency, the second portable design targets high throughput in addition to the portability. Based on the proposed inverse transform unit (Section 3.2.3.4), in order to achieve high throughput, the second design deploys a pipelining mechanism for all the modules in the architecture.

3.3.1 Proposed Inverse Integer Transform Architecture

In this design, the proposed inverse transform architecture includes the same modules as the proposed inverse transform architecture in Section 3.2.2. However, the bus width is redesigned at 96-bit input and 96-bit output (Figure 3.6) as 128 bits used in the first portable design is waste when the H.264 requirement for bit-depth is twelve bits. In addition, the blocks in the architecture are rescheduled as follows.

The inverse transform block is executed in four major stages: rescaling, 1-D transforms of columns in the computing kernel, 1-D transforms of rows previously stored in the internal RAM by looping back to the computing kernel, and rounding off the results. In the first stage, an 8-value column (of 8×8 block) or two 4-value columns (of 4×4 blocks) are fed to the rescaling multipliers, where multiplicative factors are generated based on the value of QP (Section 2.1.2.2). In the second stage, data are passed through a multiplexer to the computing kernel for 1-D column transform. The results are stored into the internal RAM. The operations of both stages are performed in one cycle, and repeated until all the columns are processed. In the third stage, an 8-value row (of 8×8 block) or two 4-value rows are fed from the internal RAM to CK for 1-D row transformation followed by a divide-by-64 rounding operation in the rounder block in the forth stage. Similarly, the operations in the third and forth stages are finished in one cycle and repeated until all the rows are processed. By this rescheduling, the proposed IIT architecture (module) can perform all four inverse transform types and rescaling, i.e., scaling or inverse quantization. A 2×2 data block can be processed in one cycle, 4×4 block in four cycles, and 8×8 in eight cycles, compared to one, eleven and nineteen cycles in the first portable design, respectively.

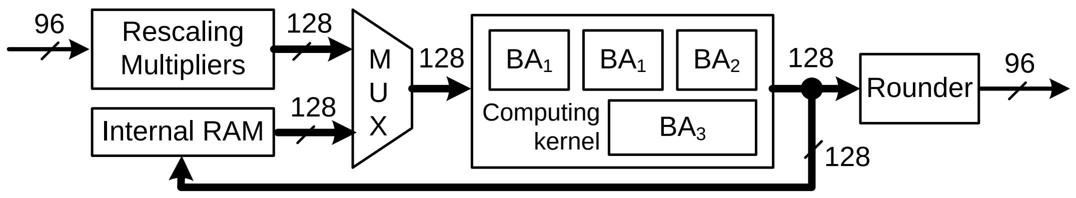


FIGURE 3.6: The proposed IIT architecture.

3.3.2 Proposed IIT System Architecture

H.264/AVC specifies a maximum of twelve bits per pixel value without compromising quality (Wiegand et al., 2003b). A 192-bit data bus is designed accordingly. I/OD for an 8×8 block is then eight cycles. Aggregate throughput is inversely proportional to total delay, and can be maximized by pipelining the input, processing, and output stages. The best pipeline performance can be achieved when all stages have the same minimum delay. In our proposed system, for 2×2, 4×4 and 8×8 blocks, I/ODs are one, two and eight cycles, while the PDs are two, four and sixteen cycles, respectively. As PD is twice as long as I/OD, it is best to incorporate two IITs.

3.3.2.1 Top-Level System Architecture

The proposed top-level architecture includes an ASIP, two external RAMs, two DMACs, and two IITs (Figure 3.7). The two external RAMs, one responsible for input data and the other for output data, are housed in the two DMACs for efficient data transfer. All modules are connected through a Wishbone two-channel crossbar switch, regulated by an arbiter (OpenCores, 2002) so that two slaves and two masters can work in parallel, facilitating the fully pipelining mechanism. By

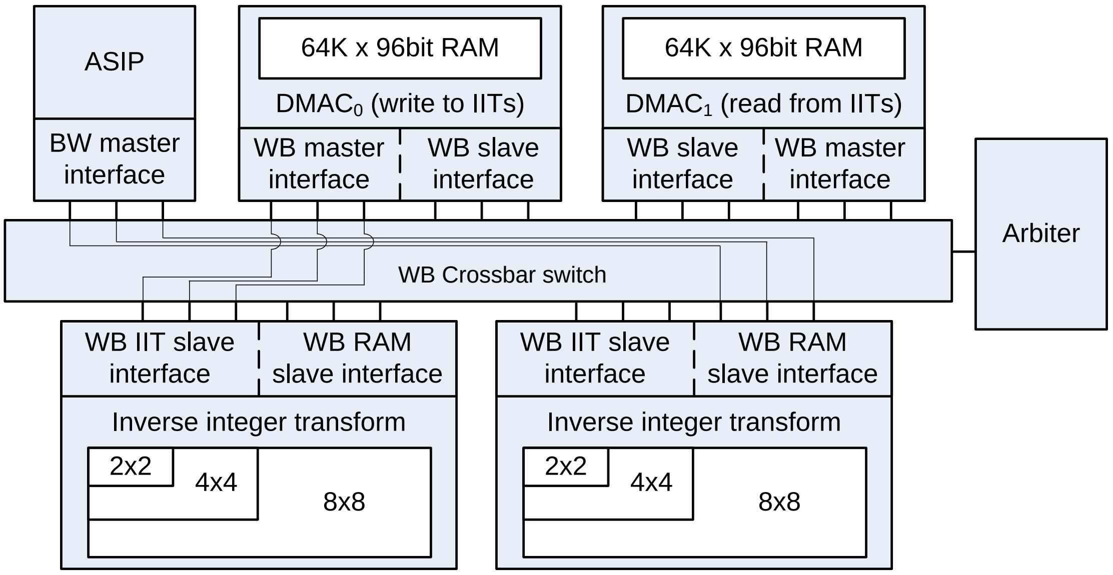


FIGURE 3.7: The proposed top-level architecture with crossbar switch bus system.

using two built-in-RAM DMACs to transfer data between RAMs, the two IIT blocks can be orchestrated to work concurrently.

The system can operate in two modes: instruction mode and automatic mode. In the first mode, all the modules in the system work under the control of the ASIP. In the second mode, they operate automatically with the stored parameters.

Figure 3.8 illustrates the pipelined scheduling of the DMAC$_0$ and DMAC$_1$ for IIT$_0$ and IIT$_1$. DMAC$_0$ is responsible for inputting data to the IITs in a staggering manner. DMAC$_1$, on the other hand, is to output data from the IITs, also in an alternative manner. As can be seen, from t_3 onward, one block is input, processed, and output in every eight cycles.

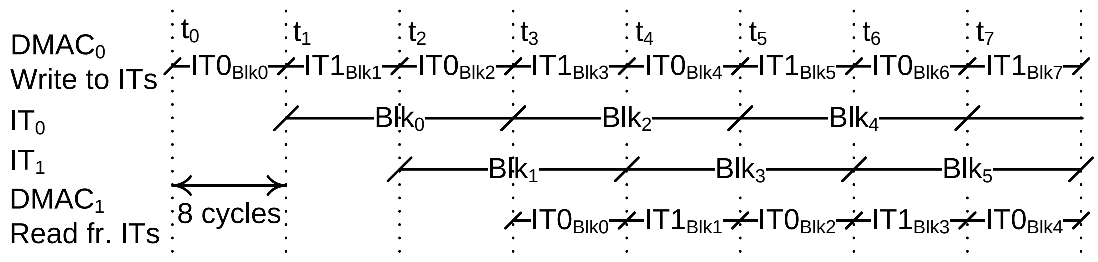


FIGURE 3.8: Pipelined scheduling with the support of two DMACs.

3.3.2.2 Proposed Inverse Integer Transform in the SoC

The proposed IIT module and its internal RAM are designed as slaves in the SoC, so that they can work independently. When the IIT operates, the DMACs can still transfer data to/from its RAM. As a result, the IITs may work concurrently because the data are always available.

The proposed IIT module can process two commands from ASIP: (1) setting the quantization parameter (QP); and (2) performing rescaling and the specified inverse transform. The following describes the implementation for the second task.

3.3.2.3 ASIP

The ASIP architecture is based on the RISC stored program machine (SPM) (Ciletti, 2003) with the following modifications allowing control of up to four IP blocks: (1) A Wishbone system bus is designed into the top level architecture allowing connections of the functional blocks to the microcontroller; (2) Four newly defined instructions: DIN, DOUT, IIT, QPS; (3) Five newly defined registers: Data, Code address, External address, Master Wishbone interface, and CYC_O register.

3.3.3 Experimental Results and Discussion

3.3.3.1 Simulation and Pre-layout Synthesis Results

The architecture with an ASIP, two DMACs, two embedded RAMs and Wishbone crossbar-switch bus system is designed and implemented in Verilog, and verified with RTL simulations using ModelSim. Simulation results show that the proposed system can compute all four types of IITs including rescaling operation. The pre-layout synthesis results on Xilinx Virtex IV platform, AMS $0.35\mu m$, and IBM $0.18\mu m$ CMOS technologies are shown in Table I.

3.3.3.2 Definition of Normalized Aggregate Throughput

We further define a normalized aggregate throughput concept to facilitate comparisons among reported designs with different I/O bus widths. Let T, W, T_N and W_N be aggregate throughput (in pixels per cycle), data bus width (in pixels, for both input and output), normalized aggregate throughput, and normalized data bus width, respectively. We have:

$$T_N = \frac{W_N}{W}T. \tag{3.1}$$

We select $W_N = 128$ pixels as it is the least common multiple for all designs in comparison.

3.3.3.3 Discussion

We have investigated many IIT designs. However, we only include the designs by Chao et al. (2007); Ngo et al. (2008); Pastuszak (2008) in Table 3.5 because they support 8×8 IIT. In these designs, only the designs by Chao et al. (2007); Ngo et al. (2008) support all types of IITs. Wang et al. (2003); Amer et al. (2005); Raja et al. (2005); Kordasiewicz and Shirani (2005); Chao et al. (2007) did not include I/OD in the throughput calculation. Hence, in Table 3.5, for aggregate throughput computation, W and I/OD of the design by Chao et al. (2007) are assumed based on its described operation.

The design by Chao et al. (2007) supports all types of IITs except rescaling. The shared architecture for IITs has 16-pixel-wide input and 8-pixel-wide output, leading to a total data bus width of twenty-four pixels. In the design, pixels are assumed available at the IIT. For each 8×8 block, it requires sixteen cycles to process and three cycles for pipelining, yielding a throughput of 3.4 ppc. However, if I/OD had been considered, its total input and output delay, total delay and aggregate throughput would have been twelve cycles, thirty-one cycles, and 2.1 ppc, respectively. In order to maximize the throughput, the same pipelining mechanism as the proposed one is presumably applied, resulting in the aggregate and normalized aggregate throughput of 3.4 and 18.0 ppc, respectively.

The design by Ngo et al. (2008) supports all IITs with rescaling. The design needs 8-pixel I/O bus, leading to total bus width of sixteen pixels. It requires nineteen cycles to perform an 8×8 IIT including I/O delay yielding an aggregate and normalized aggregate throughput of 3.4 ppc and 26.9 ppc, respectively.

The design by Pastuszak (2008) supports all IITs and rescaling, except 2×2 IHT. It requires a 64-pixel data bus to input or output an 8×8 block at a time. Therefore, the total data bus width is 128 pixels. By pipelining all the modules, and using thirty-two rescaling multipliers and eight units of 1-D hardware, it can

TABLE 3.5: Performance comparison of pre-layout synthesized designs.

Design	Type of transform	Re-scaling	Technology	Gate count [a] (gates)	Speed (MHz)	I/O Data bus width/ W (pels/pels/pels)	I/OD (cycles)	PD (cycles)	Total delay (cycles)	Throughput/ Aggregate throughput (ppc/ppc)	T_N [b] (ppc)	T_N (Mpps)
Pastuszak (2008)	4×4 (H),8×8	Yes	Stratix II	N.A.	100	64 / 64 / 128	2	2	2	32.0 / 32.0	32	3200
Ngo et al. (2008)	All [c]	Yes	Virtex IV	2.0K slcs	145	8/8/16	8	19	19	8.0 / 3.4	26.9	3900.5
Proposed	All [c]	Yes	Virtex IV	5.3K slcs	133	8/8/16	8	8	8	8.0 / 8.0	64	8512
Ngo et al. (2008)	All [c]	Yes	0.35 μm	21.5K	150	8/8/16	8	19	19	8.0 / 3.4	26.9	4042.1
Pastuszak (2008)	4×4 (H),8×8	Yes	0.35 μm	99K	82	64 / 64 / 128	2	2	2	32.0 / 32.0	32	2624
Proposed	All [c]	Yes	0.35 μm	46.6K	134	8/8/16	8	8	8	8.0 / 8.0	64	8576
Chao et al. (2007)	All [c]	No	0.18 μm	9.5K	125	16 [d] / 8 [d] / 24 [d]	4/8	16	19	8.0 / 3.4	18	2245.6
Pastuszak (2008)	4×4 (H),8×8	Yes	0.18 μm	141.2K	83	64 / 64 / 128	2	2	2	32.0 / 32.0	32	2656
Proposed	All [c]	Yes	0.18 μm	37K	144	8/8/16	8	8	8	8.0 / 8.0	64	9216

[a] Excluding buffers.
[b] Calculation is based on (3.1) where $W_N = 128$ pixels.
[c] All 4×4, 8×8 IIT, and 4×4, 2×2 inverse Hadamard transforms.
[d] Assumed value.

transform an 8×8 block in two cycles, yielding a high aggregate and normalized throughput of thirty-two ppc.

In the proposed design, using 16-pixel-wide data bus and pipelining the two IITs with I/O, I/OD and PD of eight cycles can be achieved, leading to an aggregate and normalized aggregate throughputs of eight ppc and sixty-four ppc, respectively.

Efficient VLSI designs not only yield high pixel-per-cycle throughput, but also high pixel-per-second throughput. This is done by shortening the critical path length, and thus maximizing circuit operating frequency. Based on the normalized ppc-throughput and speed of the system, the normalized pps-throughput can be computed as:

$$T_{pps} = T_{ppc} f. \tag{3.2}$$

Compared to the reported designs, the proposed design achieves a higher normalized aggregate throughputs in both ppc and pps. Compared to the design by Chao et al. (2007) using 0.18 μm technology, our design has a 3.6-fold normalized aggregate ppc-throughput (due to a 1.5 times smaller in data bus width and 2.4 times smaller in total delay), and a 4.1-time normalized pps-throughput (due to the 1.2 times higher in operating frequency). Using the same Virtex IV platform or 0.35 μm technology with Ngo et al. (2008)'s design, the normalized aggregate throughputs in ppc and pps of the proposed design are 2.4 and 2.1 times as large as those of Ngo et al. (2008)'s design, respectively, thanks to 58% less in total delay. Compared to Pastuszak (2008)'s design, our design has a two times higher in normalized aggregate ppc-throughput in both 0.35 and 0.18 μm technologies since its bus width is eight times smaller while its throughput is only four times smaller. The normalized pps-throughput depends on the operating frequency, thus, that of our design is 3.3 and 3.5 times as large as that of Pastuszak (2008)'s design in 0.35 and 0.18 μm technologies, respectively.

The proposed system can perform IIT and inverse quantization for videos with the resolutions of up to 38.4 Mpels and progressive scan frequency of 30 Hz (30 frames/sec). This supported resolution is higher than 7680 $\times$ 4800 pels.

3.4 Summary

In this chapter, we address the portability issue for integer transform hardware designs in particular and for hardware designs in general. In order to achieve portability, the designs should conform to a bus standard so that they can be easily integrated with other designs deploying the same bus standard. The Wishbone bus standard has been selected to be deployed in our designs because it out-performs other standards in terms of open property, simplicity, flexibility and portability.

Two portable inverse integer transform architecture designs are proposed in this chapter. In the first design, the most functionally complete inverse integer transform unit is proposed. The unit has the embodiment of 4×4 circuits in the 8×8 circuit and the embodiment of 2×2 circuit in the 4×4 circuits. Then, another inverse integer transform (IIT) block is also proposed based on the above shared unit and an additional quantization module. The inverse integer transform design is implemented on an ASIP-controlled SoC platform conforming to the Wishbone shared bus standard for portability and testability. The design can perform a 2×2, 4×4, and 8×8 data block in one, eleven and nineteen cycles, respectively. The design can transform videos with the resolutions of up to 16.4 Mpels and progressive scan frequency of 30 Hz (30 frames per sec). The resulting circuit area is considerably smaller compared to the designs in its class, thanks to the proposed transform unit.

The second proposed design deploys a fully pipelining mechanism supported by an ASIP, two DMACs and two proposed IIT blocks. As a result, it can transform each 2×2, 4×4, and 8×8 data block in one, four, and eight cycles, respectively, compared to one, eleven and nineteen cycles in the first portable design, respectively. The second architecture can deliver a high normalized throughput of sixty-four ppc and 15.6 Gpps at 144 MHz using 0.18 μm technology. It supports the transforms for videos with the resolutions of up to 38.4 Mpels and progressive scan frequency of 30 Hz (30 frames/sec). The resulting circuit area is more reasonable compared to the designs in its class owing to the proposed inverse integer transform unit.

Chapter 4

Very High-Throughput Forward/Inverse Transform Architectures for H.264/AVC

4.1 Introduction

In H.264/AVC, there are four transform types in total, including the 4×4 and 8×8 integer transforms, and the 2×2 and 4×4 Hadamard transforms for DC coefficients. For high-throughput and area-efficient designs, two design levels should be taken into account: the *algorithm* and *architecture* levels. At the *algorithm* level, the H.264 developers have proposed fast and low-cost algorithms for all the forward and inverse transforms. Similar to the DCT, the fast forward/inverse algorithms decompose a 2-D transform into a 1-D row/column transform, followed by a 1-D column/row transform. Most of the reported integer transform designs conform to these fast algorithms.

At the *architecture* level, two techniques to achieve high throughput designs were reported in the literature. The first technique (Amer et al., 2005; Kordasiewicz and Shirani, 2005; Raja et al., 2005) is to input all data of a block, process and output all transformed (and quantized) data in parallel, leading to unpractical wide I/O buses. The second technique (Amer et al., 2005; Kordasiewicz and Shirani, 2005; Raja et al., 2005; Pastuszak, 2008) is to deploy pipelining mechanisms among operating processes. The pipelining mechanisms can be for input, output and transform processes as in the transform designs by Chen et al. (2006); Shi et al. (2007) for the H.264/AVC main profile (4×4 in size); or they can be for transform and quantization processes as in the designs by Pastuszak (2008) with the transform size of 8×8 for the H.264/AVC high profiles. The pipelining mechanisms can be for input, output and transform processes as in the transform designs by Chen et al. (2006); Shi et al. (2007) for the H.264/AVC main profile (4×4 in size); or they can be for transform and quantization processes in the designs by Pastuszak (2008) with the transform size of 8×8 for the H.264/AVC high profiles. The pipelining mechanisms actually reduce the total delays because the modules in the architectures work in parallel. Thus, they increase system throughput and performance. Although transform and quantization processers are pipelined, the I/O delay is not counted in the reported 8×8 transform designs. In addition, the reported designs supporting these pipelining mechanisms have very large bus widths (of thousands of bits). This leads to unpractically high wiring areas and difficulties in implementing hard IP cores. Therefore, it is strongly desired to develop integer transform designs which achieve high throughputs despite a reasonable bus width.

Different hardware-sharing techniques have been reported to achieve high area efficiency. Hardware-sharing can be performed among (1) the coefficient computations (Kordasiewicz and Shirani, 2005); (2) all the forward/inverse transforms

(Chao et al., 2007; Pastuszak, 2008; Su and Fan, 2008); or (3) the forward and inverse transforms (Shi et al., 2007; Choi et al., 2008). Technique (1) leads to a very low throughput as only one coefficient is outputted at one time. Technique (3) is not practical as hardware-sharing is only efficient when one shared component works at one time. Here, in encoders, the forward and inverse transforms should be performed in parallel while in decoders, only the inverse transforms are needed. The reported architectures using technique (2) have not included the Hadamard transforms in the shared hardware units yet.

Many Forward IT (FIT) and Inverse IT (IIT) designs have been reported in the literature. Among the FIT designs, those by Amer et al. (2005); Pastuszak (2008); Park and Ogunfunmi (2009); Hu et al. (2009) support 8×8 FIT. However, none of them supports all four types of FITs. Besides, the designs by Pastuszak (2008); Park and Ogunfunmi (2009); Hu et al. (2009) support quantization. Among the IIT designs supporting 8×8 IIT by Pastuszak (2008); Ngo et al. (2008); Chao et al. (2007); Su and Fan (2008), the designs by Ngo et al. (2008); Chao et al. (2007); Su and Fan (2008) support all four types of ITTs. Only the 8×8 IIT designs by Pastuszak (2008); Ngo et al. (2008); Chao et al. (2007); Su and Fan (2008) have rescaling function. Among all these designs, no design supports all FITs and quantization, while the design by (Ngo et al., 2008) supports all IITs and rescaling functions.

In this chapter, we propose high-throughput and area-efficient System-on-Chip-based (SoC) forward integer transform (FIT) and inverse integer transform (IIT) modules for H.264/AVC. Continuing the idea that large-size transform operations should fully reuse small-size ones (including the 4×4 and 2×2 Hadamard transforms) for area efficiency, a forward transform unit is proposed. The proposed forward and inverse integer transform (FIT/IIT) architectures employ a new forward transform unit and reuse the proposed inverse transform unit from the first portable design (Section 3.2.1). FIT/IIT modules can perform both the 4×4 and 8×8 forward/inverse transforms, and the 2×2 and 4×4 forward/inverse Hadamard transforms of DC coefficients with a high hardware reuse possibility.

While in the first portable IIT design (Section 3.2), nineteen cycles are required to complete an 8×8 inverse transform, in the second portable IIT design (Section 3.3), only eight cycles are needed thanks to a pipelining mechanism and the execution of rescaling together with the 1-D transform for a column/row in the same cycle. However, this leads to a reduction of the system frequency (speed). In this design, in order to achieve an even-higher throughput, quantization/rescaling are scheduled to execute in a separate pipeline stage with the forward/inverse transform operations. This results in a system speed increment while maintaining the same number of transformation cycles.

Based on the observation that the transform units only read input data in half of the processing time, a novel series of input and output buffers is proposed to balance the data-transfer load in the I/O bus. This reduces the bus width by half for the same hardware architecture. Thanks to the novel I/O buffer designs, two transform units are integrated into the integer transform architectures, which double the processing speed and utilize the bus width. Owing to the fully pipelining mechanism among input, output, transform and quantization processes, the architectures can out-perform other reported designs in terms of throughput and area efficiency.

The remaining of the chapter is organized as follows. The design of the FIT/ITT module and its system architecture are presented in Section 4.2. Circuit simulation, pre-layout synthesis results and comparisons are discussed in Section 4.3, followed by a summary in Section 4.4.

4.2 Proposed Forward and Inverse Architectures

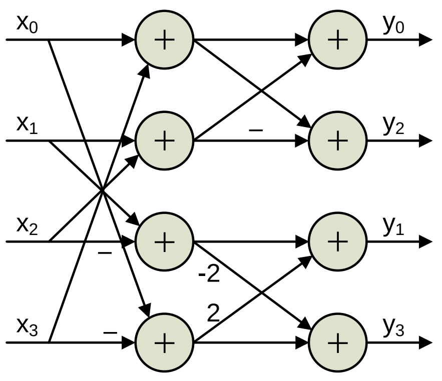
4.2.1 Proposed Forward Transform Unit

Since the inverse transform unit has already been proposed in Section 3.2.1, this chapter introduces a new shared forward transform unit for area efficiency and high throughput forward transform designs.

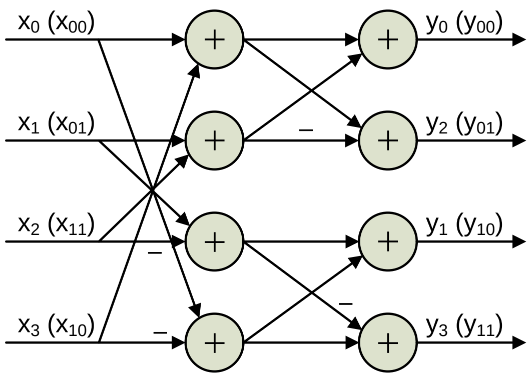
The fast and low-cost 1-D 8×8 forward algorithm (Gordon et al., 2004) is used for our 2-D 8×8 forward computation. The 1-D transform algorithms are used to transform a row/column. The 2-D transform algorithms can be done by repeatedly applying the 1-D transforms for all rows/columns of the block, then for all columns/rows.

As can be seen from Figure 4.1, the 1-D 8×8 forward integer transform (FIT) algorithm (Figure 4.1(c)) includes three sub-blocks: F1, F2 and F3. F1 and F2 are equivalent, and are used for the 1-D 4×4 FIT (Figure 4.1(a)), the 1-D 4×4 forward Hadamard transform (FHT) and the 2-D 2×2 FHT (Figure 4.1(b)) implementation. The 2-D 2×2 FHT is implemented using 1-D 4×4 FIT with changes of output order.

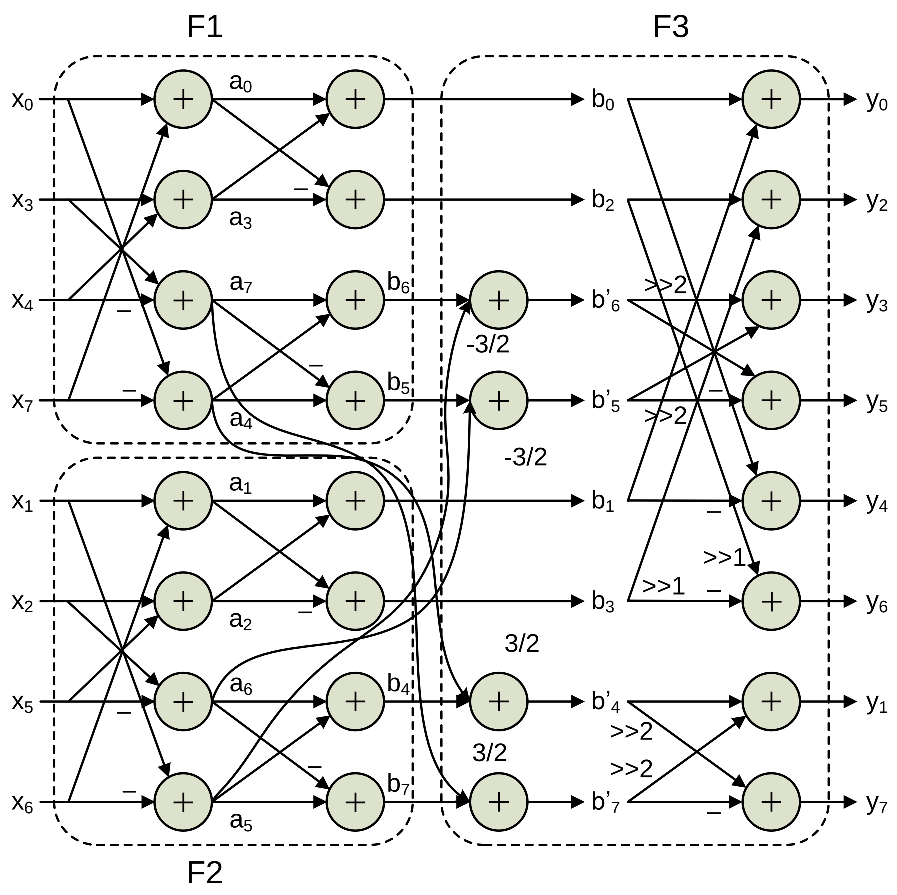
Based on these observations, a FIT unit (Figure 4.2), the so-called FIT computing kernel (CK), is designed to 1-D transform an 8-value row/column, or to 1-D



(a)



(b)



(c)

FIGURE 4.1: 1-D 8 × 8 FIT/IIT algorithms. (a) 1-D 4 × 4 FIT; (b) 1-D 4 × 4 FHT (2-D 2 × 2 FHT); and (c) 1-D 8 × 8 FIT.

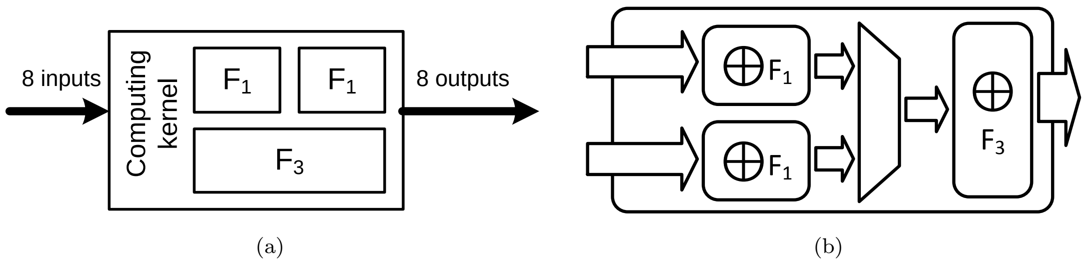


FIGURE 4.2: The proposed shared forward integer transform hardware unit. (a) Block diagram; and (b) Dataflow diagram.

transform two 4-value rows/columns, or to 2-D transform two 2×2 blocks simultaneously. The FIT CK has one F1, one F2 and one F3 sub-blocks. Using a single 1-D 8×8 transform architecture to perform all four transforms significantly reduces the system area.

4.2.2 Proposed FIT/IIT Blocks

The proposed FIT and IIT blocks share the same architecture as shown in Figure 4.3, where two CKs, either FITs or IITs are used. The 2-D FIT/IIT is executed in two major stages: (1) 1-D transforms of rows/columns in the CK; and (2) 1-D transforms of columns/rows previously stored in the internal RAM by looping back to the CK and rounding off the results. In the first stage, two 8-value columns/rows (of an 8×8 block) or a whole 4×4 block are passed through a multiplexer to the CK for 1-D transform. The results are bypassed the rounder and stored into the internal RAM. This stage is done in one cycle and repeated until all data blocks are processed. In the second stage, two 8-value columns/rows (of an 8×8 block) or a whole 4×4 block are also fed from the internal RAM back to the CK for 1-D transform followed by a rounding. Similarly, this stage is finished in one cycle, and repeated until all data in the block are processed. In total, the

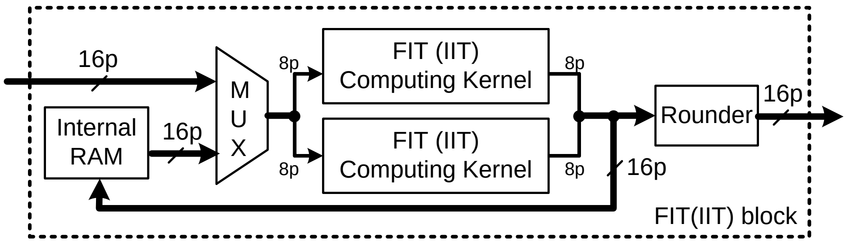


FIGURE 4.3: The proposed FIT/IIT block architectures.

architecture requires one, two, eight cycles to rescale a 2×2, 4×4, 8×8 block, respectively.

4.2.3 Proposed FIT/IIT Module

The proposed FIT/IIT modules are designed to perform FIT/IIT with quantization/rescaling functions. The proposed IIT block, for 8×8 transform, requires 16-pixel input per cycle in the first four cycles, while requiring 0-pixel input per cycle in the last four cycles out of total eight cycles. As a result, the rescaling block needs a sufficient resource to multiply sixteen pixels simultaneously in the first half, while it will idle in the second half of the total processing time. Not only the IIT block, but the FIT as well as other designs also have similar issue. Moreover, quantization/ rescaling with multipliers costs a significant gate count.

The proposed FIT/IIT module can solve the problem by using special buffers. The module consists of a FIT/IIT block, two input buffers IBs, two output buffers OBs and a quantization/rescaling block (Figure 4.4).

The IBs operate in a flip-flop manner to input eight pixels every cycle and output sixteen pixels per cycle in several consecutive cycles. In 8×8 transformation, they require eight consecutive cycles to completely input sixty-four pixels. At the output, they send out sixteen pixels per cycle in four cycles, and stay idling for

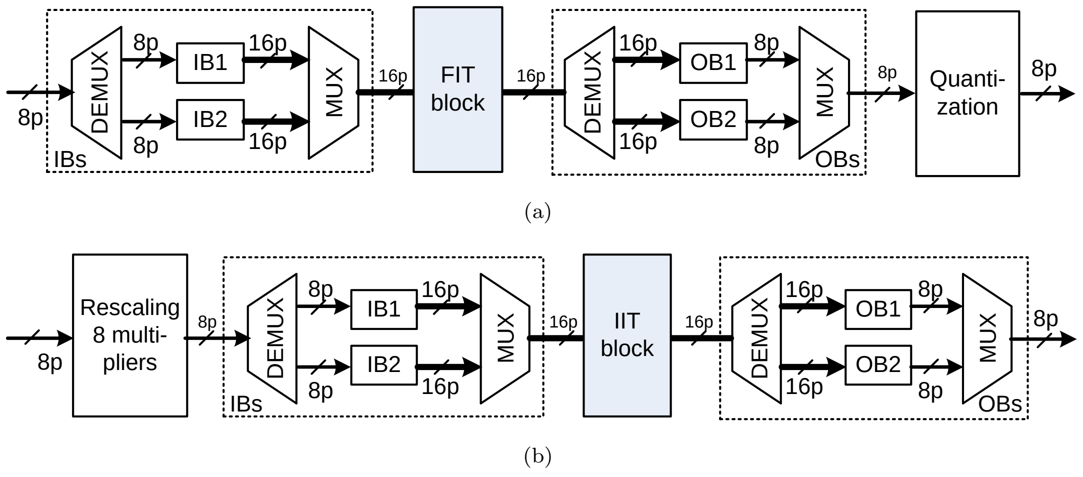


FIGURE 4.4: The proposed (a) FIT and (b) IIT modules.

four other cycles. On the other hand, the OBs operate similarly to the IBs but in a reverse scheme.

With the support of IBs and OBs, quantization/rescaling allows multiplication of eight pixels concurrently, avoiding the operations on sixteen pixels. Thus, the proposed module can reduce half of quantization resource, with a small increase in gate count for the buffers.

By pipelining FIT/IIT, quantization/rescaling blocks and buffers, the module still can process an 8×8 block in eight cycles. Figure 4.5 shows an example of 8×8 blocks IIT. As can be seen, from t_3 onward, one block is input, processed, and output in every eight cycles.

The proposed FIT/IIT modules can perform all four types of transform and quantization. They can process a 2×2 block in one cycle, a 4×4 block in two cycles, and an 8×8 in eight cycles.

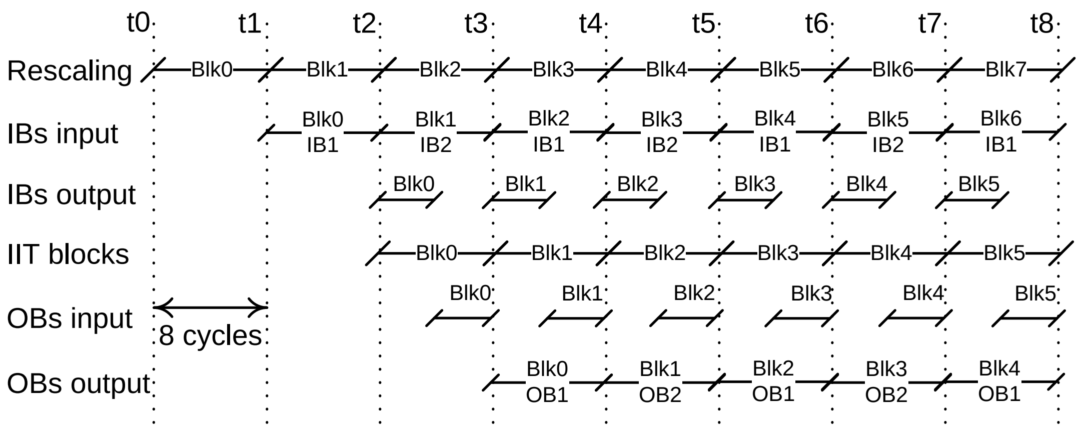


FIGURE 4.5: Pipelined scheduling for 8×8 residual blocks in the proposed IIT module.

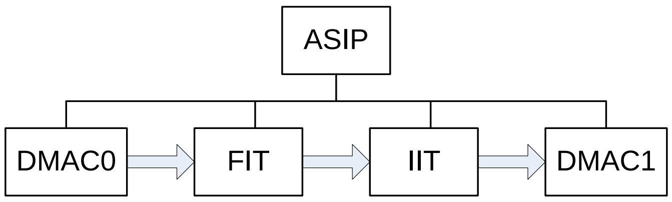


FIGURE 4.6: The proposed top-level system architecture with an ASIP support.

4.2.4 Proposed System Architecture

The proposed top-level system architecture, shown in Figure 4.6, includes an ASIP, two built-in-RAM DMACs, and FIT and IIT modules. DMAC0 is responsible for input data to FIT, while DMAC1 is to output data from IIT.

4.3 Experimental Results and Discussion

4.3.1 Simulation and Pre-Layout Synthesis Results

The FIT and IIT modules with the SoC architecture are RTL-designed and implemented in Verilog, simulated using Mentor Graphics ModelSim, and pre-layout synthesized on a Xilinx Virtex IV environment. HDL simulation results are verified with those generated by the JM 14.2 reference software (Suhring, 2008). The proposed system correctly supports all the four types of FITs/IITs including quantization/rescaling. Pre-layout synthesis results on Virtex IV and IBM 0.18 μm CMOS 7SF technology are shown in Tables 4.1 and 4.2, respectively. The inverse and forward transform architecture can support for videos with the resolution of up to 61.6 and 43.2 Mpels, respectively, with progressive scan frequency of 30 Hz. In another words, the proposed FIT/IIT can forward/inverse transform and quantize/rescale in real-time for Ultra High Definition Videos (UHDVs) with the resolutions of up to 7680×4320 pels and progressive scan frequency of 30 Hz in 0.18 μm technology.

4.3.2 Discussion

We reviewed many FIT/IIT designs. However, we only include the designs by Amer et al. (2005); Pastuszak (2008); Park and Ogunfunmi (2009); Hu et al. (2009); Ngo et al. (2008); Chao et al. (2007); Su and Fan (2008) in Tables 4.1 and 4.2 because they support 8×8 FIT/IIT, the most complex transform of H.264. Designs supporting sizes of up-to-4×4 are not included. Throughput T in pixel-per-cycle (ppc) is computed as the number of pixels processed in one cycle. In the tables, it is computed for 8×8 block processing. Throughput T in pixel-per-second (pps) is calculated as the product of T (ppc) and the operating frequency f. DTUA is defined as the T (pps) per unit of area. In ASIC implementation, the area can be reported in terms of the number of gates, whereas in FPGA, the unit area can be chosen as Virtex slices as most of the implementations are based on the Virtex platforms. The reported area in LUT unit will be converted to slices, since each slice has two LUTs (Xilinx, 2010). The area of designs implemented in Stratix can not be converted to Virtex slices. Since some designs do not support quantization or rescaling, we also report the area without quantization or rescaling to facilitate DTUA comparison.

In Table 4.1, DTUA of the proposed FIT is two and nine times as large as that of Amer et al. (2005) and Park and Ogunfunmi (2009)'s designs, respectively. The designs by Amer et al. (2005) and Park and Ogunfunmi (2009) support only 8×8 FIT, while the proposed design supports all the transforms. Using the same Virtex 4 platform, the proposed IIT design achieves a DTUA which is twice of that of Ngo et al. (2008)'s design. This better DTUA is mostly due to the significant decrease in the total delay, resulting in a large improvement in throughput.

Using 0.18 μm technology (Table 4.2), DTUA of the proposed FIT/FIT design is 1.4/2.1 times as large as that of the FIT/IIT design by Pastuszak (2008), respectively. It should be noted that the designs by Pastuszak (2008) do not support the 2×2 Hadamard transforms. Compared to the IIT designs by Chao et al. (2007) and Su and Fan (2008) without rescaling function, the proposed design achieves a 1.2 and 2.5-time DTUA, respectively.

4.4 Summary

We have reviewed a number of FIT/IIT designs and proposed a high throughput and area-efficient SoC-based FIT/IIT design. With the newly proposed forward transform unit and the proposed inverse transform unit from the first portable design (Section 3.2.1), our proposed design can perform all the 4×4 and 8×8 transforms with the additional supports for the 2×2 and 4×4 Hadamard

Table 4.1: Comparisons of FPGA FIT/IIT implementations.

Design	FIT / IIT	Types	Q	FPGA	Utilization	f (MHz)	Delay (cycles)	T (ppc)	T (Mpps)	Slices (K)	DTUA (Kp/s.slice)
Amer et al. (2005)	FIT	8×8	No	Virtex 2	29K LUT	68.0	1	64.0	4352.0	14.5	300.1
Pastuszak (2008)	FIT	4, 8 [a]	Yes	Stratic 2	7.3KALTs + 32DSP	100.0	2	32.0	3200.0	–	–
Park and Ogunfunmi (2009)	FIT	8×8	Yes	Virtex 2 Pro.	1.6K slices	111.8	64	1.0	111.8	1.6	69.9
Hu et al. (2009)	FIT	4, 8 [a]	Yes	Virtex 4	–	108.0	17	3.8	406.6	–	–
Proposed	FIT	All	Yes	Virtex 4	2.1K slices	167.1	8	8.0	1336.8	2.1	636.6
Ngo et al. (2008)	IIT	All	Yes	Virtex 4	2K slices	145.0	19	3.4	488.4	2	244.2
Pastuszak (2008)	IIT	4, 8 [a]	Yes	Stratic 2	7.3KALTs + 32DSP	100.0	2	32.0	3200.0	–	–
Proposed	IIT	All	Yes	Virtex 4	2.1K slices	133.5	8	8.0	1068.0	2.1	508.6

[a] 4×4, 8×8 including Hadamard transform.

Table 4.2: Comparisons of 0.18μm ASIC FIT/IIT implementations.

Design	FIT/IIT	Types	Q	Gate count (K)	f (MHz)	Delay (cycles)	T (ppc)	T (Mpps)	DTUA (Kp/s.gate)
Pastuszak (2008)	FIT	4, 8 [a]	Yes	162.1	77.0	2	32.0	2464.0	15.2
Proposed	FIT	All	Yes	62.2	162.1	8	8.0	1296.8	20.8
Chao et al. (2007)	IIT	All	No	9.5	125.0	19	3.4	425.0	44.7
Pastuszak (2008)	IIT	4, 8 [a]	Yes	141.3	83.0	2	32.0	2656.0	18.8
Su and Fan (2008)	IIT	All	No	18.1	100.0	16	4.0	400.0	22.1
Proposed	IIT	All	Yes	47.3	230.9	8	8.0	1847.2	39.1
Proposed	IIT	All	No	33.3	230.9	8	8.0	1847.2	55.5

[a] 4×4, 8×8 including Hadamard transform.

transforms of the DC coefficients. With the proposed pipelining mechanism among input, output, transform and quantization processes, and the special input and output buffer designs to balance the data load in data input and output buses, our FIT/IIT design achieves a higher DTUA compared to the reported designs. In the 0.18 μm technology, the FIT and IIT modules can be operated at the maximum speeds of 162.1 and 230.9 MHz, respectively. The inverse and forward transform architecture can entertain videos with the resolution of up to 61.6 and 43.2 Mpels, respectively, at the progressive scan frequency of 30 Hz.

Chapter 5

Performance-Cost Analyses for H.264 Forward/Inverse Integer Transforms

5.1 Introduction

Higher-fidelity videos for application areas like professional film production, video post production, or high-definition TV/DVD are the main target of the H.264/AVC high profiles (Marpe et al., 2005). Therefore, the primary objective of all the existing forward/inverse integer transform (FIT/IIT) designs is to achieve high performance, whereas the secondary objective is to reduce design costs. High performance is often in terms of high throughput with preferably high operating frequency (Wang et al., 2003; Kordasiewicz and Shirani, 2005; Chen et al., 2005, 2006; Hwangbo et al., 2007; Pastuszak, 2008; Choi et al., 2008; Ngo et al., 2008; Su and Fan, 2008; Do and Le, 2009, 2010). Low cost is seen as having low power (Choi et al., 2008) or low gate count (Wang et al., 2003; Kordasiewicz and Shirani, 2005; Chen et al., 2005, 2006; Hwangbo et al., 2007; Shi et al., 2007; Pastuszak, 2008; Do and Le, 2010). In the literature, throughput is either computed as the maximum amount of data input at a time (Kordasiewicz and Shirani, 2005; Chao et al., 2007; Shi et al., 2007; Ngo et al., 2008) or as the average amount of data processed in one time unit (Chen et al., 2005, 2006; Hwangbo et al., 2007; Pastuszak, 2008; Choi et al., 2008; Su and Fan, 2008; Do and Le, 2009, 2010). Typically, the former produces higher throughputs, while the latter is commonly used in practice. It should be noted that the designs by Ngo et al. (2008); Do and Le (2009, 2010) are the proposed designs in Section 3.2, Section 3.3 and Section 4, respectively.

Data throughput rate per unit area (*DTUA*) was first reported in the literature for a Discrete Fourier Transform design by Bliss and Julien (1990) for high throughput cost efficiency. *DTUA* is used to compare FIT/IIT designs in Pastuszak (2008); Do and Le (2010); Shi et al. (2007); Chen et al. (2005, 2006); Hwangbo et al. (2007). Other than *DTUA*, comparisons among designs in terms of high throughput and cost efficiency using only throughput, gate count, or power have been arbitrary. As a result, *DTUA* has been used as the metric to evaluate high throughput and cost efficiency in FIT/IIT designs. However, other than throughput and circuit area involved in *DTUA*, interconnection, power and delay are not considered. This leads to the fact that some designs use very wide buses (Pastuszak, 2008; Chen et al., 2006), which causes large wiring areas, but their authors were still able to claim their area efficiencies. Therefore, evaluation of high throughput and cost-efficient designs using only throughput and circuit area is incomprehensive. Thus, a metric which comprises most if not all performance and cost parameters is needed.

In this chapter, first, we conduct performance-cost analyses for the H.264 forward/inverse integer transforms, and second, we propose a *performance-cost metric* (*PCM*) for H.264 forward/inverse integer transform (FIT/IIT) designs. The proposed metric is defined as the ratio of data throughput over the design cost, which includes power, area, and delay; and issues associated with interconnections in sub-micron design. Compared to *DTUA*, *PCM* facilitates more comprehensive comparisons for VLSI FIT/IIT designs. Third, we develop a performance-cost analysis software to facilitate the use of the proposed *PCM* technique. When using this software, users are asked to enter some preliminary parameters of their designs. Based on the given parameters and the reference designs, it then analyzes and provides the possible boundaries of the users' designs in order to have better *PCM*s compared to the reference designs. In addition, it can also export comparison results among different designs. The software is flexibly designed to facilitate the use of not only our *PCM* technique in FIT/IIT designs, but also other different metrics in other architectures.

The rest of the chapter is organized as follows. In Section 5.2, FIT/IIT design costs are analyzed, followed by performance-cost metric definition and description in Section 5.2. In Section 5.4, a detailed discussion of *PCM* in comparison with *DTUA* through different designs in the literature is presented. PCAS is introduced in Section 5.5. The chapter ends with conclusions in Section 5.6.

5.2 Cost Analyses for FIT/IIT/Designs

5.2.1 Estimation of Power Consumption

5.2.1.1 Background

In CMOS technology, power consumption comprises of static and dynamic components. Static power consumption is caused by leakage currents, and usually neglected for systems with continuous operation in submicron technologies (Nebel and Mermet, 1997). On the other hand, dynamic power contributes marginally by short circuit power (typically below 20% (Nebel and Mermet, 1997)), and significantly by switching of load capacitance (Rabaey et al., 2003) as follows:

$$P_{dyn} = \alpha C_L V_{DD}^2 f, \tag{5.1}$$

where α, C_L, V_{DD} and f represent switching activity, load capacitance, supply voltage, and operating frequency, respectively. Dynamic power is directly proportional to the load capacitance C_L comprising: (1) the intrinsic MOS transistor capacitances; (2) extrinsic MOS transistor fan-out capacitances, and (3) interconnect capacitances (Rabaey et al., 2003). The extrinsic capacitances can be approximated by the gate capacitances of the succeeding gates (Rabaey et al., 2003), hence it can be used as the total gate capacitances of the IC. While it is shown that gate and interconnect capacitances contribute significantly to the total load capacitance (Savidis and Friedman, 2008), interconnect capacitance is growing in importance with the scaling of technology (Rabaey et al., 2003). Analysis on Intel microprocessors has shown that over 50% of the dynamic power dissipates at the interconnects (Savidis and Friedman, 2008). Therefore, total load capacitance can be approximated as the sum of gate and interconnect capacitances as expressed as:

$$C_L = C_G + C_{interconnect}. \tag{5.2}$$

5.2.1.2 Gate Capacitance

The total gate capacitance is proportional to the gate count of the design. Using the 2-input NAND gate (NAND2) in the standard cell library of a particular technology as a unit, C_G can be defined as:

$$C_G = \beta G, \tag{5.3}$$

where β and G are the gate capacitance and gate count of an unit NAND2. According to Rabaey et al. (2003),

$$\beta = C_{G\_NMOS} + C_{G\_PMOS} = C_{ox} L_{channel} (W_{NMOS} + W_{PMOS}), \tag{5.4}$$

where C_{ox}, $L_{channel}$, W_{NMOS} and W_{PMOS} are gate oxide capacitance per unit area, channel length, NMOS channel width and PMOS channel width inside the unit gate, respectively.

5.2.1.3 Interconnect Capacitance

Interconnect capacitance, on the other hand, includes three components (Figure 5.1(a)): (1) parallel-plate capacitance (Figure 5.1(b)); (2) fringe capacitance between the side walls of the wires and the substrate; and (3) inter-wire capacitance (Rabaey et al., 2003), as follows:

$$C_{interconnect} = C_{pp} + C_{fringe} + C_{interwire}. \tag{5.5}$$

As shown in Figure 5.1(c), $C_{interwire}$ dominates the total interconnect capacitance under submicron technology. $C_{interwire}$ between two conservative wires is expressed (Rabaey et al., 2003) as follows:

$$C_{interwire} = \frac{\epsilon_{di}}{t_{di}} HL, \tag{5.6}$$

where ϵ_{di} and t_{di} are permittivity and thickness of the dielectric layer, and H and L are length and thickness of the interconnect. Assume that a layer contains N parallel wires, its interwire capacitance is

$$C_{interwire\_N} = (N-1)\frac{\epsilon_{di}}{t_{di}}HL. \tag{5.7}$$

When N is large,

$$C_{interwire\_N} \approx N\frac{\epsilon_{di}}{t_{di}}HL. \tag{5.8}$$

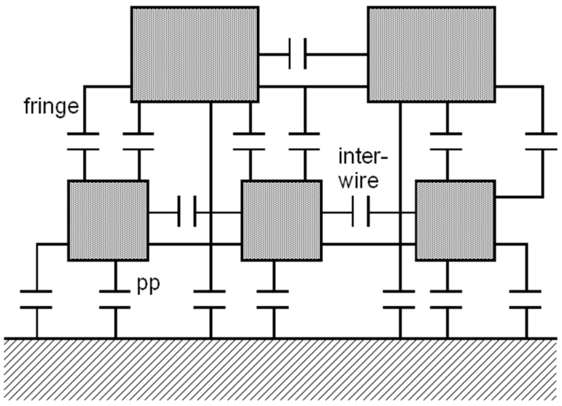
5.2.1.4 Inter-wire Capacitance

An IC has several layers of wires generally classified as local and global. In order to estimate the inter-wire capacitance using (Equation (5.8)), the number of wires N for each layer is estimated. Rent's rule (Landman and Russo, 1971) has been widely used to estimate the number of terminals for modules. For a module containing B blocks, each block has an average of K pins (terminals), and P, the average number of pins per module, is

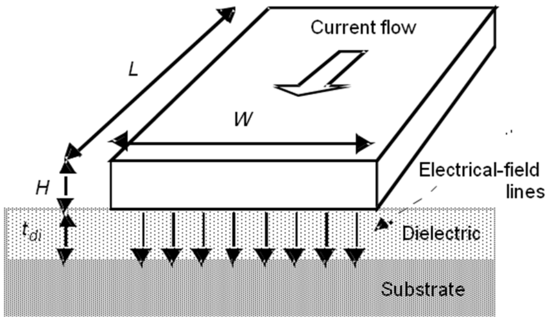
$$P = KB^r, \tag{5.9}$$

where r is a constant depending on block type.

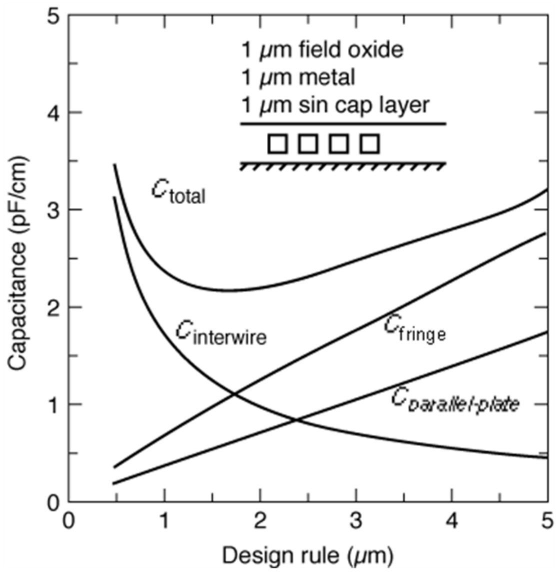
Based on Rent's rule, the number of nets (or wires) in a given module can be computed as follows. Since each block has K pins, there are KB pins in the module. Meanwhile, P pins are used as module terminals. Therefore, $(KB - P)$ pins are reserved for nets in the module. From definition, a net connects two pins



(a)



(b)



(c)

FIGURE 5.1: (a) Three types of interconnect capacitances; (b) Parallel-plate capacitance model; (c) Dominance of inter-wire capacitance with design rule.

from two blocks, thus the number of nets including terminals of the module is

$$N = \frac{KB - P}{2} + P = \frac{KB + P}{2}, \tag{5.10}$$

$$N = \frac{KB + KB^r}{2} = K(B + B^r)/2. \tag{5.11}$$

Now we apply (Equation (5.11)) to find the number of nets at each layer for an IC. Local wire layers are used to connect gates, hence gates now can be considered as blocks in (Equation (5.11)). Assume that the IC contains G NAND2 gates. When blocks are gate array, $r = 0.5$ (Landman and Russo, 1971), the total number of local wires connecting G gates is

$$N_L C = 3(G + G^{0.5})/2 = 1.5(G + \sqrt{G}). \tag{5.12}$$

Global wire layers are used to connect IP blocks in a SoC, hence IP blocks now can be considered as blocks in (Equation (5.11)). Assume that each IP block has K pins (or terminals or input/output), the total number of wires in global wire layers is

$$N_{GB} = K(B + B^r)/2. \tag{5.13}$$

Also, assume that all global wires have the same height and length, and all local wires have the same height and length, inter-wire capacitance is

$$C_{interwire} = N_{LC}\frac{\epsilon_{di}}{t_{di}}H_{LC}LLC + N_{GB}\frac{\epsilon_{di}}{t_{di}}H_{GB}LGB, \tag{5.14}$$

$$C_{interwire} = 1.5\frac{\epsilon_{di}}{t_{di}}H_{LC}LLC(G + \sqrt{G}) + \frac{1}{2}(B + B^r)\frac{\epsilon_{di}}{t_{di}}H_{GB}LGBK. \tag{5.15}$$

Let us define

$$\gamma = 1.5\frac{\epsilon_{di}}{t_{di}}H_{LC}LLC, \tag{5.16}$$

$$\theta = \frac{1}{2}(B + B^r)\frac{\epsilon_{di}}{t_{di}}H_{GB}LGB, \tag{5.17}$$

then (Equation (5.15)) becomes

$$C_{interwire} = \gamma(G + \sqrt{G}) + \theta K. \tag{5.18}$$

5.2.1.5 Formula for Power Consumption

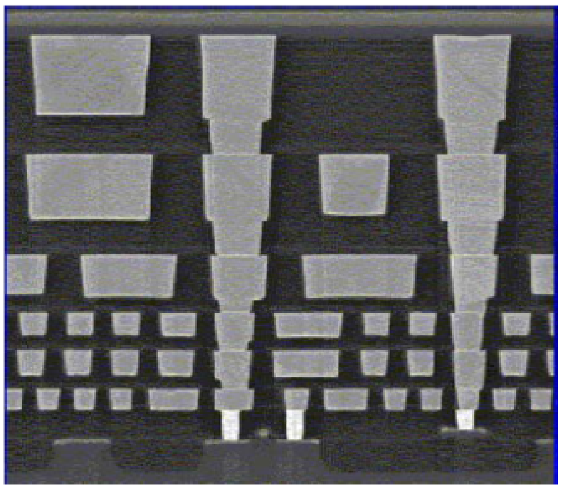
Based on Equations (5.1)-(5.3) and (5.18), the total power consumption of an IC can be estimated as follow:

$$P = \alpha[\beta G + \gamma(G + \sqrt{G}) + \theta K]V_{DD}^2 f. \tag{5.19}$$

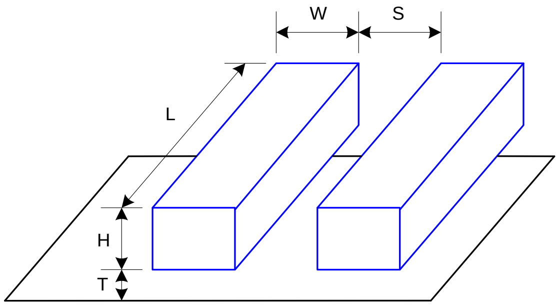
5.2.2 Estimation of Circuit Area

Transistors of IC chips are built on top of silicon substrate. Each transistor must be powered and wired to construct logic gates, circuit blocks, functional units and higher-level functional structures. The interconnect layers are laid over the transistors (Figure 5.2). Interconnect layers vary enormously in wire sizes in width W and thickness H, and are classified into local (bottom) interconnect layers and global (top) layers, one stacked on top of another. Therefore, the IC area can be estimated as the maximum among the followings: (1) area of total number of logic gates, (2) area of local interconnect layers, and (3) area of global interconnect layers:

$$A = max\{A_G, A_{LC}, A_{GB}\}. \tag{5.20}$$



(a)



(b)

FIGURE 5.2: (a) 0.13μm cross-section, source-Intel (IBM, 2008); (b) Interconnect geometry.

However, in today's submicron era, transistors are becoming relatively small, and chips are mostly made of wires (Weste and Harris, 2004). As stated in Celik et al. (2002), interconnects have dominant impact on IC area. The local wires (closest to the transistors) are very small in size, while the global wires are significantly larger in the thickness and width compared to the local ones. Since the number of wires can be estimated using Equations (5.12-5.13), assuming they needs number of layers, η_{LC} and η_{GB}, to build all local and global wires, respectively. Area of local and global layers can be estimated as:

$$A_{LC} \approx 1.5(G + \sqrt{G})(W_{LC} + S_{LC})L_{LC}/\eta_{LC}, \tag{5.21}$$

$$A_{GB} \approx \frac{1}{2}W(B + B^r)(W_{GB} + S_{GB})L_{GB}/\eta_{GB}, \tag{5.22}$$

where W, S, L are the width, spacing, and length of wires (as shown in Figure 5.2(b)), respectively. From Equations (5.21 and 5.22), we define

$$\chi = 1.5(W_{LC} + S_{LC})L_{LC}/\eta_{LC}, \tag{5.23}$$

$$\xi = \frac{1}{2}(B + B^r)(W_{GB} + S_{GB})L_{GB}/\eta_{GB}, \tag{5.24}$$

thus the area of the layers and the total IC area can be estimated as:

$$A_{LC} = \chi(G + \sqrt{G}), \tag{5.25}$$

$$A_{GB} = \xi K, \tag{5.26}$$

$$A = max\{A_{LC}, A_{GB}\} = max\left\{\chi(G + \sqrt{G}), \xi K\right\}. \tag{5.27}$$

Since the size of the global wire is much larger than that of the local one, when the average number of pins of IP blocks, K, is large enough, A_{GB} will dominate, or IC area depends on K. On the other hand, if K is small, IC area can be approximated by A_{LC}, and IC area depends on gate count G.

5.2.3 Estimation of Delay

The total delay D of a processing module includes the input and output delays and the processing delay to complete a given task. Note that the unit of delay is in second, so delay in cycles can be calculated as:

$$D_s = D_c t = D_c / f, \tag{5.28}$$

where D_s, D_c, t, f are the total delay in seconds and cycles, the period (in seconds), and operating frequency, respectively.

5.2.4 Estimation of Design Costs

Design cost is defined as the product of power, area, and delay:

$$Cost = P \times A \times D_s (\text{in } Wm^2 s), \tag{5.29}$$

where P is in W, A in m2, and D_s in seconds. Depending on the emphasis, P, A or D_s may carry different exponents. In this case, we assume they are equally important and thus assign exponent of 1 to each of P, A and D_s. From Equations (5.19) and (5.27), we have

$$Cost = \alpha[\beta G + \gamma(G + \sqrt{G}) + \theta K] V_{DD}^2 f \times max \left\{ \chi(G + \sqrt{G}), \xi K \right\}. \tag{5.30}$$

Since G is normally larger than 10,000, $\sqrt{G}$ is smaller than 1% of G. Therefore, $\sqrt{G}$ can be neglected, and Equation (5.30) becomes

$$Cost = \alpha \left[(\beta + \gamma) G + \theta K \right] V_{DD}^2 f \times max \left\{ \chi G, \xi K \right\}. \tag{5.31}$$

Sylvester and Keutzer (1999) showed that gates and local interconnects power consumption dominate total power consumption. In Equation (5.31), power consumption due to global interconnects becomes $\alpha\theta K V_{DD}^2 f$, while power due to gates and local interconnects become $\alpha(\beta+\gamma)GV_{DD}^2 f$. Therefore, Equation (5.30) can be simplified as:

$$Cost = \alpha(\beta+\gamma)GV_{DD}^2 f \times max\left\{\chi G, \xi K\right\}. \tag{5.32}$$

Since in the standard 2-input NAND cells, PMOS width is normally equal to about 0.8 times of NMOS width (AMS, 2006; IBM, 2008), while NMOS width can be approximately scaled with technology (channel length) at about 3.5 times (Ciletti, 2003). Therefore, Equation (5.4) becomes

$$\beta = 1.8 C_{ox} L_{channel} W_{NMOS} \approx 6.3 C_{ox} L_{channel}^2. \tag{5.33}$$

In addition, the gate oxide capacitance per unit area C_{ox} is calculated as follows (Rabaey et al., 2003):

$$C_{ox} = \frac{\epsilon_{ox}}{t_{ox}} = \frac{\epsilon_{rox}\epsilon_0}{t_{ox}}, \tag{5.34}$$

$$\epsilon_0 = 8.854 \times 10^{-12} (F/m), \tag{5.35}$$

where ϵ_{ox}, ϵ_0, ϵ_{rox}, t_{ox} are permittivity of SiO2, permittivity of free space (electric constant or vacuum permittivity), relative static permittivity (or dielectric constant) of SiO2, and gate oxide thickness, respectively. Therefore, Equation (5.33) becomes

$$\beta = 1.8 C_{ox} L_{channel} W_{NMOS} \approx 6.3 \frac{\epsilon_{rox}\epsilon_0}{t_{ox}} L_{channel}^2. \tag{5.36}$$

In a given SoC-based IC, if the average number of pins per IP block is small, $\chi G > \xi K$, Equation (5.32) becomes

$$Cost_G = (\psi G V_{DD}^2 f) G D_s, \tag{5.37}$$

where

$$\psi = \alpha(\beta + \gamma)\chi \tag{5.38}$$

$$\psi = 1.5\alpha \left(6.3 \frac{\epsilon_{rox}\epsilon_0}{t_{ox}} L_{channel}^2 + 1.5 \frac{\epsilon_{di}}{t_{di}} H_{LC} LLC \right) (W_{LC} + S_{LC}) L_{LC}/\eta_{LC} \tag{5.39}$$

$$\psi = 1.5\alpha\epsilon_0 \left(6.3 \frac{\epsilon_{rox}}{t_{ox}} L_{channel}^2 + 1.5 \frac{\epsilon_{rdi}}{t_{di}} H_{LC} LLC \right) (W_{LC} + S_{LC}) L_{LC}/\eta_{LC}, \tag{5.40}$$

where ϵ_{rdi} is relative static permittivity of the dielectric layer under the local wires. On the other hand, if the average number of pins per IP block is large, $\chi G < \xi K$, Equation (5.32) becomes

$$Cost_K = (\varphi G V_{DD}^2 f) K D_s, \tag{5.41}$$

where

$$\varphi = \alpha(\beta + \gamma)\xi \tag{5.42}$$

$$\varphi = \frac{1}{2}\alpha\epsilon_0 \left(6.3 \frac{\epsilon_{rox}}{t_{ox}} L_{channel}^2 + 1.5 \frac{\epsilon_{rdi}}{t_{di}} H_{LC} LLC \right) \times (B + B^r)(W_{GB} + S_{GB}) \frac{L_{GB}}{\eta_{GB}}. \tag{5.43}$$

In summary, given a system with IP blocks, design cost metric depends on whether the average number of pins per IP block is small or large. If it is small, the area due to local interconnect dominates and $Cost_G$ (Equation (5.37)) can be used. If it is exceedingly large, global interconnect area dominates, and $Cost_K$ (Equation (5.41)) can be used. In both conditions, design cost is proportional to delay.

In both cost formulas, ψ and φ parameters depend on technology design rules and process. In particular, they are proportional to the total number of gates and local interconnect capacitance, which depends on gate length $L_{channel}$, gate oxide thickness t_{ox}, and local interconnect thickness H_{LC} and length L_{LC} (Equation (5.40)), (Equation (5.43)).

5.3 The Proposed Performance-Cost Metric for FIT/IIT Designs

The throughput of a SoC-based data processing design is measured in the number of pixel-per-second (pps) (Wang et al., 2003; Chen et al., 2005, 2006; Shi et al., 2007; Hwangbo et al., 2007; Pastuszak, 2008; Choi et al., 2008; Do and Le, 2009, 2010) or pixel-per-cycle (ppc) (Chao et al., 2007; Ngo et al., 2008; Su and Fan, 2008), whose relationship is described in Equation (5.44). However, the former is commonly used in practice:

$$T_{pps} = T_{ppc} f \tag{5.44}$$

The throughput (pps, ppc) of IIT modules for $n \times n$ data blocks is computed by total delay (second, cycle) as shown below.

$$T_{pps} = \frac{n^2}{D_s}, \tag{5.45}$$

$$T_{ppc} = \frac{n^2}{D_c}. \tag{5.46}$$

In order to compare among designs with different performance and costs, we define a *performance-cost metric* (PCM) as follows:

$$PCM = \frac{Performance}{Cost} = \frac{T_{pps}}{Cost}. \tag{5.47}$$

Applying Equation (5.47) for IIT module design which processes $n \times n$ blocks, we have

$$PCM = \frac{n^2/D_s}{Cost}. \tag{5.48}$$

Since the design cost can be estimated using either Equation (5.37) or Equation (5.41) depending on the average number of pins per IP block, PCM now becomes

$$PCM_G \approx \frac{n^2/D_s}{(\psi G V_{DD}^2 f).G.D_s}, \tag{5.49}$$

$$PCM_K \approx \frac{n^2/D_s}{(\varphi G V_{DD}^2 f).K.D_s}. \tag{5.50}$$

In Equation (5.49), gate count and delay are equally important and have the inverse square effect to PCM. In Equation (5.50), delay has an inverse square effect, whereas gate count and average number of pins have an inverse linear effect on PCM. In both formulas, PCM also depends on technology design rules and process through ψ and φ parameters. In order to facilitate comparisons among designs with the same block size n, the same number of blocks B, and to hide technology design rules and process parameters ψ or φ, we define technology-hidden PCMs, $PCMG$ (used for SoC IP blocks having a small number of pins K) and $PCMK$ (used for SoC IP blocks having a large number of pins K), as follows:

$$PCMG = \frac{\psi PCM_G}{n^2} \approx \frac{1}{(G V_{DD}^2 f) G D_s^2}, \tag{5.51}$$

$$PCMK = \frac{\psi PCM_K}{n^2} \approx \frac{1}{(GV_{DD}^2 f)KD_s^2}. \tag{5.52}$$

Based on Equation (5.28), we have

$$PCMG \approx \frac{1}{(GV_{DD}^2 f)G\left(\frac{D_c}{f}\right)^2}, \tag{5.53}$$

$$PCMK \approx \frac{1}{(GV_{DD}^2 f)K\left(\frac{D_c}{f}\right)^2}. \tag{5.54}$$

In Equation (5.53), the technology-hidden *PCMG* is inversely proportional to the power, area, and delay. We note that there are a f component in the power and a $1/f^2$ component in the combined throughput-delay. As operating frequency increases, the power increases and thus *PCMG* decreases. However, as operating frequency increases, the $1/f^2$ factor decreases much faster at the quadratic rate, thus increases performance by a factor of f^2. The net effect of increasing operating frequency is an increase in *PCMG*. Similarly, the net effect of increasing operating frequency is an increase in *PCMK* as shown in Equation (5.54).

Similarly, let us define the technology-hidden C_G and C_K as:

$$C_G = \frac{Cost_G}{\psi} \approx (GV_{DD}^2 f).G.D_s = V_{DD}^2 G^2 D_c, \tag{5.55}$$

$$C_K = \frac{Cost_K}{\varphi} \approx (GV_{DD}^2 f).K.D_s = V_{DD}^2 GKD_c. \tag{5.56}$$

We also have

$$PCMG \approx \frac{f}{V_{DD}^2 G^2 D_c^2} = \frac{f}{C_G D_c}, \tag{5.57}$$

$$PCMK \approx \frac{f}{V_{DD}^2 GKD_c^2} = \frac{f}{C_K D_c}. \tag{5.58}$$

5.4 Discussion on PCMs on Different Designs and Metric Comparison to $DTUA$

5.4.1 General Discussion on Different Designs

We have reviewed many FIT/IIT designs in both 0.35 and 0.18 μm technologies. However, we only include designs in Pastuszak (2008); Choi et al. (2008); Ngo et al. (2008); Do and Le (2010); Chao et al. (2007); Su and Fan (2008), which support 8×8 FIT/IIT, and list them in Table 5.1. Since 8×8 IT is the most complex transform among the four types, we will evaluate the performance and cost of designs based on this transform. For designs whose bit-depth of a pixel was not reported, a 12-bit depth is assumed. Unless otherwise indicated, 0.18 μm technology is implied for all designs.

The design by Pastuszak (2008) supports all FITs/IITs and quantization/rescaling, except for 2×2 FHT/IHT. A 64-pixel input data bus and a 64-pixel output data bus are used resulting in a total data bus width of 2048 bits. The computing engine has eight shared 1-D hardware units and thirty-two quantization/rescaling multipliers. By pipelining all modules, the design is claimed to transform a 8×8 block in two cycles, giving a high aggregate throughput of thirty-two ppc. The gate counts of FIT/IIT, using 0.35 μm technology, are 115.3 and 99 Kgates, respectively. The gate counts of FIT/IIT using 0.18 μm technology, are 162.1 and 141.3 Kgates, respectively.

The design by Choi et al. (2008) supports most of the IIT operations except 2×2 Hadamard transform and de-quantization. A 32-pixel input and a 32-pixel output data bus are used resulting in a total data bus width of 768 bits assuming twelve bits per pixel. The computing unit has four shared 1-D hardware units. The design requires five cycles to complete an 8×8 IIT yielding an aggregate throughput of 12.8 ppc. The gate count is 20.7 Kgates using 0.35 μm technology.

The design by Ngo et al. (2008) supports all IITs with rescaling. A 8-pixel input and a 8-pixel output data bus are used resulting in a total data bus width of 256 bits. The design requires nineteen cycles to perform a 8×8 IIT including I/O delay yielding an aggregate throughput of 3.4 ppc. The gate count is 21.5 Kgates using 0.35 μm technology.

The design by Do and Le (2010) performs all FITs/IITs with quantization/rescaling. A 8-pixel input and a 8-pixel output data bus are used resulting in a total data bus width of 192 bits. The computing unit has two 8×8 1-D hardware units. By sharing computing resources and buffers for all transforms, each of the two hardware units alternatively operates on eight units of data without increasing the delay. With this alternating scheduling, only eight multipliers are required in each hardware unit. With the pipelined blocks, the buffers also help smoothen multiplications yielding an aggregate throughput of eight ppc. The gate counts of FIT/IIT are 62.2 and 47.3 Kgates, respectively.

The design by Chao et al. (2007) supports all IITs without rescaling. A 16-pixel input and a 8-pixel output data bus are used resulting in a total data bus width of 288 bits, assuming twelve bits per pixel. This design has one shared 1-D hardware unit. For each 8×8 block, sixteen cycles is required for processing and three cycles for scheduling, giving rise to an aggregate throughput of 3.4 ppc. The gate count of IIT taken as the core IIT and controller (not including buffers), is 9.5 Kgates.

The design by Su and Fan (2008) supports all the IITs and without rescaling. A 8-bit-per-pixel 64-pixel input data bus and a 12-bit-per-pixel 64-pixel output data bus are used resulting in a total data bus width of 1920 bits. The computing engine has one shared 1-D hardware unit. The throughput is sixteen cycles and aggregate throughput is 4.0 ppc. The gate count is 18.1 Kgates.

5.4.2 Discussion on Aggregate Throughput

In order to facilitate the discussion, four groups of designs are classified, including (1) group 1: 0.35 μm FIT designs; (2) group 2: 0.35 μm IIT designs; (3) group 3: 0.18 μm FIT designs; and (4) group 4: 0.18 μm IIT designs (as shown in Table 5.1). Note that group 1 has only one design and will not be analyzed further.

In group 2, as can be seen in column 10, the throughput in terms of pps (pps-throughput) of the design by Pastuszak (2008) is 5.1 times as high as that of the design by Ngo et al. (2008). This is mainly because the delay of Pastuszak (2008)'s design is 10.5% of that of Ngo et al. (2008)'s design, although its speed is only 52.6% of that of Ngo et al. (2008)'s design. The design by Choi et al. (2008) cannot be compared as there is no design with similar features.

In group 3, the pps-throughput of the design by Pastuszak (2008) is 1.9 times as high as that of the design by Do and Le (2010). This is mainly because the delay of Pastuszak (2008)'s design is 25% of that of Do and Le (2010)'s design, although its speed is only 47.6% of that of Do and Le (2010)'s design.

In group 4, among the designs without rescaling function, the pps-throughput of the design by Do and Le (2010) is 4.3 and 4.6 times as high as that of the design by Chao et al. (2007) and Su and Fan (2008), respectively. This is because the delay of Do and Le (2010)'s design is 41.7% and 50% of that of Chao et al. (2007)'s

TABLE 5.1: Performance-cost function *PCM* comparison of pre-layout synthesized designs.

Design	FIT/ IIT	Type of trans-form	Quan-tiza-tion	Tech. (μm)	Gate count (Kgates)	Speed (MHz)	Bus width (pel-s/bits)	Delay (cy-cles)	Throughput (ppc/Mpps)	C_G (GV^2 ppc)	C_K (MV^2 $bit\ ppc$)	$PCMG$ ($Hz/$ MV^2 ppc^2)	$PCMK$ ($Hz/$ KV^2 bit ppc^2)	$DTUA$ (pps)
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)
Pastuszak (2008)	FIT	4×4(H), 8×8 [a]	Yes	0.35	115.3	80	128/2048	2	32.0/2560.0	289.5	5143	138.1	7.8	22.2
Choi et al. (2008)	IIT	4×4(H), 8×8 [a]	No	0.35	20.7	27	64/768 [b]	5	12.8/345.6	23.3	865.6	231.4	6.2	16.7
Ngo et al. (2008)	IIT	All [c]	Yes	0.35	21.5	150	16/256	19	3.4/510.0	95.6	1138.8	82.5	6.9	23.7
Pastuszak (2008)	IIT	4×4(H), 8×8 [a]	Yes	0.35	99	82	128/2048	2	32.0/2624.0	213.5	4415.9	192.1	9.3	26.5
Pastuszak (2008)	FIT	4×4(H), 8×8 [a]	Yes	0.18	162.1	77	128/2048	2	32.0/2464.0	170.3	2151.2	226.1	17.9	15.2
Do and Le (2010)	FIT	All [c]	Yes	0.18	62.2	162.1	16/192	8	8.0/1296.8	100.3	309.5	202.1	65.4	20.8
Chao et al. (2007)	IIT	All [c]	No	0.18	9.5	125	24/288 [b]	19	3.4/425.0	5.6	168.4	1184.1	39.1	44.7
Su and Fan (2008)	IIT	All [c]	No	0.18	18.1	100	128/1920	16	4.0/400.0	17	1801.5	368	3.5	22.1
Do and Le (2010) [d]	IIT	All [c]	No	0.18	33.3	230.9	16/192	8	8.0/1847.2	28.7	165.7	1004.2	175.4	55.5
Pastuszak (2008)	IIT	4×4(H), 8×8 [a]	Yes	0.18	141.3	83	128/2048	2	32.0/2656.0	138.1	1937.6	300.4	21.4	18.8
Do and Le (2010)	IIT	All [c]	Yes	0.18	47.3	230.9	16/192	8	8.0/1847.2	58	235.4	497.7	122	39.1

[a] Design supports all IITs, except 2×2 inverse Hadamard transforms.
[b] Assuming the bit-depth of twelve bits for one pixel.
[c] Design supports all IITs, including 4×4, 8×8 IIT, and 4×4, 2×2 Hadamard transforms.
[d] The quantization in the design is removed to facilitate comparison with Chao et al. (2007); Su and Fan (2008).

and Su and Fan (2008)'s design, respectively, and its speed is 1.8 and 2.3 as high as that in the other two designs, respectively.

Also in group 4, among the designs with rescaling, the pps-throughput of the design by Pastuszak (2008) is 1.4 times as high as that of Do and Le (2010)'s design. This is because the delay of Pastuszak (2008)'s design is 25% of that of Do and Le (2010)'s design, although its speed is only 35.7% of that of Do and Le (2010)'s design.

In general, the design by Pastuszak (2008) has highest pps-throughput followed by those by Do and Le (2010) and Chao et al. (2007). Yet, throughput alone does not indicate the effectiveness of a design. We discuss $DTUA$s of the designs in the next section.

5.4.3 Discussion on $DTUA$

$DTUA$ increases when the throughput increases or the area decreases.

In group 2, $DTUA$ of the design by Pastuszak (2008) is 1.1 times as high as that of the design by Ngo et al. (2008). This is because the pps-throughput of Pastuszak (2008)'s design is 5.1 times as high as that of Ngo et al. (2008)'s design, although its gate count is 4.6 times as large as that of the other design.

In group 3, $DTUA$ of the design by Do and Le (2010) is 1.4 times as high as that of the design by Pastuszak (2008). This is because the gate count of Do and Le (2010)'s design is only 3.5% of that of Pastuszak (2008)'s design, although its pps-throughput is only 52.6% of the pps-throughput of Pastuszak (2008)'s design.

In group 4, among the designs without rescaling, $DTUA$ of the design by Do and Le (2010) is 1.2 and 2.5 times as high as that of the design by Chao et al. (2007) and Su and Fan (2008), respectively. This is mainly because the pps-throughput of Do and Le (2010)'s design is 4.3 and 4.6 times as high as that of Chao et al. (2007)'s and Su and Fan (2008)'s design; and its gate count is only 28.6% and 5.6% of that of the other two designs, respectively.

Also in group 4, among designs with rescaling, design in Do and Le (2010) is 2.1 times higher in $DTUA$ than that of Pastuszak (2008) mainly due to a 3.0-time smaller in gate count, despite a 1.4-time lower in pps-throughput.

In the previous section, the design by Pastuszak (2008) has higher pps-throughput followed by those by Do and Le (2010) and Chao et al. (2007). In this section, the performance is gauged by the area-based $DTUA$. The design by Pastuszak (2008) is no longer the best. The design by Do and Le (2010) has the highest in most cases.

$DTUA$ does not include power consumption and delay. In sub-micron technology, the area and power consumption of interconnects have played important roles. This is a weakness of $DTUA$ as a performance indicator.

5.4.4 Discussion on Design Costs

An assumption is made for comparison among different designs that the SoC includes only IIT module. Therefore, designs using the same technology can have the same ψ or φ. As a result, C_G and C_K can be used for $Cost_G$ and $Cost_K$ comparison, and $PCMG$ and $PCMK$ for PCM_G and PCM_K comparison. Since the widths of the address bus and control bus are small compared to those of the data bus in IIT, the average number of pins K for IIT module can be approximated by the data bus width.

In Table 5.1, columns 11 and 12 show C_G (Equation (5.55)) and C_K (Equation (5.56)) of different designs, respectively. Note that C_G is proportional to the square of both operating voltage and gate count, and linearly proportional to the delay. On the other hand, C_K is proportional to the square of operating voltage, and linearly proportional to gate count, bus width, and delay. Within a design technology, since the operating voltage is the same, it can be left out of the analysis.

In group 2, C_G of the design by Ngo et al. (2008) is only 45.4% of that of Pastuszak (2008)'s design. This is mainly because the gate count of Ngo et al. (2008)'s design is only 21.7% of that of Pastuszak (2008)'s design, although its delay is 9.5 times as long as the other design's delay. Likewise, cost C_K of the design by Ngo et al. (2008) is 25.6% of that of Pastuszak (2008)'s design. This is mainly because its bus width and gate count are both smaller, which are only 12.5% and 21.7%, respectively, of those of Pastuszak (2008)'s design, although its delay is 9.5 times as long as the delay in Pastuszak (2008)'s design.

In group 3, C_G of the design by Do and Le (2010) is only 58.8% of that of the design by Pastuszak (2008). This is mainly because the gate count of Do and Le (2010)'s design is only 38.5% of that of Pastuszak (2008)'s design, although its delay is 4.0 times as long as the other delay. Likewise, cost C_K of the design by Do and Le (2010) is only 14.5% of that of Pastuszak (2008)'s design. This is because its bus width is only 9.3% of the Pastuszak (2008)'s design's bus width, although its delay is 4.0 times as long as the other delay.

In group 4, among designs without rescaling, C_G of the design by Chao et al. (2007) is only 33.3% and 19.6% of that of the design by Su and Fan (2008) and Do and Le (2010), respectively. This is mainly because the gate count of Chao et al. (2007)'s design is only 52.6% and 28.6% of that of the design by Su and Fan (2008) and Do and Le (2010), respectively, although its delay is 1.2 and 2.4 times as long as the other two designs' delays, respectively. On the other hand, design in Do and Le (2010) is slightly smaller and its C_K is only 9.2% of that of Chao et al. (2007)'s and Su and Fan (2008)'s design, respectively. This is because its bus width is only 66.7% and 10.0% of that of the other two designs, respectively.

Among designs with rescaling, C_G of the design by Do and Le (2010) is only 41.7% of that of the design by Pastuszak (2008). This is mainly because the gate count of Do and Le (2010)'s design is only 33.3% of that of Pastuszak (2008)'s design, although its delay is 4.0 times as long as the other delay. Likewise, C_K of Do and Le (2010)'s design is only 12.2% of that of Pastuszak (2008)'s design because its bus width is only 9.3% of the other bus width.

In general, designs by Do and Le (2010) and Chao et al. (2007) have the lowest costs, where Chao et al. (2007)'s design is without rescaling. Among C_G and C_K, it is easier to choose reasonable bus width for lowest C_K than to find smallest gate count for lowest C_G.

In summary, up to this section, the design by Pastuszak (2008) has the best aggregate pps-throughput followed by those by Do and Le (2010) and Chao et al. (2007). However, the design by Do and Le (2010) has the highest $DTUA$ followed by Pastuszak (2008)'s design, and those by Do and Le (2010) and Chao et al. (2007) have the lowest costs.

5.4.5 Discussion on PCMs with respect to $DTUA$

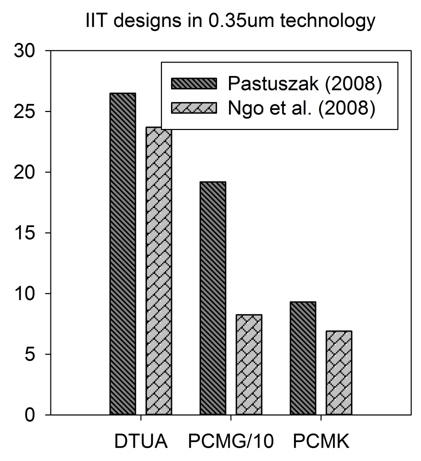
In Table 5.1, columns 13 and 14 show $PCMG$ (Equation (5.53)) and $PCMK$ (Equation (5.54)) of all designs. Figure 5.3 shows the bar graphs of the values of $DTUA$s, $PCMG/10$ (to be able to fit in the same graphs), and $PCMK$ among the designs in the various comparison groups.

We note that $PCMG$ (Equation (5.57)) is proportional to operating frequency, and inversely proportional to the cost C_G and delay. $PCMK$ (Equation (5.58)) is proportional to operating frequency, and inversely proportional to the cost C_K and the delay.

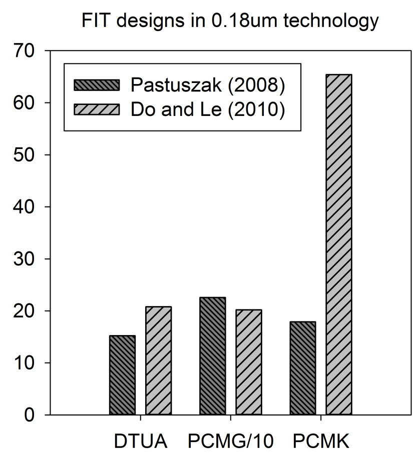
In group 2, $PCMG$ of the design by Pastuszak (2008) is 2.3 times as high as that of the design by Ngo et al. (2008). This is mainly because the delay in Pastuszak (2008)'s design is only 10.5% of that of Ngo et al. (2008)'s design, although its cost C_G is 2.2 times as high as C_G of Ngo et al. (2008)'s design and its operating frequency is 55.6% of that of Ngo et al. (2008)'s design. On the other hand, $PCMK$ of the design by Pastuszak (2008) is 1.3 times as high as that of Ngo et al. (2008). This is mainly because the delay in Pastuszak (2008)'s design is only 10.5% of that of Ngo et al. (2008)'s design, although its cost C_K is 3.9 times as large as Ngo et al. (2008)'s design'sC_K and its frequency is only 55.6% of the other frequency.

Together with $DTUA$, this is illustrated in Figure 5.3(a). Even though $DTUA$ and $PCMG$ of Pastuszak (2008) are relatively higher than those of Ngo et al. (2008), its $PCMK$ is lower. The much larger bus width of Pastuszak (2008) is reflected in the lower $PCMK$ compared to that in Ngo et al. (2008).

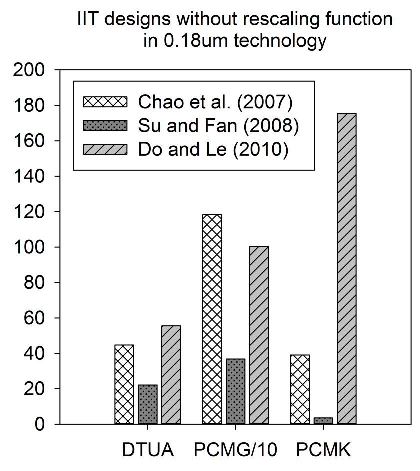
The question is when to use $PCMG$ and when to use $PCMK$? It has shown that the design by Pastuszak (2008) has good performance due to its high throughput,



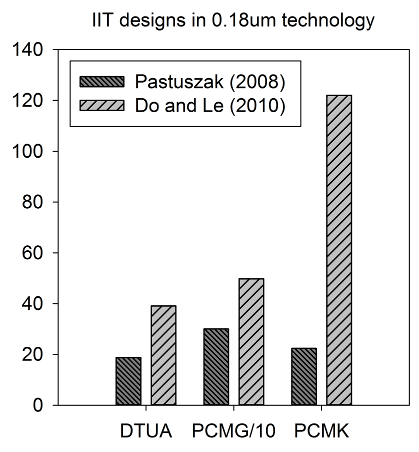
(a)



(b)



(c)



(d)

FIGURE 5.3: *DTUA*, *PCMG*/10, and *PCMK* among the designs in the various comparison groups. (a) Group 2, 0.35μm, IIT; (b) Group 3, 0.18μm, FIT; (c) Group 4, 0.18μm, IIT without rescaling; and (d) Group 4, 0.18μm, IIT with rescaling.

high *DTUA*, and high *PCMG* value. However, when the technology shrinks, the interconnection issues start to dominate and thus *PCMK* cost becomes large. It is suggested to use *PCMK* in sub-micron technology or when bus width is exceedingly large, and to use *PCMG* otherwise.

In group 3, $PCMG$ of the design by Do and Le (2010) is similar to that of the design by Pastuszak (2008), while its $PCMK$ is 3.7 time as large as that of Pastuszak (2008)'s design. This is because C_G and C_K of Do and Le (2010)'s design are only 58.8% and 14.3% of those of Pastuszak (2008)'s design, respectively; and its operating frequency is 2.1 times as fast as that of Pastuszak (2008)'s design, although its delay is 4.0 times longer than the other delay. As shown in Figure 5.3(b), the design by Do and Le (2010) has higher $DTUA$, comparable $PCMG$, and much higher $PCMK$ indicating that Do and Le (2010) is more favorable over Pastuszak (2008).

In group 4, among the designs without rescaling, the design by Chao et al. (2007) has the highest $PCMG$, which is 3.2 and 1.2 times as high as those of Su and Fan (2008)'s and Do and Le (2010)'s designs, respectively. This is because C_G of Chao et al. (2007)'s design is only 33.3% and 19.6% of those of Su and Fan (2008)'s and Do and Le (2010)'s designs, respectively, although its delay is 1.2 and 2.4 times as long as the delays in the other two designs, respectively. The design by Chao et al. (2007) specifically targets small gate count compared to others.

On the other hand, the design by Do and Le (2010) has the highest $PCMK$, which is 4.4 and 49.6 times as high as those of the designs by Chao et al. (2007) and Su and Fan (2008). This is because its operating frequency is 1.8 and 2.3 times as fast as those of the other two designs; its delay is only 41.7% and 50% of the other two delays; and its cost C_K is 100% and 9.2% of those in the other two designs, respectively. The design by Do and Le (2010) is very much larger in $PCMK$ compared to that of Su and Fan (2008). This is because Su and Fan (2008) requires extremely large I/O bus of 1920 bits, compared to 192 bits. This is another scenario where $DTUA$ clearly fails to report. This is shown in Figure 5.3(c). Clearly, the design by Su and Fan (2008) has the lowest values in all metrics. Chao et al.

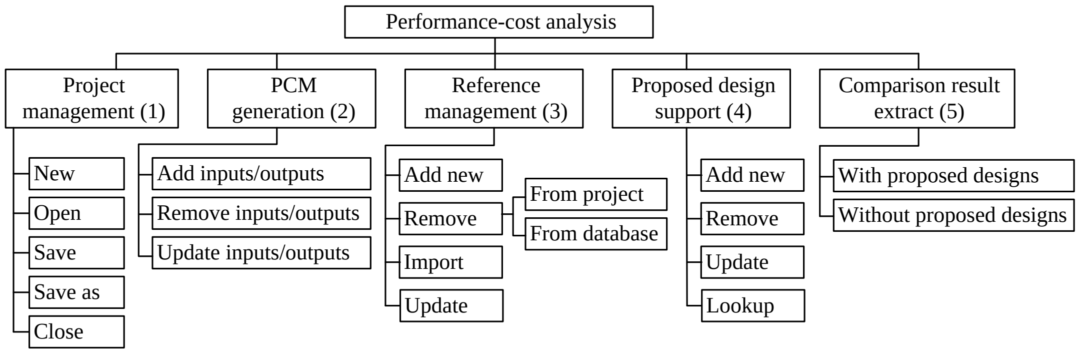


FIGURE 5.4: PCAS functions.

(2007) dominates in *PCMG* (and *DTUA*), while Do and Le (2010) dominates in *PCMK*.

Among the designs with rescaling function, *PCMG* and *PCMK* of the design by Do and Le (2010) are 1.6 and 5.6 times as high as those of the design by Pastuszak (2008). This is because C_G and C_K of Do and Le (2010)'s design are only 45.4% and 12.5% of those of Pastuszak (2008)'s design, respectively, although its delay is 25% of the other delays. This is shown in Figure 5.3(d). The three metrics show the same trend when *DTUA*, *PCMG* and *PCMK* of Do and Le (2010) is 2.1, 1.6 and 5.6 times larger than those of Pastuszak (2008).

In general, *DTUA* provides conventional view on assessing the performance-cost based on the area-only cost function. PCMA provides the area-centric cost function, while *PCMK* provides the interconnection-centric cost existed mostly in submicron designs with large number of pins.

DTUA has been used to assess the performance of a design using only throughput and area. If interconnections and large number of I/O pins, power consumption are concerned, *DTUA* fails to report. Thus, the most performed designs should have highest values in both *PCMG* and *PCMK*.

5.5 Performance-Cost Analysis Software

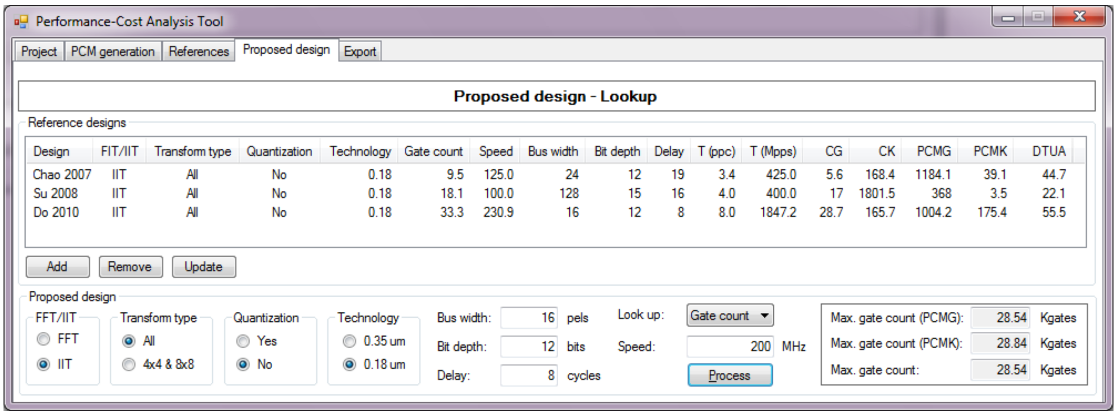
5.5.1 Overview of PCAS functions

Based on *PCM*, a performance-cost analysis software (PCAS) has been developed to analyze and compare users' designs with reference designs. In particular, it helps to manage the references, generate different metric formulas, analyze the designs based on these metrics, lookup the allowed boundaries of their designs in order to have the best designs, and export comparison tables. In addition, due to a flexible function design, PCAS can be used not only for FIT/IITs using *PCM* but also other designs and metrics.

5.5.2 PCAS function description

The detail functions of PCAS are illustrated in Figure 5.4. Branch 1 of the figure lists the functions for users to organize their work in projects. Functions that allow users to manage metrics, the central of analysis and comparison, are illustrated in Branch 2. In a project, different metrics, i.e., outputs, can be created and reused in other projects. Formulas of the metrics can be generated and modified based on variables, i.e., design input parameters. These inputs also can be added or removed. In the FIT/IIT case study, the outputs are throughput, C_G, C_K, *PCMG*, *PCMK* and *DTUA*, while the inputs are G, f, K and D_C. There also are classified inputs such as whether FIT or IIT the module is, whether quantization step is included and which technology is used. In addition, as published designs are also essential for analysis and comparison, PCAS provides functions to manage them for reference (Branch 3). Users can add or remove reference designs in current project or database. Once the references are added to a project, they are stored in the database and can be reused in other projects; thus, the users do not have to add them again. On the other hand, when a user removes a reference, it is removed by default only in the current project, not in database, since other projects might still use it. When a reference is added, the user needs to provide all the required input information of the reference. Besides adding and removing, importing function is available for users to import a reference from database to the current project. In addition, users can update information of a reference, leading to the update of the database and all the projects.

The main object of a project is the proposed design. Similar to reference designs, the proposed design is managed by adding, removing and updating functions (Branch 4). In addition, when users design a new architecture, they probably need to know how good their design is compared to the existing ones. PCAS can provide a suggestion through the lookup function. The software allows users to choose one input and one output for the lookup function, in which the input is the lookup parameter for the new design and the output is the main metric. Next, the other input parameter values of the new design need to be provided. PCAS starts to compute all the metrics of the references, then search for the best designs based on the main metric, and finally compute the optimal value for the lookup parameter of the new design so that the users can use it as a goal to achieve a design which is better than the best reference designs. In the FIT/IIT case study, gate count or operating frequency (speed) can be chosen as the lookup parameter, while *PCMG* and *PCMK* can be selected as the main metrics. PCAS helps the users analyze the possible maximum gate count or minimum speed of their proposed designs based on its preliminary parameters in order to have a higher *PCMG* and *PCMK* compared to the highest *PCMG* and *PCMK* of the reference designs (Figure 5.5).



Proposed design - Lookup

Reference designs

Design	FIT/IIT	Transform type	Quantization	Technology	Gate count	Speed	Bus width	Bit depth	Delay	T (ppc)	T (Mpps)	CG	CK	PCMG	PCMK	DTUA
Chao 2007	IIT	All	No	0.18	9.5	125.0	24	12	19	3.4	425.0	5.6	168.4	1184.1	39.1	44.7
Su 2008	IIT	All	No	0.18	18.1	100.0	128	15	16	4.0	400.0	17	1801.5	368	3.5	22.1
Do 2010	IIT	All	No	0.18	33.3	230.9	16	12	8	8.0	1847.2	28.7	165.7	1004.2	175.4	55.5

Proposed design

FFT/IIT: FFT [✓ IIT]
Transform type: [✓ All] 4x4 & 8x8
Quantization: Yes [✓ No]
Technology: 0.35 um [✓ 0.18 um]

Bus width:	16	pels
Bit depth:	12	bits
Delay:	8	cycles
Look up:	Gate count	
Speed:	200	MHz
Max. gate count (PCMG):	28.54	Kgates
Max. gate count (PCMK):	28.84	Kgates
Max. gate count:	28.54	Kgates

FIGURE 5.5: PCAS lookup function.

Finally, PCAS can export comparison results to a table (Branch 5). The table looks similar to Table 5.1 and may or may not contain the information of the new design.

5.5.3 Optimal value calculation for look-up parameter in FIT/IIT case study

In the FIT/IIT case study, assuming that gate count is chosen as the lookup parameter, and $PCMG$ is the main metric. Assuming that design X is the best among all reference designs, i.e., having the highest $PCMG$, and design A is the current design which is being processed by the lookup function. From Equation (5.57), we have

$$PCMG_A \approx \frac{f_A}{V_{DD_A}^2.G_A^2.D_{C_A}^2}, \tag{5.59}$$

$$PCMG_X \approx \frac{f_B}{V_{DD_Z}^2.G_X^2.D_{C_X}^2}. \tag{5.60}$$

The current design A is better than the best reference design X when

$$PCMG_A > PCMG_X, \tag{5.61}$$

$$\frac{f_A}{V_{DD_A}^2 . G_A^2 . D_{C_A}^2} > \frac{f_B}{V_{DD_Z}^2 . G_X^2 . D_{C_X}^2}. \tag{5.62}$$

As technology is used to classify the designs, we only select the same technology for comparison. This means A and X have the same working voltage V_{DD}. So we have

$$G_A < \frac{G_X D_{C_X}}{D_{C_A}} \sqrt{\frac{f_A}{f_X}}. \tag{5.63}$$

Therefore, the gate count of the new design A needs to be smaller than $\frac{G_X D_{C_X}}{D_{C_A}}$ so that A is better than the best reference design X.

Similarly, if gate count and *PCMK* are the chosen input/output, and Y is the highest-*PCMK* reference, we have

$$G_A < \frac{G_Y K_Y D_{C_Y}^2}{K_A D_{C_A}^2} \frac{f_A}{f_Y}. \tag{5.64}$$

Overall, in order to have the best design in terms of both *PCMG* and *PCMK*, the gate count of the new design A needs to be smaller than the smaller value between $\frac{G_X D_{C_X}}{D_{C_A}} \sqrt{\frac{f_A}{f_X}}$ and $\frac{G_Y K_Y D_{C_Y}^2}{K_A D_{C_A}^2} \frac{f_A}{f_Y}$.

Similar to the gate count lookup process, speed lookup function will compute the optimal value for speed. In order to have a better design in terms of both *PCMG* and *PCMK*, the speed of the new design A need to be smaller than the smaller value between $\frac{f_X G_A^2 D_{C_A}^2}{G_X^2 D_{C_X}^2}$ and $\frac{f_Y G_A K_A D_{C_A}^2}{G_Y K_Y D_{C_X}^2}$ (using *PCMG* and *PCMK* metrics, respectively).

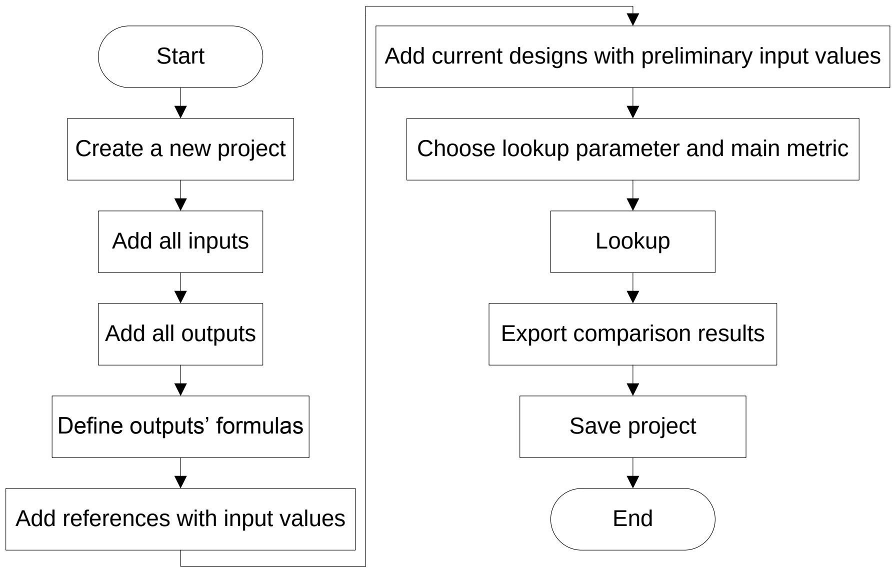

Start
Create a new project
Add all inputs
Add all outputs
Define outputs' formulas
Add references with input values
Add current designs with preliminary input values
Choose lookup parameter and main metric
Lookup
Export comparison results
Save project
End

FIGURE 5.6: PCAS example flowchart.

5.5.4 Using PCAS example in FIT/IIT case study

This part presents an example of using PCAS in the FIT/IIT case (Figure 5.6). After creating a project, defining all inputs, and generating all output formulas, three references (Chao et al., 2007; Su and Fan, 2008; Do and Le, 2010) are added with their inputs. Their outputs are then automatically computed and presented in Figure 5.5. The users may want to design an IIT with the shown inputs. If the users estimate their design speed as 200MHz, and choose to lookup for the gate count, PCAS can analyze the references and compute the possible maximum gate counts. The results are 28.54 Kgates and 28.84 Kgates for *PCMG* and *PCMK* metrics, respectively. As a result, in order to have better *PCM*s than the references, the new design must have the gate count smaller than 28.54 Kgates.

5.5.5 Flexible design of PCAS

PCAS is developed with the aim to facilitate the use of *PCM* technique. Moreover, PCAS is flexibly designed so that it can facilitate different metrics for other architectures. Its flexibility is supported by the followings arguments. Firstly, the inputs and outputs of projects are easily added or removed. Secondly, the formulas of the outputs are modifiable with *PCM*s as the default formulas. Thirdly, the lookup parameter and the metric can be arbitrarily chosen among the inputs and outputs for the lookup function.

5.6 Summary

High performance and low cost are design objectives for SoC and IP designs in general, and H.264 forward/inverse integer transform (FIT/IIT) designs in particular. The *DTUA* has been the metric in the literature for comparisons among the high throughput and area-efficient FIT/IIT designs. However, due to the incomprehensiveness of the current metric(s) used for comparison, some designs use very large bus widths but their authors were still able to claim their area efficiencies. In this chapter, a novel *PCM* concept is proposed for the H.264 forward/inverse integer transforms. *PCM* is defined as the ratio of data throughput over the design cost - a product of power, area, and delay. Compared to *DTUA*, *PCM* facilitates more comprehensive comparisons among the FIT/IIT designs.

The design by Do and Le (2010) can be considered as the best FIT and IIT design in groups 3 and 4 in 0.18 μm technology, respectively. Besides, in group 4 without quantization function, the design by Chao et al. (2007) is the second best. On the other hand, the design by Pastuszak (2008) is considered as the best IIT design in group 2 with quantization.

Performance-cost analysis software is subsequently proposed to facilitate the use of the *PCM* technique. Using the software, users can manage the reference designs, generate analyzing formulas, analyze and lookup the allowed boundaries in their designs and export comparison results. PCAS is flexibly designed in order to facilitate the use of not only our *PCM* technique for FIT/IITs, but also other metrics for other architectures.

Chapter 6

Fast and Low-Cost Algorithms and A High-Throughput Area-Efficient Architecture for HEVC Integer Transforms

6.1 Introduction

Motivated by the impressive coding efficiency and phenomenal success of H.264/AVC in industry, and a high growth in demand for band-width driven video applications (such as 3-D, multi-view, web-based, smart phone and tablet applications), the H.264 developers, ISO/IEC Moving Picture Experts Group and ITU-T Visual Coding Experts Group, have been working together again to develop a novel High Efficiency Video Coding (HEVC) standard. It is currently finalizing and going to be in early 2013, with the aim of (1) supporting increased resolution videos, i.e., beyond full high definition; and (2) saving half of bit-rate with equivalent quality for high definition (HD) and full high definition (FHD) resolution videos compared to the current H.264/AVC standard. However, together with the improvement on compression capability, the complexity of HEVC decoder and encoder is about 1.5 and several times, respectively, of those of H.264/AVC (Bossen et al., 2012).

In HEVC, transform coding is still one of the most important coding tools. In the H.264 high profiles, residual data are transformed in blocks with sizes of up to 8×8. In HEVC, a wide range of block sizes, from 4×4 to 32×32, is used to adapt the transforms to the varying space-frequency characteristics of residual data. As a result, the computational complexity of the integer transforms is dramatically increased. On the other hand, in order to support beyond-FHD resolutions, HEVC coding tools in general, and transform coding in particular need to achieve a very high throughput. However, all the core transform matrices in HEVC are totally different from those of H.264/AVC. It is desired to develop fast transform and low-complexity algorithms, and high throughput and area-efficient architectures for the HEVC forward and inverse integer transforms.

Video encoders are always more complex than video decoders. Therefore, more effort should be made to reduce the complexity of the modules in video encoders, especially in HEVC encoder.

However, there are not many forward or inverse transform algorithms and architectures reported in the literature till now (December 2012). In the HEVC test models (HMs), the Partial Butterfly algorithms with butterfly additions and scalar multiplications are used. Due to the multiplications, they are far more complex than the H.264 fast transform algorithms and are questionable to achieve a high throughput to support beyond-FHD videos in hardware implementation. Rithe et al. (2012) proposed an algorithm and architecture for the 4×4 and 8×8 2-D transforms with hardware implementation. However, it is not feasible to extend their work to larger sizes of transforms because the algorithms are developed based on the H.264 transform matrices. Due to the similar reason, the reported 8×8 1-D inverse transform architecture design by Martuza and Wahid (2012) is not applicable to other transform sizes. In addition, cost is set at the highest priority in the design by Martuza and Wahid (2012) instead of throughput. It should be noted that in order to enable to support beyond-FHD videos in HEVC, throughput is the crucial parameter in design optimizations. As a result, the maximum resolution that the design by Martuza and Wahid (2012) can 2-D transform is below FHD.

There is an urgent need to develop fast and low-cost algorithms and high-throughput architectures for HEVC transforms with the sizes from 4×4 to 32×32. Due to the complexity of the 32×32 transform, manual development of its fast algorithm is challenging. We explore in this chapter the way to develop fast transform algorithms for all the sizes of the HEVC transforms to facilitate high throughput designs. We propose a novel method to automatically generate fast algorithms even for 32×32 transforms. Based on the proposed method, we develop a series of 4×4 and 8×8 hardware-oriented fast and low-cost transform algorithms for HEVC. Fast and low-cost transform algorithms for larger sizes can also be developed using the same method. Finally, we develop, implement and fabricate a high-throughput and area-efficient architecture based on the proposed algorithms.

The experimental results show that both the running time and the cost of the automatically generated implementations for scalar multiplication algorithms by the proposed method are about 90% less than those of the original implementation by multiplications. The method is computationally feasible to be applied to all the Partial Butterfly algorithms from 4×4 to 32×32. The running time and the cost of the proposed fast and low-cost transform algorithms are 75% and 87%

less than that of the original Partial Butterfly algorithms in HMs, respectively. Compared to the reported HEVC transform algorithms in the literature, the proposed algorithms are faster and their costs are lower. With a number of proposed techniques, under a very challenging constraint on the I/O pin count of half-pixel, the proposed architecture can support the transforms for up to Quad-Full High Definition (QFHD) videos at the progressive scan frequency of 30 Hz. This is eight times as large as that of Martuza and Wahid (2012)'s design. The proposed architecture consumes only 44% power of the Martuza and Wahid (2012)'s design.

The chapter is organized as follows. Section 6.2 describes the Original Partial Butterfly Transform Algorithms in HEVC test model HM. In Section 6.3, a novel optimization method for scalar-multiplication-containing algorithms and a series of novel integer transform algorithms for HEVC are proposed. Section 6.4 introduces a novel high-throughput and area-efficient architecture. The chapter ends with a summary in Section 6.5.

6.2 The Partial Butterfly Transform Algorithms

In general, a 2-D forward/inverse transformation of a block can be computed by repeatedly applying the 1-D forward/inverse transform algorithms to all rows and columns of the block. The forward transformation of a residual block includes two stages. In the first stage, all rows are 1-D transformed by applying an 1-D transform algorithm; while in the second stage, all columns of the first stage's result are transformed using the same algorithm.

The original 4×4 and 8×8 1-D Partial Butterfly forward transform algorithms in HM7.0 (Appendix) are illustrated in Figure 6.1 and Figure 6.2, respectively. As

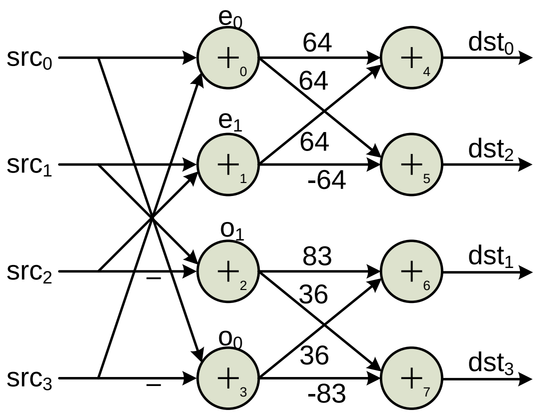


FIGURE 6.1: The 4×4 1-D Partial Butterfly forward transform algorithms.

can be seen, four inputs (from src_0 to src_3) or eight inputs (from src_0 to src_7) are added, subtracted and multiplied to produce four outputs (from dst_0 to dst_3) or eight outputs (from dst_0 to dst_7), respectively. Each arrow represents a data flow from its tail to head. Each number on one arrow represents a multiplication of the data by the number. Each adder symbol in front of dst_1,dst_3, dst_5 and dst_7 in Figure 6.2 represents one 4-input adder or three 2-input adders. Especially, the dashed region in this figure has the same structure with Figure 6.1. Therefore, the 4×4 1-D Partial Butterfly algorithm can be implemented as a part of the 8×8 1-D Partial Butterfly algorithm. Table 6.1 lists the number of additions and multiplications needed in the 4×4 / 8×8 1-D/2-D transforms using the Partial Butterfly algorithms. The total number of 2-input additions at Column 4 of Table 6.1 is calculated based on the fact that a 16-bit multiplication can be implemented by fifteen 2-input additions. As can be seen from Table 6.1, as the $4 \times 4/8 \times 8$ 1-D Partial Butterfly algorithms contain a number of multiplications, their 2-D transformations are relatively complex, leading to long running times with the longest calculation path of nineteen adders for the 1-D 8×8 algorithm (Table 6.2) and a high resource consumption of 6280 adders for the 2-D 8×8 algorithm, which consequently leads to a low throughput and a large area, respectively. Therefore,

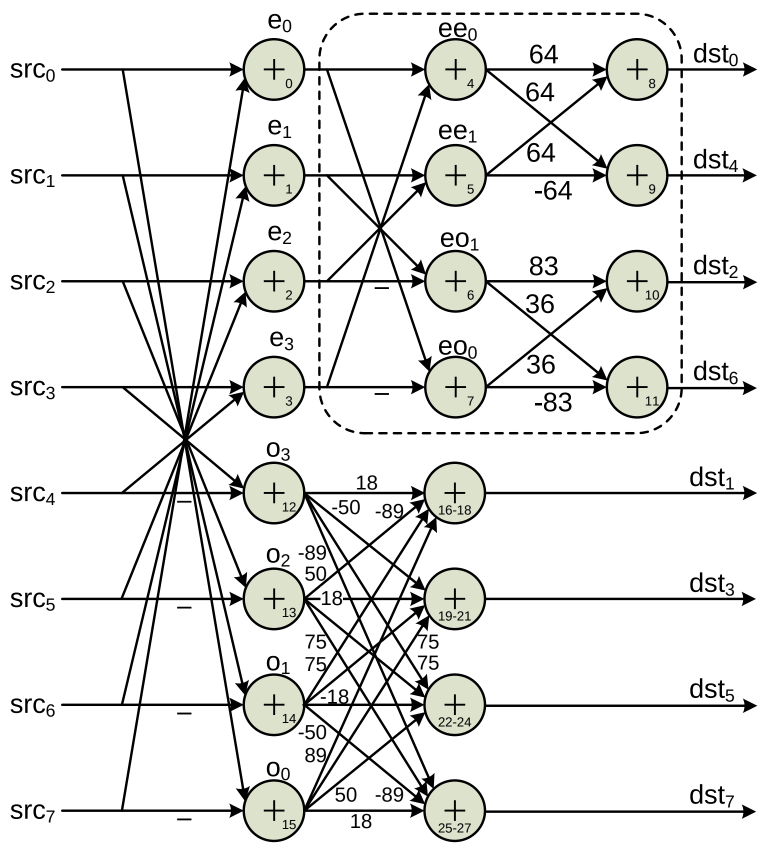


FIGURE 6.2: The 8 × 8 1-D Partial Butterfly forward transform algorithms.

TABLE 6.1: Number of additions and multiplications in the Partial Butterfly algorithms for the 4 × 4/8 × 8 1-D/2-D forward transforms.

Transform	Number of 16-bit Multiplications	Number of 2-input Additions / Subtractions	Total number of 2-input Additions
4×4 1-D	8	8	$8 \times 15 + 8 = 128$
8×8 1-D	24	28	$24 \times 15 + 28 = 388$
4×4 2-D	64	64	$64 \times 15 + 64 = 1024$
8×8 2-D	384	448	$384 \times 15 + 448 = 6208$

TABLE 6.2: The length of the longest path of the 1-D Partial Butterfly algorithms for the 4 × 4/8 × 8 1-D forward transforms.

Transform	Number of adders used in the longest path
4×4 1-D	1 adder + 1 multiplier + 1 adder = 17 adders
8×8 1-D	1 adder + 1 multiplier + 3 adders = 19 adders

it is desired to have develop complex 1-D transform algorithms which require less time to perform and consume less resource than the Partial Butterfly algorithms.

6.3 A Novel Optimization Method for Scalar-Multiplication-Containing Algorithms and Novel Integer Transform Algorithms for HEVC

The complexity caused by the scalar multiplications in the Partial Butterfly algorithms leads to a low throughtput and large area. This can be reduced by converting scalar multiplications to addition/subtractions. It should be noted that each scalar multiplication is considered as a set of multiplications of a number by different multipliers. For example, a multiplication by 18 can be implemented by using two shift operations and one addition/subtraction as $x \times 18 = x \ll 4 + x \ll 1$. Instead of using a 16-bit multiplication, using one 16-bit addition and two shift operations is definitely less complex.

Several algorithms have been proposed in the literature to convert constant multiplications to addition/subtractions, including Canonical Sign Digit (CSD) (Hewlitt and Swartzlander, 2000), Minimal Signed Digit (MSD) (Park and Kang, 2001) and Booth encoding (Patterson and Hennessy, 1998). However, none of them deal with scalar multiplication conversion.

A multiplication may be converted to addition/subtractions in different ways. For instance, a multiplication by 7 can be performed by using two shift operations and two addition/subtractions as $x \times 7 = x \ll 2 + x \ll 1 + x$, or can be performed using one shift operation and one addition/subtraction as $x \times 7 = x \ll 3 - x$, which is less complex than the previous way. As can be seen, different conversions may have different complexities, affecting running time and resource usage of algorithm implementation. Therefore, finding the least complex Multiplication-to-Addition Conversion Result (MACR) for each multiplication component plays an essential role in improving the original Partial Butterfly algorithms in terms of complexity reduction.

Even when the least complex MACR for a multiplication is found, different operation orders may lead to different running times and different throughputs. This is because operations may be performed in parallel in hardware implementation. For example, a multiplication by 89, which is converted to additions as $x \times 89 = x \ll 6 + x \ll 4 + x \ll 3 + 1$, may be performed in $((x \ll 6 + x \ll 4) + x \ll 3) + 1$ order, or in $(x \ll 6 + x \ll 4) + (x \ll 3 + 1)$ order, or in $(x \ll 6 + 1) + (x \ll 4 + x \ll 3)$ order, or in other orders. In the first operation order, the required time to perform the multiplication is about three times as large as the time required for an addition. This ratio is about two times in the second and the third operation orders as two additions in the two brackets can be performed simultaneously. The time needed for shift operations is omitted in hardware designs as the shift operations will be implemented by shifting wires.

Furthermore, the Partial Butterfly algorithms contain scalar multiplications, where each scalar multiplication is considered as a set of multiplications of a number by different multipliers. In order to ensure a high throughput of HEVC hardware implementation for high definition video compression, the multiplications are required to perform in parallel. Although the multiplications are different, it is possible to find some Common Operation Regions (CORs) in the implementation of the multiplications by optimizing the operation orders of the least complex MACRs. Let's take multiplications by 6 and 13 as an example. The multiplication by 6, $x \times 6$, has the least complex MACR of $x \ll 2 + x \ll 1$ and the multiplication by 13, $x \times 13$, has the least complex MACR of $x \ll 3 + x \ll 2 + x$. While the multiplication by 6 has only one operation order, the multiplication by 13 may have three operation orders with the same running time, which is also the shortest among all the orders. They are (1) $(x \ll 3 + x \ll 2) + x$; (2) $x \ll 3 + (x \ll 2 + x)$; and (3) $(x \ll 3 + x) + x \ll 2$. If order (1) is selected for the multiplication by 13, a Common Operation Region (COR) found between the two multiplications is $x \ll 1 + x$, hence $x \times 6 = (x \ll 1 + x) \ll 1$ and $x \times 13 = (x \ll 1 + x) \ll 2 + x$. However, if order (2) or (3) is selected, no COR can be detected. If CORs are found among the multiplications, they just need to be performed once for all or a number of multiplications, and the multiplications then can utilize their outputs. Hence, the resource can be saved and the hardware area can be reduced. In this example, the implementation of the multiplications by 6 and 13 with order (1), after reformulated based on the COR, only needs two addition/subtractions instead of three. Therefore, after having the least complex MACRs for all multiplications, determining their best operation orders with the shortest running time to achieve the largest CORs is the key to improve the Partial Butterfly algorithms and build up fast and low-cost integer transform algorithms for HEVC.

In this section, we propose (1) a novel optimization method on complexity, running time and resource consumption for scalar-multiplication-containing algorithms; and (2) a series of novel algorithms named hardware-oriented fast and low-cost integer transform algorithms for HEVC. The optimization method (Figure 6.3) can be applied to algorithms which (a) contain scalar multiplications; (b) enable operations to perform in parallel; and (c) require a short running time with a low resource consumption. These algorithms are undergone through a general Multiplication-Addition Conversion step and three levels of optimization including

(i) Complexity Optimization, (ii) Timing Optimization and (iii) Resource Optimization (Figure 6.3). In the first level of optimization, a Complexity Optimization algorithm is proposed to find the least complex Multiplication-to-Addition Conversion Result (MACR) for each multiplication of the original scalar-multiplication-containing algorithms, e.g., the Partial Butterfly algorithms. In other words, the first optimization level minimizes the number of additions for each MACR from the previous general Multiplication-Addition Conversion step. In the second optimization level, a Timing Optimization scheme is proposed to (1) generate a hardware-oriented adding tree for each minimized MACR with a given or a shortest running time; and to (2) find all possible options to assign addition operands of the minimized MACR to the leaves of the adding tree. As each option leads to an operation order, each minimized MACR with its optimized adding tree has a set of different operation orders, named Shortest Timing Operation Order Set ($STOOS$). Finally, in the last optimization level, a Resource Optimization algorithm is proposed to find the Best Order (BO) in each $STOOS$. This is to achieve the largest Common Operation Regions (COR) among all the minimized MACRs. By utilizing these CORs, this resource optimization algorithm can minimize the number of additions for all the multiplications, i.e., for the entire scalar-multiplication-containing algorithms. A case study of the optimization method (Figure 6.3) is conducted to prove its advantages. In particular, by applying the optimization method to the Partial Butterfly algorithms, we develop a series of novel hardware-oriented integer transform algorithms for HEVC. Compared to the original Partial Butterfly or conventional multiplication-free Partial Butterfly algorithms, the novel algorithms are much simpler, require shorter running time and use less resource.

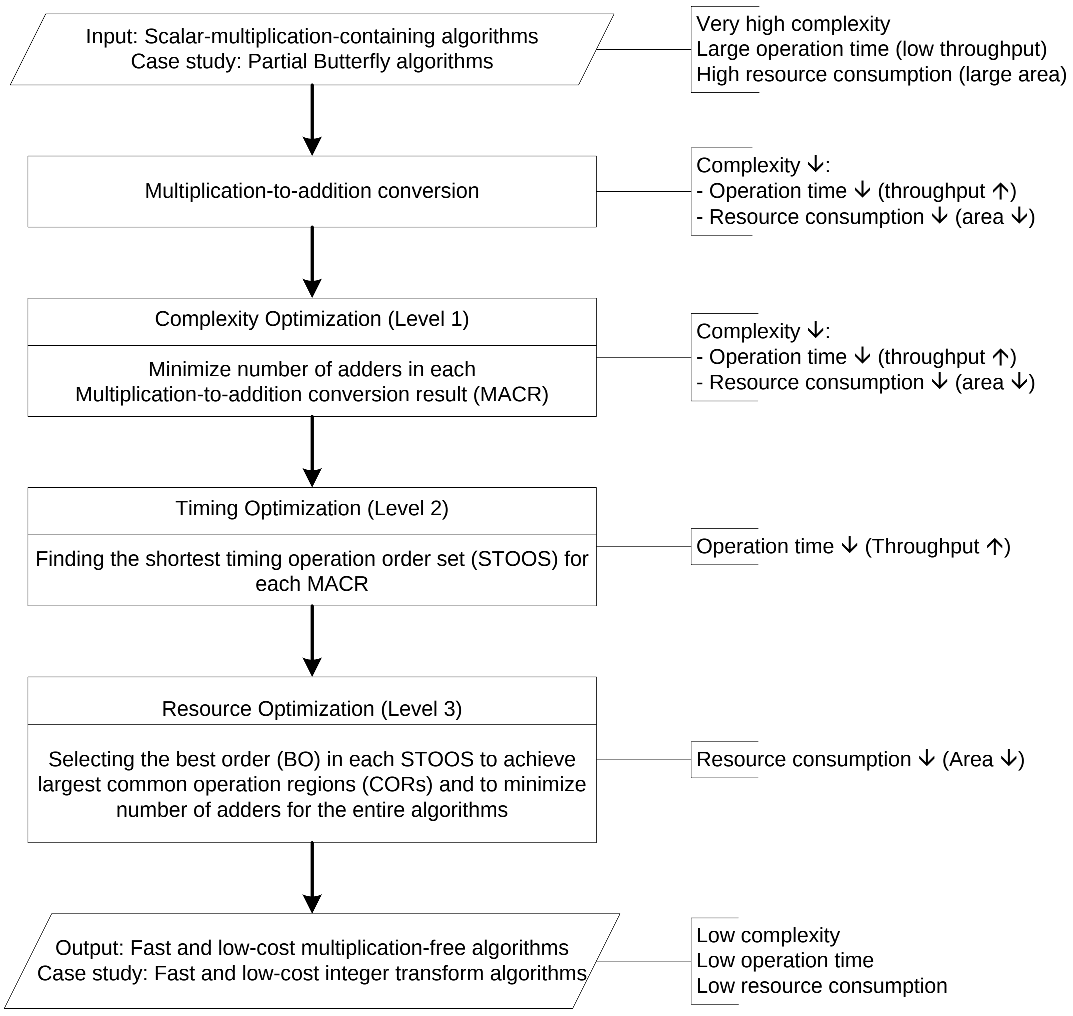

Input: Scalar-multiplication-containing algorithms
Case study: Partial Butterfly algorithms
Very high complexity
Large operation time (low throughput)
High resource consumption (large area)
Multiplication-to-addition conversion
Complexity ↓:
- Operation time ↓ (throughput ↑)
- Resource consumption ↓ (area ↓)
Complexity Optimization (Level 1)
Minimize number of adders in each Multiplication-to-addition conversion result (MACR)
Complexity ↓:
- Operation time ↓ (throughput ↑)
- Resource consumption ↓ (area ↓)
Timing Optimization (Level 2)
Finding the shortest timing operation order set (STOOS) for each MACR
Operation time ↓ (Throughput ↑)
Resource Optimization (Level 3)
Selecting the best order (BO) in each STOOS to achieve largest common operation regions (CORs) and to minimize number of adders for the entire algorithms
Resource consumption ↓ (Area ↓)
Output: Fast and low-cost multiplication-free algorithms
Case study: Fast and low-cost integer transform algorithms
Low complexity
Low operation time
Low resource consumption

FIGURE 6.3: The proposed optimization method for scalar-multiplication-containing algorithms.

6.3.1 A Novel Optimization Method

Figure 6.3 shows the steps and algorithms in the proposed method to reduce complexity, running time and resource consumption of the scalar-multiplication-containing algorithms. These steps and algorithms are sequently described in detail in Section 6.3.1.1, Section 6.3.1.2, Section 6.3.1.3 and Section 6.3.1.4.

Table 6.3: Examples of general multiplication addition conversions.

Multiplications	Binary Representations	General Multiplication Addition Conversion
$x \times 64$	100 0000	$x \ll 6$
$x \times 83$	101 0011	$x + (x \ll 1) + (x \ll 4) + (x \ll 6)$
$x \times 36$	10 0100	$(x \ll 2) + (x \ll 5)$
$x \times 18$	1 0010	$(x \ll 1) + (x \ll 4)$
$x \times 50$	11 0010	$(x \ll 1) + (x \ll 4) + (x \ll 5)$
$x \times 75$	100 1011	$x + (x \ll 1) + (x \ll 3) + (x \ll 6)$
$x \times 89$	101 1001	$x + (x \ll 3) + (x \ll 4) + (x \ll 6)$

6.3.1.1 General Multiplication-to-Addition Conversion

In general, a multiplication can be converted to additions and shift operations by decomposing the multiplier into a sum of powers of 2. For example, 18 can be decomposed as $18 = 16 + 2 = 1 \ll 4 + 1 \ll 1$. Therefore, a multiplication by 18 can be decomposed as $x \times 18 = x \ll 4 + x \ll 1$. More examples can be found in Table 6.3.

6.3.1.2 Proposed Complexity Optimization Algorithm

In hardware implementation, it is possible to design a component to perform both addition and subtraction functions with a negligible increment in area compared to a single adder or to a single subtractor. This adder/subtractor is commonly used in hardware design and FPGA since using a combined adder/subtractor components can save resources and is much more convenient than using two separate components.

For the input scalar-multiplication-containing algorithm types of this optimization method (Figure 6.3), it is also better to consider converting a multiplication to Addition/Subtractions and shift operations (ASs) instead of converting to addition and shift operations only. It is not only because of the above advantages but also because when using ASs, a multiplication may be converted in different ways and the least complex result should be selected for implementation. If only additions and shift operations are used, there is only one conversion result. For example, a multiplication by 7 ($Ob111$) can be converted to ASs as $x \times 7 = x \ll 2 + x \ll 1 + x$, or as $x \times 7 = x \ll 3 - x$. Clearly, the latter is less complex than the former. As complexity affects running time and resource consumption, and consequently, affects the throughput and area of the final system hardware, it is essential to find the least complex Multiplication-to-Addition Conversion Result (MACR) for each multiplication component in the input algorithms to ensure a high throughput and effective area design.

Assuming that we have a multiplication by B to be converted to ASs. B can be represented in binary as:

$$B = b_{N-1}b_{N-2}...b_1b_0, \text{ with } b_i = 0, 1, \tag{6.1}$$

where N is the number of bits needed to represent B:

$$N = \lceil \log_2 B \rceil . \tag{6.2}$$

Putting all the bits in the binary representation of B into a Bit Array, BA, we have

$$\begin{aligned} BA = [b_i], \quad \text{where } i = 0 \rightarrow N-1,\\ b_i = 0, 1. \end{aligned} \tag{6.3}$$

For each conversion result, MACR, by generating a Conversion Array, CA, as an $N+1$ element array of $\{0, 1, -1\}$,

$$\begin{aligned} CA = [c_i], &\quad \text{where } i = 0 \to N \\ &c_i = 0, 1, -1, \end{aligned} \tag{6.4}$$

we have the corresponding conversion of B:

$$B = \sum_{i=0}^{N} c_i(1 \ll i), \tag{6.5}$$

and the corresponding MACR:

$$x \times B = \sum_{i=0}^{N} c_i(x \ll i). \tag{6.6}$$

Taking $B = 7$ as an example, we have $N = 3$ and $BA = [1, 1, 1]$. After that,

$$\begin{array}{lll} \text{if we convert} & B & = 1 \ll 2 + 1 \ll 1 + 1 \ll 0, \\ & x \times B & = x \ll 2 + x \ll 1 + x \ll 0, \\ \text{then we have} & CA & = [0, 1, 1, 1]; \\ \text{if we convert} & B & = 1 \ll 3 - 1 \ll 0, \\ & x \times B & = x \ll 3 - x \ll 0, \\ \text{then we have} & CA & = [1, 0, 0, -1]. \end{array}$$

CA has one more element compared to BA array, because it reserves this element for the conversion to subtraction at bit $N-1$ of B. This is illustrated in the second conversion of the example above.

If we use only additions and shift operations for conversion, CA is the same as BA with element N equal to 0:

$$CA = [0, BA]. \tag{6.7}$$

The corresponding MACR is

$$B = \sum_{i=0}^{N} c_i(1 \ll i) = \sum_{i=0}^{N-1} b_i(1 \ll i). \tag{6.8}$$

$$x \times B = \sum_{i=0}^{N} c_i(x \ll i) = \sum_{i=0}^{N-1} b_i(x \ll i). \tag{6.9}$$

As can be seen, if b_i is equal to 0, the corresponding power-of-2 components, $1 \ll i$ and $x \ll i$, are omitted in the sums in Equation (6.8) and Equation (6.9), respectively. If there is any chain containing consecutive $'1'$ elements in array $BA = [b_i]$, $i = 0 \rightarrow N - 1$, that is

$$\begin{aligned} b_i = 1, \text{ where } i = k \rightarrow k + p; \\ k = 0 \rightarrow N - 1; \\ \text{and } 0 \leq p \leq N - k - 1, \end{aligned}$$

then we can replace:

$$\sum_{i=k}^{k+p} b_i \ll i = 1 \ll (k + p + 1) - 1 \ll k. \tag{6.10}$$

Therefore, if the combined ASs are used for multiplication conversion, instead of using p ASs, i.e., additions, we need to use only one AS, i.e., subtraction. This replacement will have advantages when p is greater than 1. The number of ASs reduced is then equal to $p-1$. When p is equal to 1, or the $'1'$ chain only includes two elements, which is corresponding to one addition, there is no AS reduction when replacing this addition by a subtraction. However, we may gain benefits at the next $'1'$ chain if it starts at position $k+p+2$, i.e., one position away from the current $'1'$ chain, due to the appearance of $'1'$ element at the position $k+p+1$ after the replacement. If p is equal to 0, the replacement of zero addition by a subtraction actually increase the number of ASs used for the current $'1'$ chain. Even if the next $'1'$ chain starts at position $k+p+2$, the present of $'1'$ at position $k+p+1$ only can reduce one more AS in the next chain replacement. Hence, in total, subtraction replacement when p is equal to 0 does not benefit the optimization process.

Based on the above analysis, a complexity optimization algorithm (Figure 6.4, Algorithm 2) is proposed to find all possibilities to reduce the number of ASs needed for a multiplication implementation by effectively replacing each group of additions by a subtraction. The input to the algorithm is the multiplier, while the output is Conversion Array, CA. The final conversion result, MACR (Equation (6.6)), is optimal because the algorithm optimizes one by one $'1'$ element chain and guarantee that the optimization at the current chain does not affect the optimization process for the next chain. Some results, i.e., the final Conversion Array CA, when applying the proposed Complexity Optimization Algorithm to several multiplications are shown in Table 6.4.

Algorithm 1 Complexity optimization algorithm for MAC (unoptimized version).

```
Require: An integer $B > 0$ (Input: Multiplier)
Require: An integer $N \geq 0$ (Number of bits representing $B$)
Require: A $N$-element array BA of 0, 1 (Bits representing $B$)
Require: A $N + 1$-element array CA of 0, 1, -1 (Conversion Array)
Require: Two integer arrays $CS$, $CL$ (The least significant bits and Lengths of
    $'1'/'0'$ chains in $BA$. If position 0 exists a $'0'$ chain, it is omitted in both arrays.
    Add a $'0'$ chain before the most significant position)
Require: An integer $K \geq 0$ (Number of elements in $CL$) and two integer $i$, $j$
 1: Initialize $N \leftarrow \lceil log_2 B \rceil$, $BA$, $CA \leftarrow [0, BA]$, $CS$, $CL$, $K$, $i \leftarrow 0$
 2: while $i < K$ do                                                 ▷ chain $i$ ('1')
 3:     if $CL[i] = 1$ then
 4:         $i \leftarrow i + 2$                                      ▷ Go to the next '1' chain
 5:     end if
 6:     if $CL[i] = 2$ then
 7:         if $CL[i + 1] = 1$ then
 8:             $CA[CS[i]] \leftarrow -1$                             ▷ Convert
 9:             $CA[CS[i] + CL[i]] \leftarrow 1$
10:             for $j = CS[i] + 1 \to CS[i] + CL[i] - 1$ do
11:                 $CA[j] \leftarrow 0$
12:             end for
13:             $CS[i + 2] \leftarrow CS[i + 2] - 1$                  ▷ Update next chain's parameters
14:             $CL[i + 2] \leftarrow CL[i + 2] + 1$
15:             $i \leftarrow i + 2$                                  ▷ Go to the next '1' chain
16:         else
17:             $i \leftarrow i + 2$                                  ▷ Go to the next '1' chain
18:         end if
19:     end if
20:     if $CL[i] > 2$ then
21:         $CA[CS[i]] \leftarrow -1$                                 ▷ Convert
22:         $CA[CS[i] + CL[i]] \leftarrow 1$
23:         for $j = CS[i] + 1 \to CS[i] + CL[i] - 1$ do
24:             $CA[j] \leftarrow 0$
25:         end for
26:         if $CL[i + 1] = 1$ then
27:             $CS[i + 2] \leftarrow CS[i + 2] - 1$                  ▷ Update next chain's parameters
28:             $CL[i + 2] \leftarrow CL[i + 2] + 1$
29:             $i \leftarrow i + 2$                                  ▷ Go to the next '1' chain
30:         else
31:             $i \leftarrow i + 2$                                  ▷ Go to the next '1' chain
32:         end if
33:     end if
34: end whilereturn $CA$
```

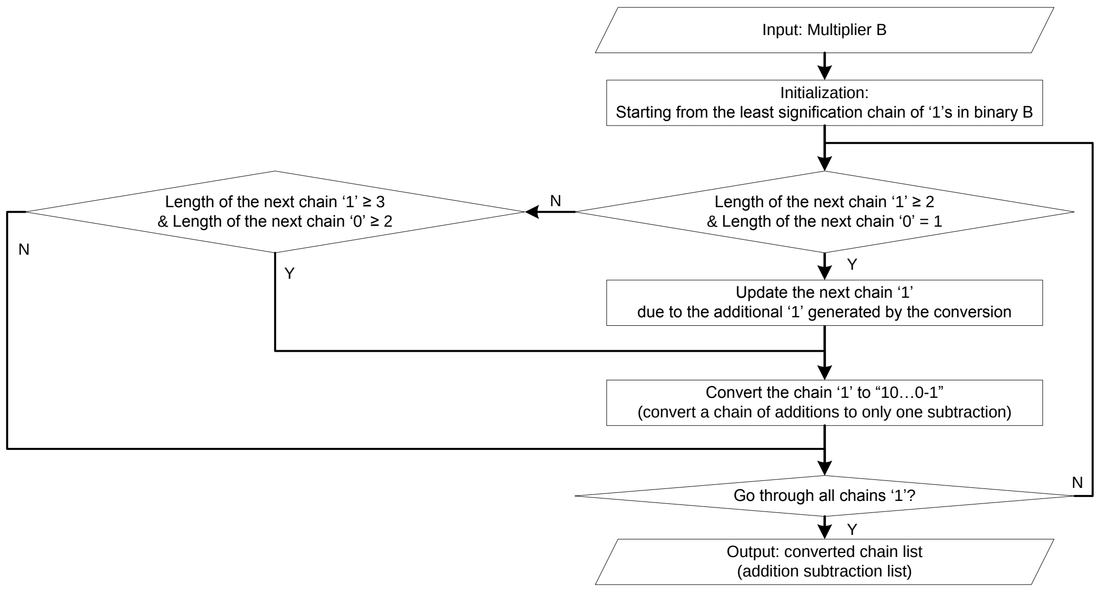

Input: Multiplier B
Initialization:
Starting from the least signification chain of '1's in binary B
Length of the next chain '1' ≥ 2
& Length of the next chain '0' = 1
N
Length of the next chain '1' ≥ 3
& Length of the next chain '0' ≥ 2
N
Y
Y
Update the next chain '1'
due to the additional '1' generated by the conversion
Convert the chain '1' to "10…0-1"
(convert a chain of additions to only one subtraction)
Go through all chains '1'?
N
Y
Output: converted chain list
(addition subtraction list)

FIGURE 6.4: The proposed complexity optimization algorithm for for MAC.

TABLE 6.4: Results when applying complexity optimization algorithm (Algorithm 2) to several multiplications.

B	BA	CA	No. ASs saved
54	[1, 1, 0, 1, 1, 0]	[1, 0, 0, -1, 0, -1, 0]	1
55	[1, 1, 0, 1, 1, 1]	[1, 0, 0, -1, 0, 0, -1]	2
438	[1, 1, 0, 1, 1, 0, 1, 1, 0]	[1, 0, 0, -1, 0, 0, -1, 0, -1, 0]	2
1910	[1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0]	[1, 0, 0, 0, -1, 0, 0, 0, -1, 0, -1, 0]	4
3276	[1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0]	[1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0]	0
214	[1, 1, 0, 1, 0, 1, 1, 0]	[1, 0, 0, -1, 0, -1, 0, -1, 0]	1
86	[1, 0, 1, 0, 1, 1, 0]	[1, 1, 0, -1, 0, -1, 0]	0

6.3.1.3 Proposed Timing Optimization Strategy

After the first level of optimization in Chapter 6.3.1.2, a multiplication is converted to a subtraction. The minuend of the subtraction is a sum of the power-of-2 components corresponding to $'1'$ elements in CA. The subtrahend of the subtraction is the power-of-2 components corresponding to $'-1'$ elements. We denote the number

Algorithm 2 Complexity optimization algorithm for MAC (optimized version).

```
Require: An integer $B > 0$ (Input: Multiplier)
Require: An integer $N \geq 0$ (Number of bits representing $B$)
Require: A $N$-element array BA of 0, 1 (Bits representing $B$)
Require: A $N+1$-element array CA of 0, 1, -1 (Output: Conversion Array)
Require: Two integer arrays $CS$, $CL$ (The least significant bits and Lengths of
    $'1'/'0'$ chains in $BA$. If position 0 exists a $'0'$ chain, it is omitted in both arrays.
    Add a $'0'$ chain before the most significant position)
Require: An integer $K \geq 0$ (Number of elements in CL) and two integer $i$, $j$
 1: function CONVERT($i$)                ▷ Convert an Addition Chain to Subtraction
 2:     $CA[CS[i]] \leftarrow -1$                                        ▷ Lsb
 3:     $CA[CS[i] + CL[i]] \leftarrow 1$                                 ▷ Msb
 4:     for $j = CS[i] + 1 \rightarrow CS[i] + CL[i] - 1$ do
 5:         $CA[j] \leftarrow 0$                                         ▷ Middle bits
 6:     end for
 7: end function
 8: function UPDATE($i$)          ▷ Merge $'1'$ from previous conversion to the chain
 9:     $CS[i] \leftarrow CS[i] - 1$
10:     $CL[i] \leftarrow CL[i] + 1$
11: end function
12: Initialize $N \leftarrow \lceil log_2 B \rceil$, $BA$, $CA \leftarrow [0, BA]$, $CS$, $CL$, $K$, $i \leftarrow 0$
13: while $i < K$ do                                                    ▷ chain $i$ ($'1'$)
14:     if $(CL[i] \geq 2)$ & $(CL[i+1] = 1)$ then                 ▷ chain $'1'$ and chain $'0'$
15:         Convert($i$)
16:         Update($i + 2$)                                         ▷ Update next $'1'$ chain
17:     end if
18:     if $(CL[i] \geq 3)$ & $(CL[i+1] \geq 2)$ then
19:         Convert($i$)
20:     end if
21:     $i \leftarrow i + 2$                                          ▷ Go to the next $'1'$ chain
22: end whilereturn $CA$
```

of non-zero elements in CA as N_I. We assume the number of Addition/Subtractions (ASs), N_A, is needed to perform the multiplication. It can be computed as in Equation (6.11).

$$N_A = N_I - 1. \tag{6.11}$$

Because scalar-multiplication-containing algorithms (Figure 6.3) are to be implemented in hardware, their operations can be performed in parallel. Therefore,

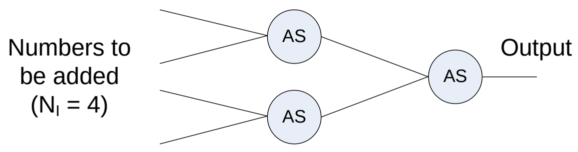


FIGURE 6.5: Optimized adding tree with $N_I = 4$. Addition depth $AD = 2$. Running time $RT = 2 \times RT_A$.

different orders of their operations can lead to different running-times. If the operations are performed in sequence, the running-time is

$$RT = N_A \times RT_A, \tag{6.12}$$

where RT_A is the running-time of an AS. However, if ASs are allowed to run in parallel, the shortest running time can be achieved when we add all the numbers in pairs at one time. Next, we add all the outputs of the previous ASs in pairs. This step is repeated until there is only one output left. Since this strategy utilizes addition parallelism as much as possible, the running time is the optimal value.

We define Addition Depth, AD, as the largest number among the numbers of ASs required from any inputs to the final output. The relationship between the running time of the multiplication and Addition Depth is

$$RT = AD \times RT_A. \tag{6.13}$$

The optimized Addition Depth, AD_o, which is achieved by the proposed strategy, is

$$AD_o = \lceil \log_2 N_I \rceil . \tag{6.14}$$

Figure 6.5 and Figure 6.6 illustrate the adding tree generated for two examples when N_I is equal to 4 and 5, respectively.

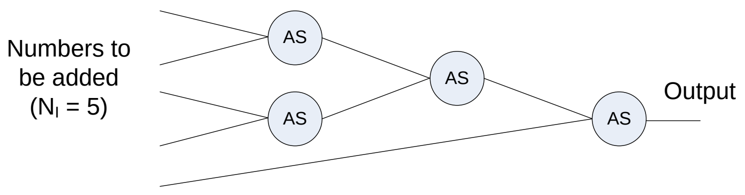


FIGURE 6.6: Optimized adding tree with $N_I = 5$. Addition depth $AD = 3$. Running time $RT = 3 \times RT_A$.

Although the adding trees generated by the proposed timing-optimization strategy leads to the shortest running times, they do not lead to an unique addition order or operation order. When the operands corresponding to non-zero elements in array CA are assigned as the inputs of an optimized adding tree, their different permutations may provide different operation orders. With a permutation, the adding tree may add the component corresponding to the most significant element in CA with the second most together. In another permutation, it may add the most significant one with the least significant component. Different orders may cause different effects on the optimization process. Therefore, some properties of the adding trees generated using the proposed timing-optimization strategy are going to be described in detail for future use.

1. Generated adding trees for N_I inputs have AD levels of ASs, computed using Equation (6.14) (Figure 6.7).

2. If the number of inputs to level j is NIL_j, the number of ASs used in level j, NAL_j, is

$$NAL_j = \left\lfloor \frac{NIL_j}{2} \right\rfloor, \tag{6.15}$$

and the number of inputs left for level $j+1$ is

$$NIL_{j+1} = NIL_j - NAL_j = NIL_j - \left\lfloor \frac{NIL_j}{2} \right\rfloor. \tag{6.16}$$

Algorithm 3 is designed to implement these calculation.

Algorithm 3 Calculation of the number of inputs (NIL) and ASs (NAL) for each level of a multiplication.

```
Require: An integer $NI > 0$ (Input: Number of non-zero elements in array $CA$)
Require: An integer $AD > 0$ (Addition Depth)
Ensure: $AD = \lceil \log_2 N_I \rceil$
Require: Two integer arrays $NIL$, $NAL$
Require: An integer $j$
1: for $j = 1 \rightarrow AD$ do                                  ▷ For each level
2:     if $j = 1$ then
3:         $NIL[j] \leftarrow NI$                                   ▷ Inputs of Level 1
4:     else
5:         $NIL[j] \leftarrow NIL[j-1] - \left\lfloor \frac{NIL[j-1]}{2} \right\rfloor$
6:     end if
7:     $NAL[j] \leftarrow \left\lfloor \frac{NIL[j]}{2} \right\rfloor$
8: end forreturn $NIL$, $NAL$
```

3. Most of the time, the inputs of ASs in level j ($j = 1 \rightarrow AD$) are from the adjacent level $j-1$. However, in some cases, they can be from even lower levels, e.g., Figure 6.7(b), Figure 6.7(d), Figure 6.7(e) and Figure 6.7(f). Particularly, in Figure 6.7(d), the AS at level 3 has two inputs: one is from level 2 and the other is from level 0, i.e., original input.

4. If the situation of taking inputs from non-adjacent levels happens in one level, it only happens at the last AS among the ASs of the level.

5. If the number of ASs at level j (NAL_j) multiplied by 2 is greater than the number of ASs at the lower-adjacent level $j-1$ (NAL_{j-1}), the second input of the last AS at level j is from non-adjacent level. This non-adjacent level is the nearest lower level having the number of ASs greater than 2 times of that of its upper-adjacent level. Table 6.5 lists numbers of ASs in different levels for N_I from 2 to 12, and shows the cases $^{(\star)}$ when ASs take inputs from non-adjacent level.

Table 6.5: Levels with non-adjacent inputs in the proposed adding trees when $N_I = 2 \rightarrow 12$.

N_I	L_1	L_2	L_3	L_4	Levels with Non-Adjacent Input	Input Levels
2	1	0	0	0		
3	1	$1^{(\star)}$	0	0	L_2	L_0
4	2	1	0	0		
5	2	1	$1^{(\star)}$	0	L_3	L_0
6	3	1	$1^{(\star)}$	0	L_3	L_1
7	3	$2^{(\star)}$	1	0	L_2	L_0
8	4	2	1	0		
9	4	2	1	$1^{(\star)}$	L_4	L_0
10	5	2	1	$1^{(\star)}$	L_4	L_1
11	5	$3^{(\star)}$	1	$1^{(\star)}$	L_4, L_2	L_2, L_0
12	6	3	1	$1^{(\star)}$	L_4	L_2

Taking $N_I = 11$ as an example, we can see that level 4 and level 2 have inputs from the non-adjacent levels, which are level 2 and level 0.

As mentioned before, after having the optimized adding tree with the shortest running time for a multiplication, the components corresponding to non-zero elements in array CA are assigned as the inputs of the tree. Different permutations of these components may lead to different operation order, consequently, different effects for the optimization process. Therefore, we need to find all the permutations which lead to different operation orders, named Shortest Timing Operation Order Set ($STOOS$). It should be noted that for any addition, exchanging the order of its two inputs does not lead to any changes in operation order. Hence, we can always assign the component which is corresponding to the more significant bits in CA to the left input of ASs in level 1 compared to the right. This assignment does not affect $STOOS$ at all. Based on this analysis, the number of permutations, S, for a multiplication with N_I non-zero elements in array CA can

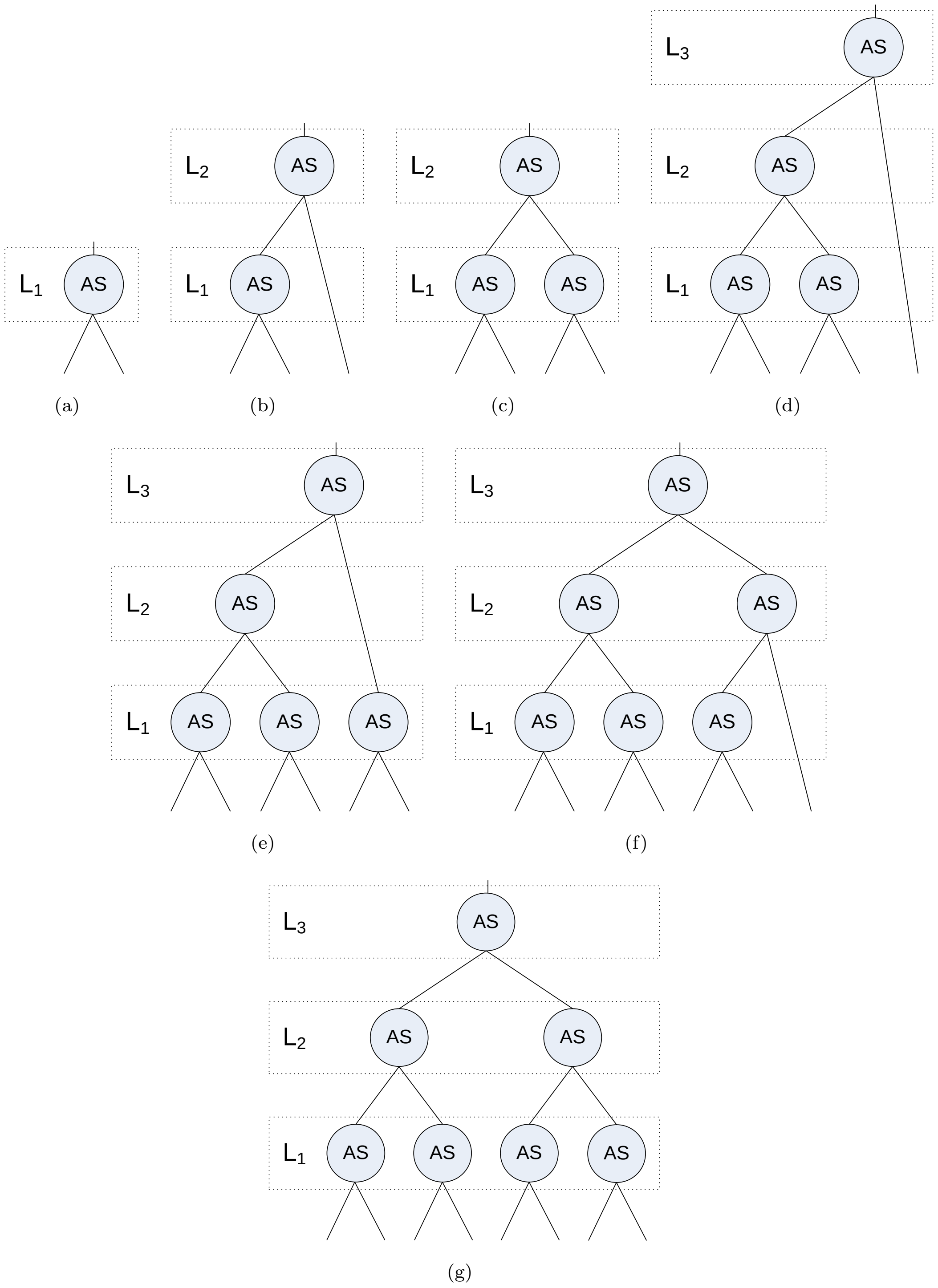


FIGURE 6.7: Levels of optimized adding trees. (a) $N_I = 2$; (b) $N_I = 3$; (c) $N_I = 4$; (d) $N_I = 5$; (e) $N_I = 6$; and (f) $N_I = 7$.

be computed as

$$S = \frac{N_I!}{\lfloor \frac{N_I}{2} \rfloor! \, (2!)^{\lfloor \frac{N_I}{2} \rfloor}}. \tag{6.17}$$

Also based on the above analysis, Algorithm 4 is proposed to generate $STOOS$ of a given multiplication.

Table 6.6 shows the arrays STOSSs generated by Algorithm 4 for several multiplications. As can be seen from row 1 of the table, when $B = 54$, after the first optimization, we have $CA = [1, 0, 0, -1, 0, -1, 0]$. Since CA has 3 non-zero components, we only have three permutations which can lead to different operation orders based on the optimized adding tree (Figure 6.7(b)). In particular, with permutation $[2, 1, 0]$, $x \ll 6$ and $-(x \ll 3)$ are added in level 1 of the adding tree. This sum is added together with $-(x \ll 1)$ in level 2 of the adding tree. It should be noted that permutation $[1, 2, 0]$ leads to the same operation order with that of permutation $[2, 1, 0]$. Therefore, $[1, 2, 0]$ is not included in STOSS. For permutation $[2, 0, 1]$, $x \ll 6$ and $-(x \ll 1)$ are added in level 1 of the adding tree. This sum is added together with $-(x \ll 3)$ in level 2 of the adding tree. For permutation $[1, 0, 2]$, $-x \ll 3$ and $-(x \ll 1)$ are added in level 1 of the adding tree. This sum is added together with $(x \ll 6)$ in level 2 of the adding tree.

6.3.1.4 Proposed Resource Optimization Algorithm

In the novel timing and resource consumption optimization method for scalar-multiplication-containing algorithms, the optimization level 1 minimize the number of ASs used for each multiplication. The inputs of this optimization level are multipliers of all multiplications in scalar multiplications, while its output is the least complex Multiplication Addtion Conversion Results (MACRs) of all the multiplications. Each MACR corresponding to a multiplication is represented in

Algorithm 4 STOOS generation for a multiplication.

```
Require: An integer $NI > 0$ (Input: Number of non-zero elements in array $CA$)
Require: An integer $S > 0$ (Number of permutations in STOOS)
Ensure: $S = \frac{N_I!}{\lfloor \frac{N_I}{2} \rfloor ! (2!)^{\lfloor \frac{N_I}{2} \rfloor}}$
Require: One $N_I$-element binary array $STATUS$
Require: One $N_I$-element integer array $LABEL$
Require: One integer array $STOOS[S, N_I]$

 1: function SELECT($j$)                         ▷ Select 2 components for 2 inputs of AS $j$
 2:     for $k = N_I - 1 \to 1$ do                                      ▷ Msb first
 3:         if $STATUS[k] = 0$ then                                 ▷ Still available
 4:             $STATUS[k] \leftarrow 1$                        ▷ Select $k$ for the left input
 5:             $LABEL[2 \times j] \leftarrow k$
 6:             for $l = k - 1 \to 0$ do                    ▷ less significant than $k$
 7:                 if $STATUS[l] = 0$ then
 8:                     $STATUS[l] \leftarrow 1$              ▷ Select $l$ for the right input
 9:                     $LABEL[2 \times j + 1] \leftarrow l$
10:                     if $j + 1 < \lfloor \frac{N_I}{2} \rfloor$ then
11:                         SELECT($j + 1$)
12:                     else                                   ▷ the last $AS$ of level 1
13:                         if $N_I \mod 2 = 0$ then
14:                             Store $LABLE$ into STOOS array
15:                         else
16:                             for $t = N_I - 1 \leftarrow 0$ do
17:                                 if $STATUS[t] = 0$ then
18:                                     $LABEL[2 \times j + 2] \leftarrow t$
19:                                 end if
20:                             end for
21:                         end if
22:                     end if
23:                     $STATUS[l] \leftarrow 0$                     ▷ Re-assign $l$ status
24:                 end if
25:             end for
26:             $STATUS[k] \leftarrow 0$                             ▷ Re-assign $k$ status
27:         end if
28:     end for
29: end function

30: Initialize $STATUS[i]$, $i = 0 \to N_I - 1$; Empty $STOOS$
31: SELECT(0) return $STOOS$
```

TABLE 6.6: Generated STOSSs using Algorithm 4 for several multiplications.

B	CA	N_I	$STOOS[0 \to S-1, 0 \to N_I - 1]$
54	[1, 0, 0, -1, 0, -1, 0]	3	[2, 1, 0; 2, 0, 1; 1, 0, 2]
55	[1, 0, 0, -1, 0, 0, -1]	3	[2, 1, 0; 2, 0, 1; 1, 0, 2]
438	[1, 0, 0, -1, 0, 0, -1, 0, -1, 0]	4	[3, 2, 1, 0; 3, 1, 2, 0; 3, 0, 2, 1]
1910	[1, 0, 0, 0, -1, 0, 0, 0, -1, 0, -1, 0]	4	[3, 2, 1, 0; 3, 1, 2, 0; 3, 0, 2, 1]
3276	[1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0]	6	[5, 4, 3, 2, 1, 0; 5, 4, 3, 1, 2, 0; 5, 4, 3, 0, 2, 1; 5, 3, 4, 2, 1, 0; 5, 3, 4, 1, 2, 0; 5, 3, 4, 0, 2, 1; 5, 2, 4, 3, 1, 0; 5, 2, 4, 1, 3, 0; 5, 2, 4, 0, 3, 1; 5, 1, 4, 3, 2, 0; 5, 1, 4, 2, 3, 0; 5, 1, 4, 0, 3, 2; 5, 0, 4, 3, 2, 1; 5, 0, 4, 2, 3, 1; 5, 0, 4, 1, 3, 2]
214	[1, 0, 0, -1, 0, -1, 0, -1, 0]	4	[3, 2, 1, 0; 3, 1, 2, 0; 3, 0, 2, 1]
86	[1, 1, 0, -1, 0, -1, 0]	3	[2, 1, 0; 2, 0, 1; 1, 0, 2]

an array CA storing values $'0'$, $'1'$ and $'-1'$ and can be computed using Equation (6.6). It should be noted that each non-zero element in CA is corresponding to a component to be added or subtracted in the optimized MACR.

Optimization level 2 is based on the fact that ASs can be performed in parallel since scalar-multiplication-containing algorithms will be implemented in hardware with aim of achieving high throughputs with area efficiencies. In this level of optimization, for each multiplication, running time is optimized and the shortest adding tree is selected. Then, the Shortest Timing Operation Order Set (STOOS) is generated for each multiplication storing permutations of operands in the minimized MACR of the multiplication. Each permutation in STOOS leads to different operation orders based on the optimized adding trees. The input of the second optimization level is arrays CAs of multiplications from the first optimization level, and its output is the shortest adding tree and array STOOS corresponding to each multiplication.

After minimizing the complexity in optimization level 1 and minimizing the running time for each multiplication in level 2, in optimization level 3, Common Operation Regions (CORs) among all multiplications will be investigated based on STOOS and the adding tree for each multiplication. If the minimized MACRs of two multiplications with specific operation orders has a COR, the operations in the COR can be shared for both multiplications. Therefore, Saved Resource (SR) is all operations in the COR. If n multiplications share a COR, SR is $n-1$ times of all operations in the COR. Different operation orders of the minimized MACR for each multiplication may have different CORs, and consequently, lead to different SRs. Hence, the maximum SR is searched in all multiplication with all operation orders. By sharing the operations of the CORs corresponding to the maximum SR, resource consumption can be minimized. The input of the optimization level 3 is arrays CAs, arrays STOOS and the optimized adding trees for multiplications. Its output is the BO of in STOOS, maximum SR and largest CORs.

In this optimization problem, the objective function SR can be computed as the number of reducible ASs. An AS can be an addition/subtractions of maximum 2^l inputs where l is the level of the AS and inputs are power-of-2s corresponding to the non-zero elements in CA. The resource optimization method is to determine the largest common operation regions (CORs). Two ASs are called "common" or replaceable when (1) they are at the same level in the adding trees; (2) the distances from other elements to the lowest element added by the two ASs are the same; and (3) all signs of their elements are the same in pairs or are opposite in pairs. If we simply compare the positions and signs of the elements, we will miss many common ASs. COR includes common ASs.

In order to automatically search for the maximum SR in all the STOOS space of the multiplications, adding tree data representation should support the followings:

1. The representation needs to facilitate the search within one level. It is because an AS at one level is only comparable to other AS at the same level.

2. From a particular AS representation, it is possible to find the information of its children.

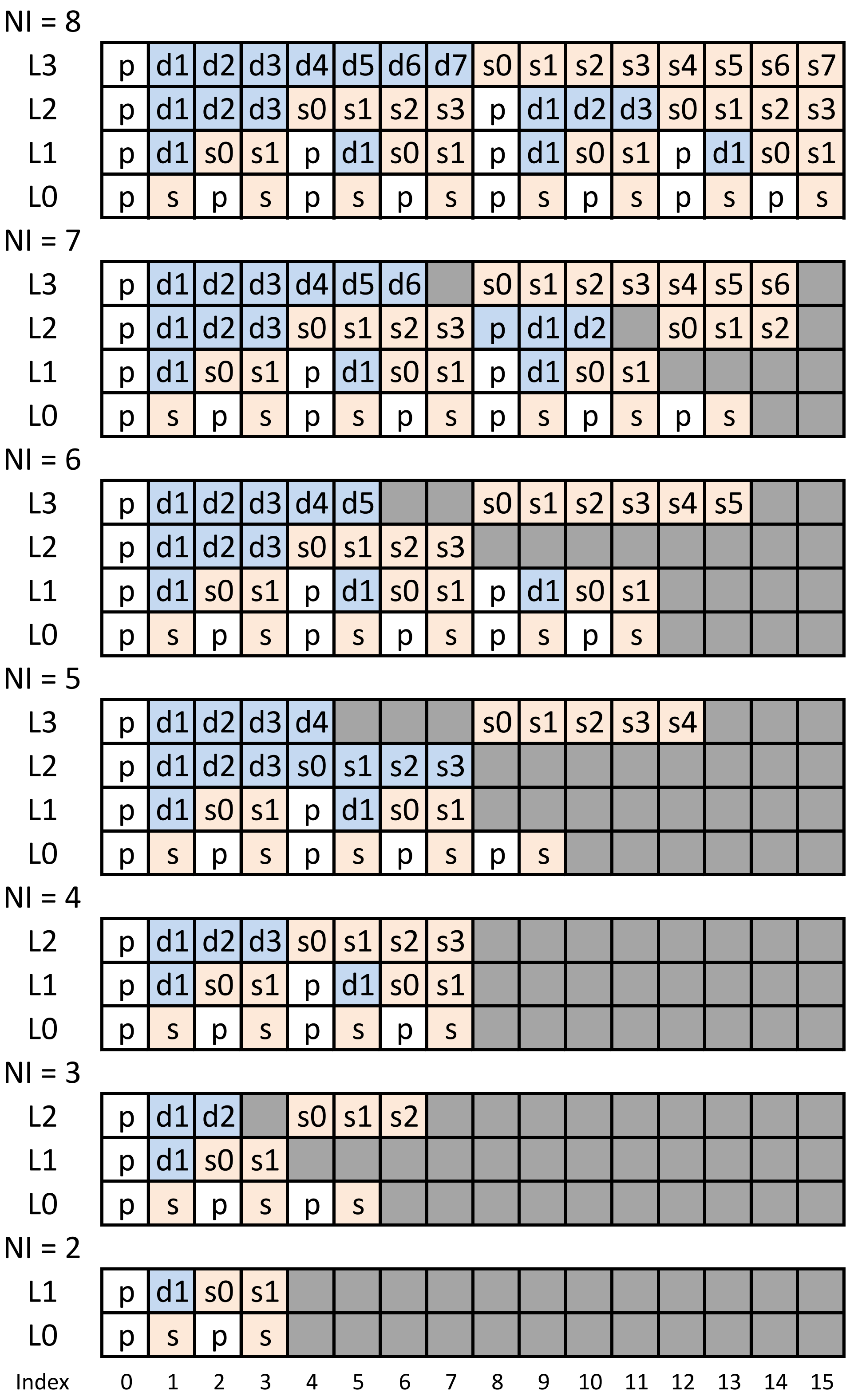
3. The representation of an AS needs to show the least signification (lowest) element position, distances from other elements to the least significant element, and signs of all elements.

Figure 6.8 illustrates the proposed data structure for the optimized adding tree for $N_I = 2 \rightarrow 8$.

The size of data, $T[m]$, $m = 0 \rightarrow M - 1$, for representing all ASs in level m of multiplication is

$$T = 2^{\lceil \log_2 NI[m] \rceil + 1}. \tag{6.18}$$

Given the multiplications by M numbers: $B[m]$, $m = 0 \rightarrow M - 1$, after the first optimization level, we have Conversion Array $CA[M, Nx + 1]$, where Nx is the maximum number of bits representing $B[m]$ (Equation (6.2)), and $CA[m, nx]$, $nx = 0 \rightarrow Nx$, is for each multiplication by $B[m]$. We generate array $P[M, NI[M]]$ to store the positions of non-zero elements in CAs, where $P[m, n]$ ($m = 0 \rightarrow M - 1$, $n = 0 \rightarrow NI[m] - 1$) is the position of n^{th} non-zero elements in $CA[m]$. In the second optimization level, we generate $STOOS[M, S[M], NI[M]]$, where $STOOS[m, s, n]$, $n = 0 \rightarrow NI - 1$ is the s^{th} permutation of STOOS corresponding to multiplication by $B[m]$. Hence, $P[m, STOOS[m, s, n]]$, $n = 0 \rightarrow NI - 1$ is s^{th} permutation of operands corresponding to the minimized MACR of the multiplication by $B[m]$.



NI = 8																
L3	p	d1	d2	d3	d4	d5	d6	d7	s0	s1	s2	s3	s4	s5	s6	s7
L2	p	d1	d2	d3	s0	s1	s2	s3	p	d1	d2	d3	s0	s1	s2	s3
L1	p	d1	s0	s1	p	d1	s0	s1	p	d1	s0	s1	p	d1	s0	s1
L0	p	s	p	s	p	s	p	s	p	s	p	s	p	s	p	s
NI = 7																
L3	p	d1	d2	d3	d4	d5	d6	▨	s0	s1	s2	s3	s4	s5	s6	▨
L2	p	d1	d2	d3	s0	s1	s2	s3	p	d1	d2	▨	s0	s1	s2	▨
L1	p	d1	s0	s1	p	d1	s0	s1	p	d1	s0	s1	▨	▨	▨	▨
L0	p	s	p	s	p	s	p	s	p	s	p	s	p	s	▨	▨
NI = 6																
L3	p	d1	d2	d3	d4	d5	▨	▨	s0	s1	s2	s3	s4	s5	▨	▨
L2	p	d1	d2	d3	s0	s1	s2	s3	▨	▨	▨	▨	▨	▨	▨	▨
L1	p	d1	s0	s1	p	d1	s0	s1	p	d1	s0	s1	▨	▨	▨	▨
L0	p	s	p	s	p	s	p	s	p	s	p	s	▨	▨	▨	▨
NI = 5																
L3	p	d1	d2	d3	d4	▨	▨	▨	s0	s1	s2	s3	s4	▨	▨	▨
L2	p	d1	d2	d3	s0	s1	s2	s3	▨	▨	▨	▨	▨	▨	▨	▨
L1	p	d1	s0	s1	p	d1	s0	s1	▨	▨	▨	▨	▨	▨	▨	▨
L0	p	s	p	s	p	s	p	s	p	s	▨	▨	▨	▨	▨	▨
NI = 4																
L2	p	d1	d2	d3	s0	s1	s2	s3	▨	▨	▨	▨	▨	▨	▨	▨
L1	p	d1	s0	s1	p	d1	s0	s1	▨	▨	▨	▨	▨	▨	▨	▨
L0	p	s	p	s	p	s	p	s	▨	▨	▨	▨	▨	▨	▨	▨
NI = 3																
L2	p	d1	d2	▨	s0	s1	s2	▨	▨	▨	▨	▨	▨	▨	▨	▨
L1	p	d1	s0	s1	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨
L0	p	s	p	s	p	s	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨
NI = 2																
L1	p	d1	s0	s1	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨
L0	p	s	p	s	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

FIGURE 6.8: The proposed data structure for the optimized adding trees for $N_I = 2 \rightarrow 8$. p, d and s stand for position, distance and sign, respectively.

Algorithm 5 is to generate data for all the adding trees of all M given multiplications. Its output is the adding tree data, $L[M, AD[M], S[M], T[M]]$, where M is the number of multiplication; $AD[m]$ is Addition Depth of multiplications m (Equation (6.14)); $S[m]$ is number of permutations in $STOOS$ of multiplication m (Equation (6.17)); and $T[m]$ is the size of data to represent all ASs in each level of multiplication m (Equation (6.18)).

After adding tree data are generated, Algorithm 8 is applied to search for all BTOOS space to find the COR having the highest SR.

Utilizing all the common ASs as labeled in the algorithm, we can optimize the scalar-multiplication containing algorithms to become hardware-oriented multiplication-free algorithms with the shortest running time and least resource consumption.

6.3.2 Proposed Fast and Low-Cost Transform Algorithms

The Partial Butterfly algorithms presented in HEVC test Model HM7.0 are the algorithms to perform the integer transforms for HEVC with different sizes of 4×4, 8×8, 16×16 and 32×32. Figure 6.1 and Figure 6.2 illustrates the 4×4 and 8×8 Partial Butterfly algorithms, respectively. As can be seen, they contain scalar multiplications. In particular, the 4×4 algorithms contain multiplications of $[83, 36]$, while the 8×8 algorithms contain multiplications of $[83, 36]$ and $[18, 50, 75, 89]$. Since the Partial Butterfly algorithms are urgently needed to be implemented in hardware with high throughput and area-efficient requirement, they satisfy the conditions for being applied to the proposed Optimization Method to reduce complexity and facilitate high throughput and area-efficient hardware implementation.

Algorithm 5 Data generation for all adding trees of all M given multiplications.

```
Require: An integer $M > 0$ (Input: Number of Multiplication)
Require: An integer array $B[M] > 0$ (Input: Multipliers)
Require: An integer $N$ (Max no. of bits representing $B[m]$, $m = 0 \rightarrow M-1$)
Require: An integer array $CA[M, N+1]$ (Conversion Array)
Require: An integer array $NI[M] > 0$ (No. of non-zero elements in array $CA$)
Require: An integer array $PS[M, NI[M]]$ (Positions of non-zero $CA$ elements)
Require: An integer array $AD[M] > 0$ (Addition Depth of multiplications)
Require: An integer $ADmax$
Require: Two integer arrays $NIL[M, AD[M]]$, $NAL[M, AD[M]]$
    (Number of Inputs and Adders for each level of each multiplication)
Require: Integer $i$, $j$, $k$, $l$, $m$, $s$
Require: An integer array $S[M] > 0$ (No. of permutations in $STOOS$ of mul.)
Require: An integer array $STOOS[M, S[M], NI[M]]$
Require: An integer array $T[M] > 0$ (size of $L[M, AD[M], S[M])$
Require: An integer array $L[M, AD[M], S[M], T[M]]$
    (Data for each multiplication, level, permutation)

 1: Compute $N = \max\{\lceil \log_2 B[m] \rceil)\}$, $m = 0 \rightarrow M-1$ (Equation (6.2))
 2: Compute optimized $CA[M, N+1]$ using Complexity Optimization algorithm
    (Algorithm 2) at the optimization level 1.
 3: Initialize $NI[m]$ = Number of non-zero elements in $CA[m]$, $m = 0 \rightarrow M-1$
 4: Initialize $PS[m, n]$ = Position of non-zero element $n$ in $CA[m]$, $m = 0 \rightarrow M-1$,
    $n = 0 \rightarrow NI[m]-1$
 5: Initialize $AD[m]$ (Equation (6.14)), $ADmax = \max\{AD[m]\}$, $m = 0 \rightarrow M-1$
 6: Initialize $NAL[m, l]$, $NIL[m, l]$, $m = 0 \rightarrow M-1$, $l = 1 \rightarrow AD[m]$ (Equation (6.15)), Equation (6.16) and Algorithm 3)
 7: Initialize $NAL[m, 0] \leftarrow NI[m]$, $m = 0 \rightarrow M-1$
    (to use check adjacent input at level 1)
 8: Initialize $S[m]$ (Equation (6.17)), $m = 0 \rightarrow M-1$
 9: Initialize $STOOS[M, S[M], N_I[M]]$ using Algorithm 4
10: Compute $T[m]$, $m = 0 \rightarrow M-1$ (Equation (6.18))

11: for $m = 0 \rightarrow M-1$ do                          ▷ each multiplication
12:     for $s = 0 \rightarrow S[m]-1$ do               ▷ each permutation in $STOOS$
13:         for $l = 0 \rightarrow AD[m]$ do                            ▷ each level
14:             if $l = 0$ then
15:                 for $t = 0 \rightarrow NI[m]-1$ do
16:                     $L[m, l, s, 2 \times t] \leftarrow PS[m, STOOS[m, s, t]]$
17:                     $L[m, l, s, 2 \times t+1] \leftarrow CA[m, PS[m, STOOS[m, s, t]]]$
18:                 end for
```

Algorithm 6 Data generation for STOOS of multiplications (cont.).

```
19:             else
20:                 if $2 \times NAL[m, l] \leq NAL[m, l-1]$ then        ▷ Adjacent inputs
21:                     for $t = 0 \to NAL[m, l] - 1$ do            ▷ each AS in level $l$
22:                         Take position and sign information of $2^l$ components in
23:                         level $l - 1$ starting from $L[m, l-1, s, 2^{l+1} \times t]$
24:                         Find the smallest position among $2^l$ components and
25:                         store into $L[m, l, s, 2^{l+1} \times j]$
26:                         Re-compute all distances based on the new based position
27:                         Copy signs up
28:                     end for
29:                 else                          ▷ The last AS has non-adjacent input
30:                     for $t = 0 \to NAL[m, l] - 2$ do            ▷ each AS in level $l$
31:                         Take position and sign information of $2^l$ components in
32:                         level $l - 1$ starting from $L[m, l-1, s, 2^{l+1} \times t]$
33:                         Find the smallest position among $2^l$ components and
34:                         store into $L[m, l, s, 2^{l+1} \times j]$
35:                         Re-compute all distances based on the new based position
36:                         Copy signs up
37:                     end for
38:                     for $t = l - 2 \to 0$ do  ▷ Find the non-adjacent input level $ti$
39:                         if $2 \times NAL[m, t+1] < NAL[m, t]$ then
40:                             $ti \leftarrow t$
41:                             Stop For loop
42:                         end if
43:                         $t \leftarrow NAL[m, l] - 1$                        ▷ last AS
44:                         Copy position and sign information of $2^{l-1}$ components
45:                         in level $l - 1$ starting from $L[m, l-1, s, 2^(l+1) \times t]$
46:                         Copy position and sign of the last components in level $ti$
47:                         Find the smallest position among $2^l$ components and store
48:                         into $L[m, l, s, 2^(l+1) \times j]$
49:                         Re-compute all distances based on the new based position
50:                         Copy signs up
51:                     end for
52:                 end if
53:             end if
54:         end for
55:     end for
56: end forreturn $L[M, AD[M], S[M], T[M]]$
```

Algorithm 7 Resource optimization algorithm: Search for all BTOOS space to find the COR having the highest SR.

Require: An integer $M > 0$ (Input: Number of Multiplication)
Require: An integer array $L[M, AD[M], S[M], T[M]]$ (Input: Adding tree data)
Require: Two integer array $SP[M]$, $SPx[M]$ (Selected Permutation for Mul.s)
Require: Two integer SR, SRx (Saved Resource)
Require: An integer ADx (Maximum Addition Depth)
Ensure: $ADx = \max\{AD[m]\}$, $m = 0 \to M - 1$
Require: An integer array $CORAS[M, AD[M], NAL[M, AD[M]]]$
(Common Operation Region label for each AS)
Require: An Integer R (Region ID)
Require: An Boolean CMN (Have common AS or not)

```
 1: function SELECT4MUL($i$)    ▷ Select a permutation in STOOS for mul. $i$
 2:     for $j = S[i] - 1 \to 0$ do
 3:         $SP[i] \leftarrow j$    ▷ Select permutation $j$ for mul. $i$
 4:         if $i < M - 1$ then
 5:             SELECT4MUL($i + 1$)
 6:         else    ▷ Selected permutations for all mul.
 7:             $SR \leftarrow 0$    ▷ Initialize $SR$
 8:             $R \leftarrow 0$    ▷ Initialize $R$
 9:             for $l = ADx - 1 \to 1$ do    ▷ Level
10:                 for $m = 0 \to M - 1$ do    ▷ Mul
11:                     for $t = 0 \to NAL[m, l] - 1$ do    ▷ No. of ASs in a Level
12:                         if $CORAS[m, l, t] = 0$ then    ▷ No COR assigned
13:                             $CMN \leftarrow FALSE$
14:                             $R \leftarrow R + 1$
15:                             for $k = t + 1 \to NAL[m, l] - 1$ do    ▷ in same mul
16:                                 if 2 ASs $[m, l, t]$&$[m, l, t]$ are “common” then
17:                                     $CMN \leftarrow TRUE$
18:                                     $CORAS[m, l, k] = R$
19:                                     Label all childs of $AS[m, l, k]$: $CORAS = R$
20:                                     $SR \leftarrow SR+$ AS[m,l,t] size
21:                                 end if
22:                             end for
23:                             for $u = m + 1 \to M - 1$ do    ▷ Search in other muls
24:                                 for $v = 0 \to NAL[u, l] - 1$ do
25:                                     if 2 ASs $[m, l, t]$&$[u, l, v]$ are “common” then
26:                                         $CMN \leftarrow TRUE$
27:                                         $R \leftarrow R + 1$
28:                                         $CORAS[u, l, v] = R$
29:                                         Label all childs of $AS[u, l, v]$: $CORAS = R$
30:                                         $SR \leftarrow SR+$ AS[m,l,t] size
```

Algorithm 8 Resource optimization algorithm: Search for all BTOOS space to find the COR having the highest SR.

```
31:                                 end if
32:                             end for
33:                         end for
34:                         if $CMN = TRUE$ then
35:                             $CORAS[m, l, t] = R$
36:                             Label all childs of $AS[m, l, t]$: $CORAS = R$
37:                         else
38:                             $R \leftarrow R - 11$
39:                         end if
40:                     end if
41:                 end for
42:             end for
43:         end for
44:         if $SR > SRx$ then                      ▷ New $SP$ is better, → select
45:             $SRx \leftarrow SR$
46:             $SPx \leftarrow SP$
47:             Store $CORAS[M, AD[M], NAL[M, AD[M]]]$
48:         end if
49:     end if
50:   end for
51: end function

52: Initialize $SRx \leftarrow 0$
53: Initialize $CORAS[M, AD[M], NAL[M, AD[M]]] \leftarrow 0$
54: SELECT4MUL(0) return $L[M, AD[M], S[M], T[M]]$
```

When applying the proposed optimization method to the 4×4 and 8×8 Partial Butterfly algorithms, we compute and generate all related data, including B, CA (at optimization level 1 with Complexity Optimization algorithm 2), NI, $STOOS$ (at optimization level 2 by algorithm 4), PS, AD, NIL, NAL (algorithm 3), S and T (algorithm 5) for each multiplication in the scalar multiplications. The computation result can be found in Table 6.7. It should be noted that since the binary representation of the multipliers in the 4×4 and 8×8 Partial Butterfly algorithms are sparse, the CA after being applied the complexity optimization level in the method is the same as before.

TABLE 6.7: Intermediate data during execution of the proposed optimization method for 4×4 and 8×8 Partial Butterfly algorithms.

B	B (0b)	NI	STOOS (index from $0 \rightarrow +$)	PS	AD	NIL	NAL	S	T
83	101 0011	4	[3, 2, 1, 0; 3, 1, 2, 0; 3, 0, 2, 1]	[0, 1, 4, 6]	2	4 / 2	1 /2	3	8
36	10 0100	2	[2, 1, 0]	[2, 5]	1	2	1	1	4
18	1 0010	2	[1, 0]	[1, 4]	1	2	1	1	4
50	11 0010	3	[2, 1, 0; 2, 0, 1; 1, 0, 2]	[1, 4, 5]	2	3 / 2	1 / 1	3	8
75	100 1011	4	[3, 2, 1, 0; 3, 1, 2, 0; 3, 0, 2, 1]	[0, 1, 3, 6]	2	4 / 2	1 / 2	3	8
89	110 1001	4	[3, 2, 1, 0; 3, 1, 2, 0; 3, 0, 2, 1]	[0, 3, 5, 6]	2	4 / 2	1 / 2	3	8

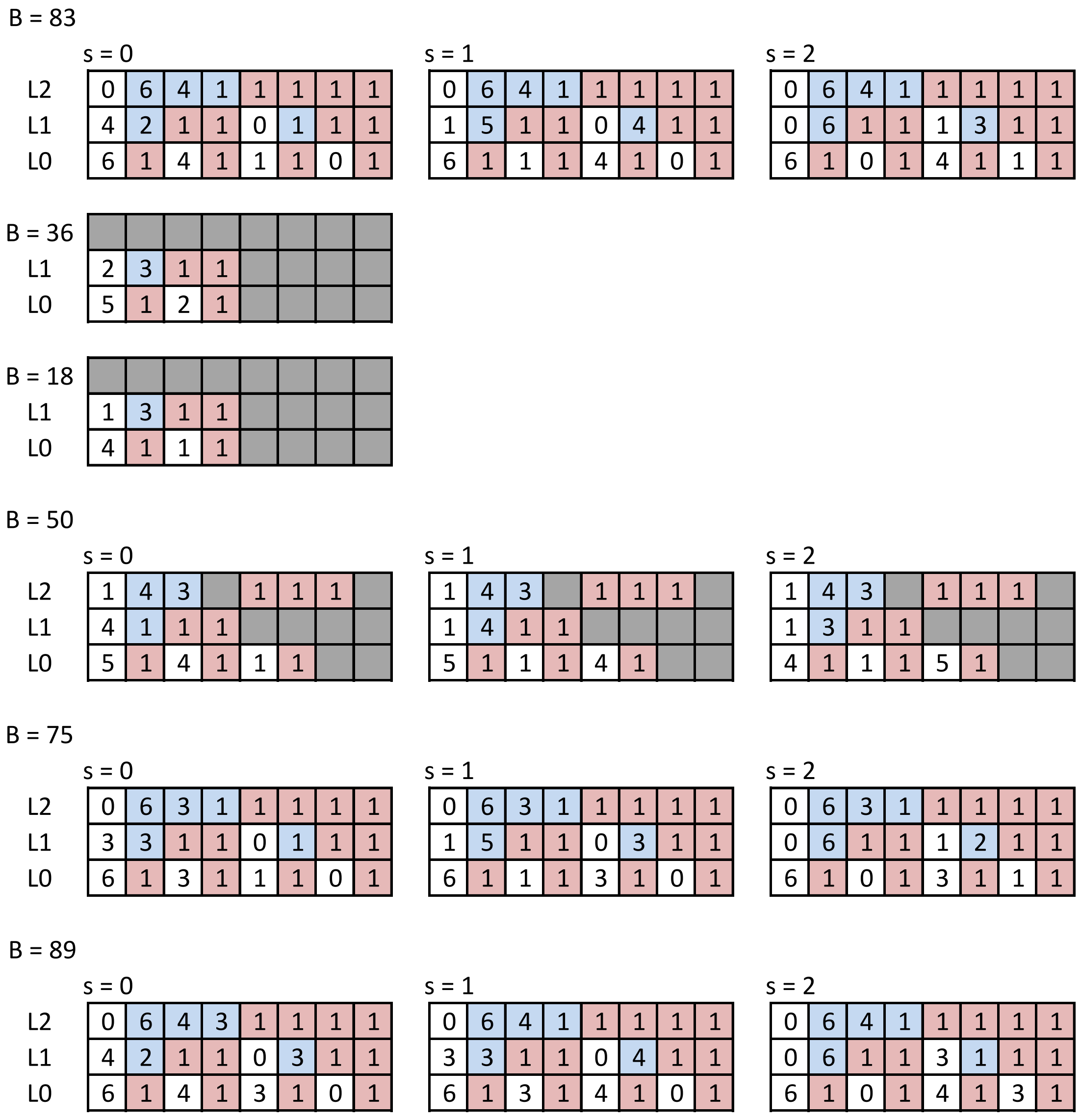
TABLE 6.8: Execution results of the resource optimization algorithm (Algorithm 8) when applying to scalar multiplication by $[83, 36]$ of the 4×4 Partial Butterfly algorithm.

$SP(0 \rightarrow +)$	SR
[0, 0]	1
[1, 0]	0
[2, 0]	1

Adding tree data are then generated (Algorithm 5) for all the multiplications in the 4×4 and 8×8 Partial Butterfly Algorithms (Figure 6.9).

Table 6.8 shows the intermediate results when searching for the permutation set SP with the highest SR in all STOOS space using Algorithm 8. The scalar multiplication in this example is $[83, 36]$ of the 4×4 Partial Butterfly algorithm. When designing integer transform architectures, a good strategy is to design the small transform blocks as parts of the large transform blocks to save resources. Although the 8×8 Partial Butterfly algorithm includes $[83, 36]$ and $[18, 50, 75, 89]$, we do not apply the optimization method to the six multipliers. The optimized implementation of $[83, 36]$ in the 4×4 Partial Butterfly algorithm are kept. We only apply the method to optimize the implementation of scalar multiplication $[18, 50, 75, 89]$. Table 6.9 shows the searching results of the resource optimization algorithm (Algorithm 8).

According to the result when running the Resource Optimization Algorithm 8 for



B = 83

s = 0

L2	0	6	4	1	1	1	1	1
L1	4	2	1	1	0	1	1	1
L0	6	1	4	1	1	1	0	1

s = 1

L2	0	6	4	1	1	1	1	1
L1	1	5	1	1	0	4	1	1
L0	6	1	1	1	4	1	0	1

s = 2

L2	0	6	4	1	1	1	1	1
L1	0	6	1	1	1	3	1	1
L0	6	1	0	1	4	1	1	1

B = 36								
L1	2	3	1	1				
L0	5	1	2	1				

B = 18								
L1	1	3	1	1				
L0	4	1	1	1				

B = 50

s = 0

L2	1	4	3		1	1	1	
L1	4	1	1	1				
L0	5	1	4	1	1	1		

s = 1

L2	1	4	3		1	1	1	
L1	1	4	1	1				
L0	5	1	1	1	4	1		

s = 2

L2	1	4	3		1	1	1	
L1	1	3	1	1				
L0	4	1	1	1	5	1		

B = 75

s = 0

L2	0	6	3	1	1	1	1	1
L1	3	3	1	1	0	1	1	1
L0	6	1	3	1	1	1	0	1

s = 1

L2	0	6	3	1	1	1	1	1
L1	1	5	1	1	0	3	1	1
L0	6	1	1	1	3	1	0	1

s = 2

L2	0	6	3	1	1	1	1	1
L1	0	6	1	1	1	2	1	1
L0	6	1	0	1	3	1	1	1

B = 89

s = 0

L2	0	6	4	3	1	1	1	1
L1	4	2	1	1	0	3	1	1
L0	6	1	4	1	3	1	0	1

s = 1

L2	0	6	4	1	1	1	1	1
L1	3	3	1	1	0	4	1	1
L0	6	1	3	1	4	1	0	1

s = 2

L2	0	6	4	1	1	1	1	1
L1	0	6	1	1	3	1	1	1
L0	6	1	0	1	4	1	3	1

FIGURE 6.9: Data generation of the optimization method for the 4×4 And 8×8 Partial Butterfly algorithms: $L[m, l, s, t]$, where $m = 0 \rightarrow M-1$, $l = 0 \rightarrow AD[m]$, $s = 0 \rightarrow S[m]-1$, and $t = 0 \rightarrow T[m]-1$. For 4×4 and 8×8 algorithms, $M = 2$ and 4, respectively.

Table 6.9: Execution results of the resource optimization algorithm (Algorithm 8)when applying to scalar multiplication by $[18, 50, 75, 89]$ of the 8×8 Partial Butterfly algorithm.

$SP(0 \rightarrow +)$	SR	$SP(0 \rightarrow +)$	SR	$SP(0 \rightarrow +)$	SR	$SP(0 \rightarrow +)$	SR	$SP(0 \rightarrow +)$	SR
[0, 0, 0, 0]	3	[0, 0, 2, 0]	2	[0, 1, 1, 0]	2	[0, 2, 0, 0]	3	[0, 2, 2, 0]	3
[0, 0, 0 ,1]	3	[0, 0, 2 ,1]	1	[0, 1, 1 ,1]	3	[0, 2, 0 ,1]	3	[0, 2, 2 ,1]	2
[0, 0, 0, 2]	3	[0, 0, 2, 2]	2	[0, 1, 1, 2]	1	[0, 2, 0, 2]	3	[0, 2, 2, 2]	2
[0, 0, 1, 0]	2	[0, 1, 0, 0]	2	[0, 1, 2, 0]	2	[0, 2, 1, 0]	3		
[0, 0, 1 ,1]	2	[0, 1, 0 ,1]	3	[0, 1, 2 ,1]	2	[0, 2, 1 ,1]	3		
[0, 0, 1, 2]	2	[0, 1, 0, 2]	1	[0, 1, 2, 2]	1	[0, 2, 1, 2]	2		

the Partial Butterfly algorithms, Selected Permutation SP for 4×4 is $[2, 0]$ with the highest number of reducible ASs $SR = 2$. For 8×8, SP is $[0, 2, 0, 0]$ with the highest $SR = 3$. Utilizing all the common ASs as labeled by the optimization algorithm, we can achieve AS implementations for the scalar multiplications in the Partial Butterfly algorithms ($[83, 36]$ and $[18, 50, 75, 89]$) as shown in Figure 6.10(c) and Figure 6.10(f). As can be seen, the proposed implementation for $[83, 36]$ only requires three ASs with the longest path of two ASs, while the proposed implementation for $[18, 50, 75, 89]$ only requires six ASs with the longest path of two ASs.

Based on the strategy of designing the small transforms as parts of the large transforms, and based on the optimized implementations of the two scalar multiplications in the Partial Butterfly algorithms, we propose a series of novel 4×4 and 8×8 fast and low-cost forward transform algorithms (Figure 6.11). In the figure, the 4×4 algorithm implementation is shown as a part of the 8×8 implementation (the dashed region). The numbers of ASs of the proposed implementations for the 4×4 and 8×8 transforms are fourteen and fifty-eight, respectively. Their longest paths consist of four and five ASs, respectively.

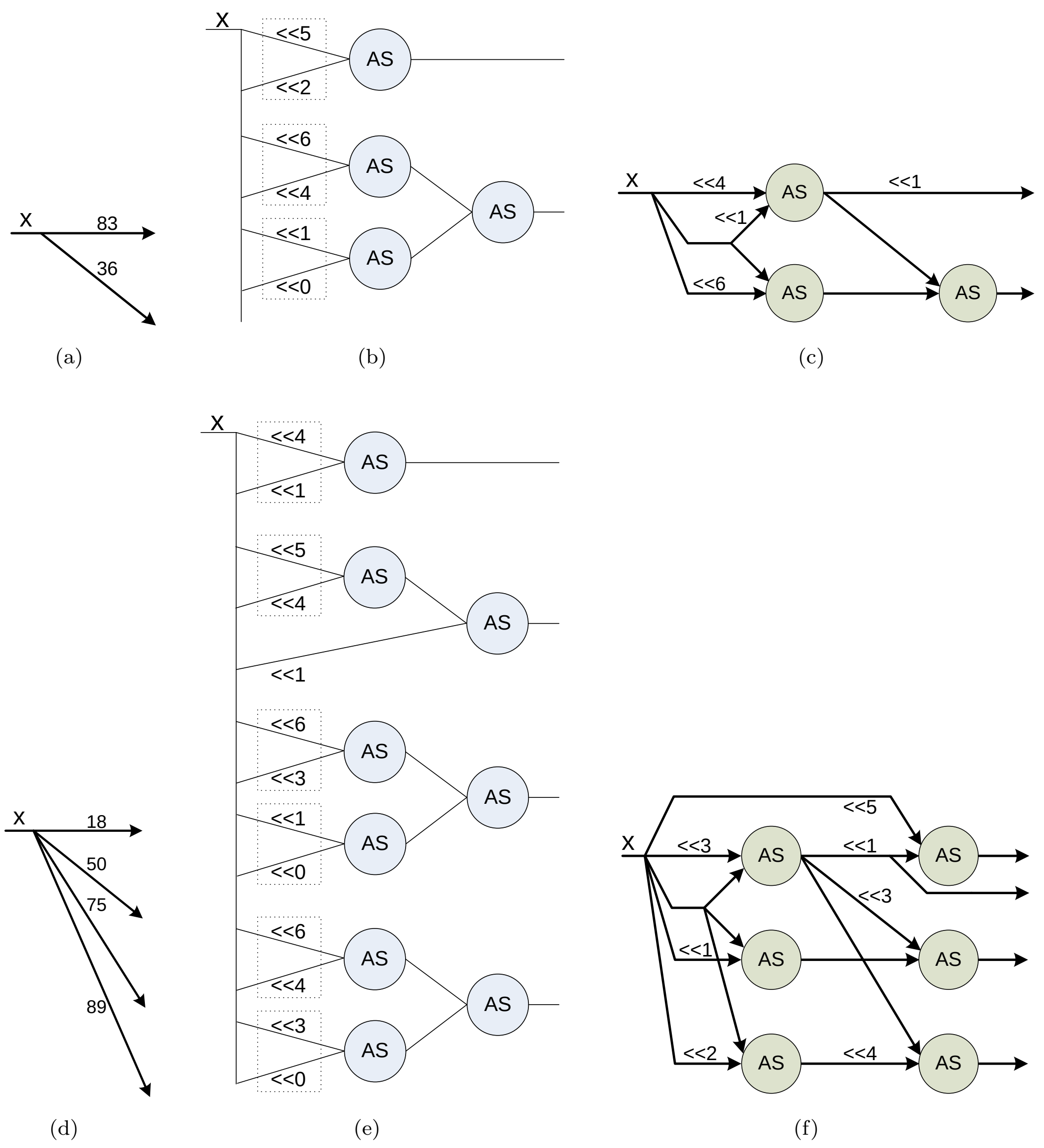


FIGURE 6.10: Data flows of different implementations for the two scalar multiplications by $[83, 36]$ and $[18, 50, 75, 89]$. (a) The original multiplication-by-$[83, 36]$ implementation using multiplications in the 4×4 Partial Butterfly algorithm; (b) the conventional parallel multiplication-free implementation for the multiplication by $[83, 36]$; and (c) the optimized implementation using the proposed optimization method for the multiplication by $[83, 36]$; (d) the original multiplication-by-$[18, 50, 75, 89]$ implementation using multiplications in the 8×8 Partial Butterfly algorithm; (e) the conventional parallel multiplication-free implementation for the multiplication by $[18, 50, 75, 89]$; and (f) the optimized implementation using the proposed optimization method for the multiplication by $[18, 50, 75, 89]$.

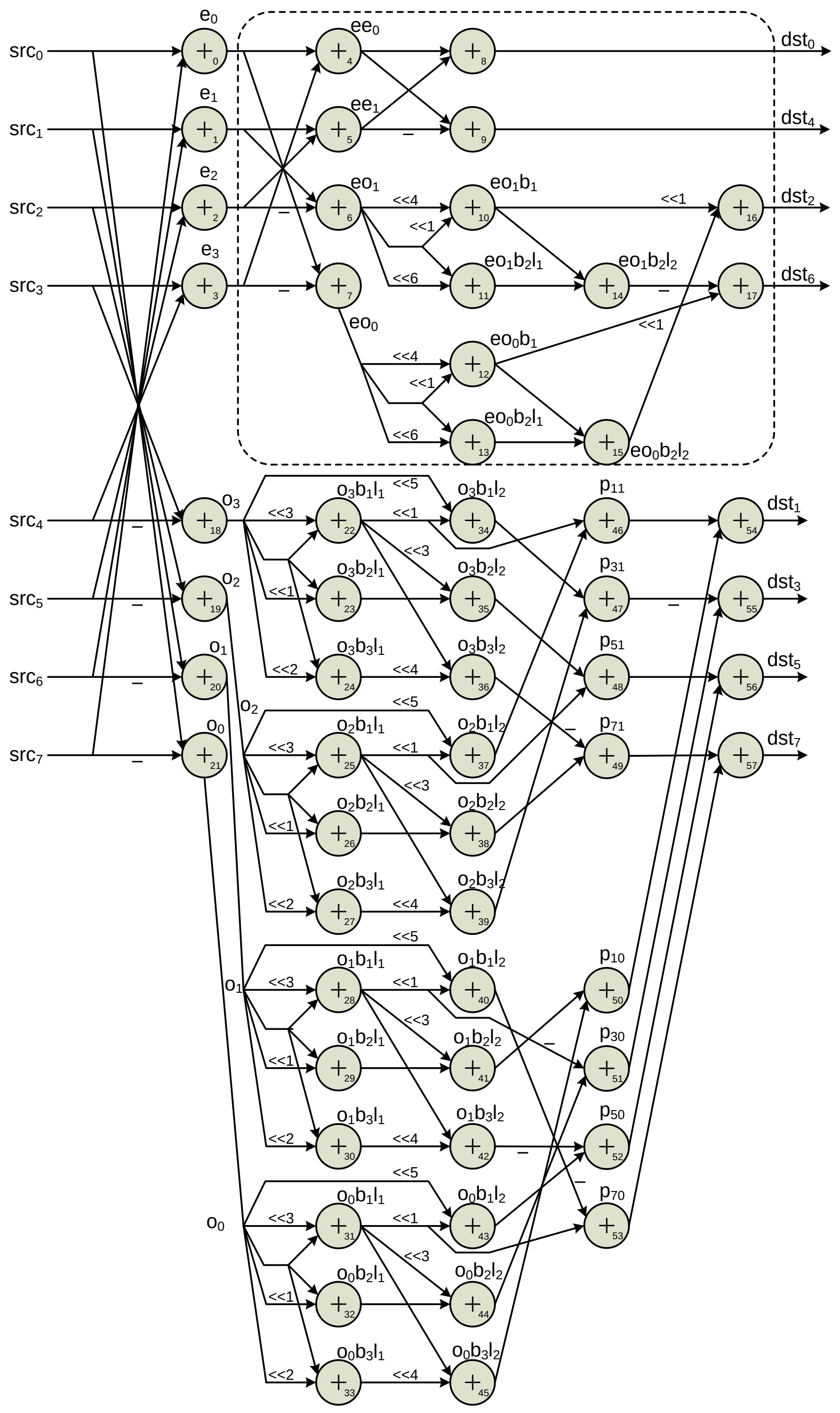


FIGURE 6.11: The proposed 8×8 1-D fast and low-cost Transform algorithms.

6.3.3 Discussion

6.3.3.1 Discussion on the Proposed Method

Table 6.10 and Figure 6.12 illustrate the longest path lengths and running time of three different implementations for four scalar multiplications. The four scalar multiplications includes two scalar multiplications by $[83, 36]$ and $[18, 50, 75, 89]$; the scalar multiplication portion in the 4×4 1-D Partial Butterfly algorithm; and the scalar multiplication portion in the 8×8 1-D Partial Butterfly algorithm. The three implementations includes (1) the original implementation using multiplications in the 1-D Partial Butterfly algorithms; (2) the conventional sequence multiplication-free implementation; and (3) the proposed implementation. It should be noted that ASs in implementation (2) are performed in sequence. In the case as many ASs are performed in parallel as possible, the longest path or the running time is the shortest, and it has the same value as that in implementation (3). It also should be noted that the running time of an implementation is proportional to its longest path. As can be seen, the implementation developed by the proposed method can finish the four scalar multiplications after two addition/subtractions. This running time is about 87% and 33% less than that of the original Partial Butterfly algorithm implementation and the conventional sequence multiplication-free implementation, respectively.

Table 6.11, Figure 6.13 and Figure 6.14 illustrate the number of ASs in three different implementations for four scalar multiplication algorithms. The four scalar multiplication algorithms includes two scalar multiplications by $[83, 36]$ and $[18, 50, 75, 89]$; the scalar multiplication portion in the 4×4 1-D Partial Butterfly algorithm; and the scalar multiplication portion in the 8×8 1-D Partial Butterfly

Table 6.10: The longest path lengths of three different implementations for four scalar multiplications. The four scalar multiplications includes two scalar multiplications by $[83, 36]$ and $[18, 50, 75, 89]$; the scalar multiplication portion in the 4×4 1-D Partial Butterfly algorithm; and the scalar multiplication portion in the 8×8 1-D Partial Butterfly algorithm. The three implementations includes (1) the original implementation using multiplications in the 1-D Partial Butterfly algorithms; (2) the conventional sequence multiplication-free implementation; and (3) the proposed implementation.

Algorithms	(1)	(2)	(3)
Scalar multiplication by $[83, 36]$	15 ASs	3 ASs	2 ASs
Scalar multiplication by $[18, 50, 75, 89]$ (multilications are implemented in parallel)	15 ASs	3 ASs	2 ASs
4×4 1-D Partial Butterfly with 2 scalar multiplications by $[83, 36]$ (scalar multilications are implemented in parallel)	15 ASs	3 ASs	2 ASs
8×8 1-D Partial Butterfly with 2 scalar multiplications by $[83, 36]$ and 4 scalar multiplications by $[18, 50, 75, 89]$ (scalar multilications are implemented in parallel)	15 ASs	3 ASs	2 ASs

Table 6.11: Number of ASs in three different implementations for four scalar multiplications. The four scalar multiplications includes two scalar multiplications by $[83, 36]$ and $[18, 50, 75, 89]$; the scalar multiplication portion in the 4×4 1-D Partial Butterfly algorithm; and the scalar multiplication portion in the 8×8 1-D Partial Butterfly algorithm. The three implementations includes (1) the original implementation using multiplications in the 1-D Partial Butterfly algorithms; (2) the conventional sequence / parallel multiplication-free implementation; and (3) the proposed implementation.

Algorithms	(1)	(2)	(3)
Scalar multiplication by $[83, 36]$	$2 \times 15 = 30$ ASs	4 ASs	3 ASs
Scalar multiplication by $[18, 50, 75, 89]$	$4 \times 15 = 60$ ASs	9 ASs	6 ASs
4×4 1-D Partial Butterfly with 2 scalar multiplications by $[83, 36]$	60 ASs	8 ASs	6 ASs
8×8 1-D Partial Butterfly with 2 scalar multiplications by $[83, 36]$ and 4 scalar multiplications by $[18, 50, 75, 89]$	300 ASs	44 ASs	30 ASs

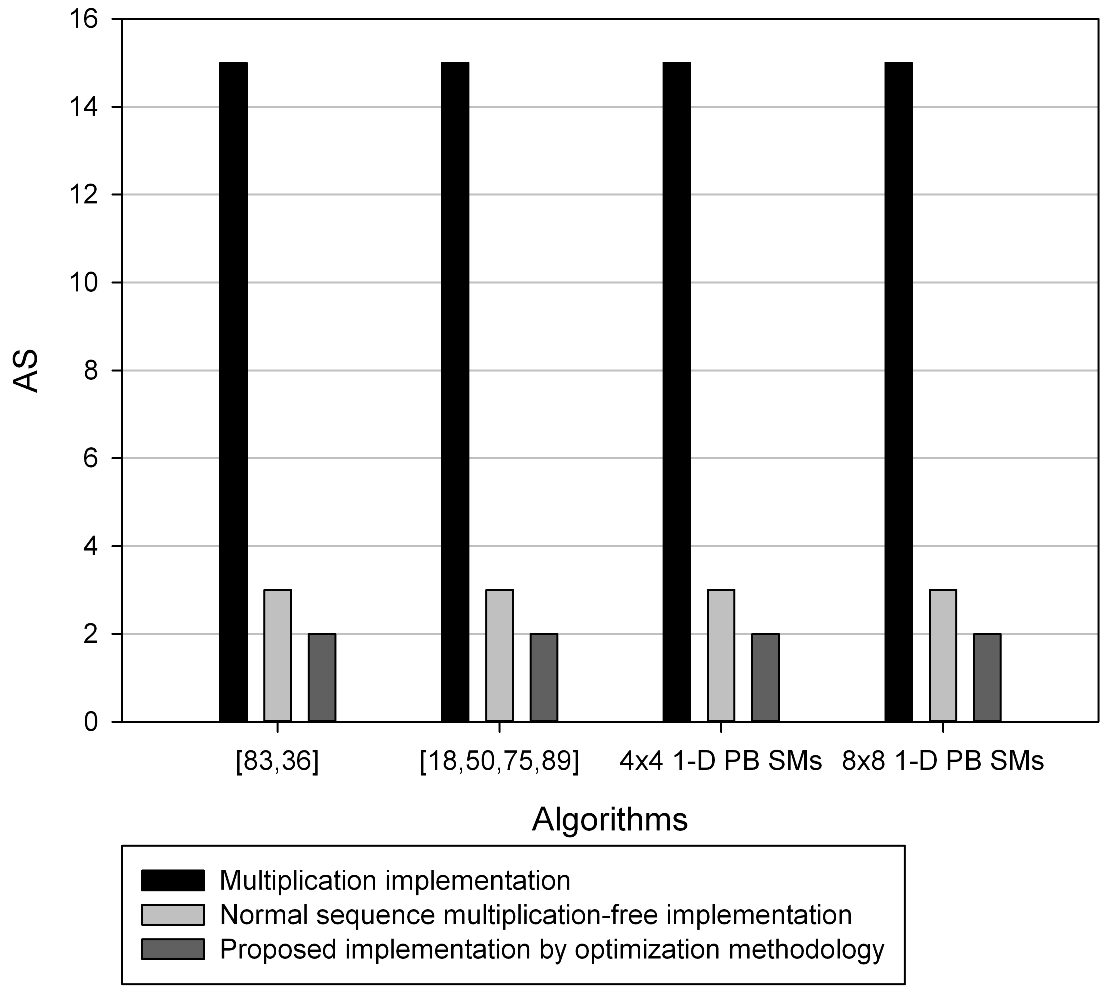


FIGURE 6.12: Running time of three different implementations for four scalar multiplications. The four scalar multiplications includes two scalar multiplications by $[83, 36]$ and $[18, 50, 75, 89]$; the scalar multiplication portion in the 4×4 1-D Partial Butterfly algorithm; and the scalar multiplication portion in the 8×8 1-D Partial Butterfly algorithm. The three implementations includes the original implementation using multiplications in the 1-D Partial Butterfly algorithms, the conventional sequence multiplication-free implementation, and the proposed implementation.

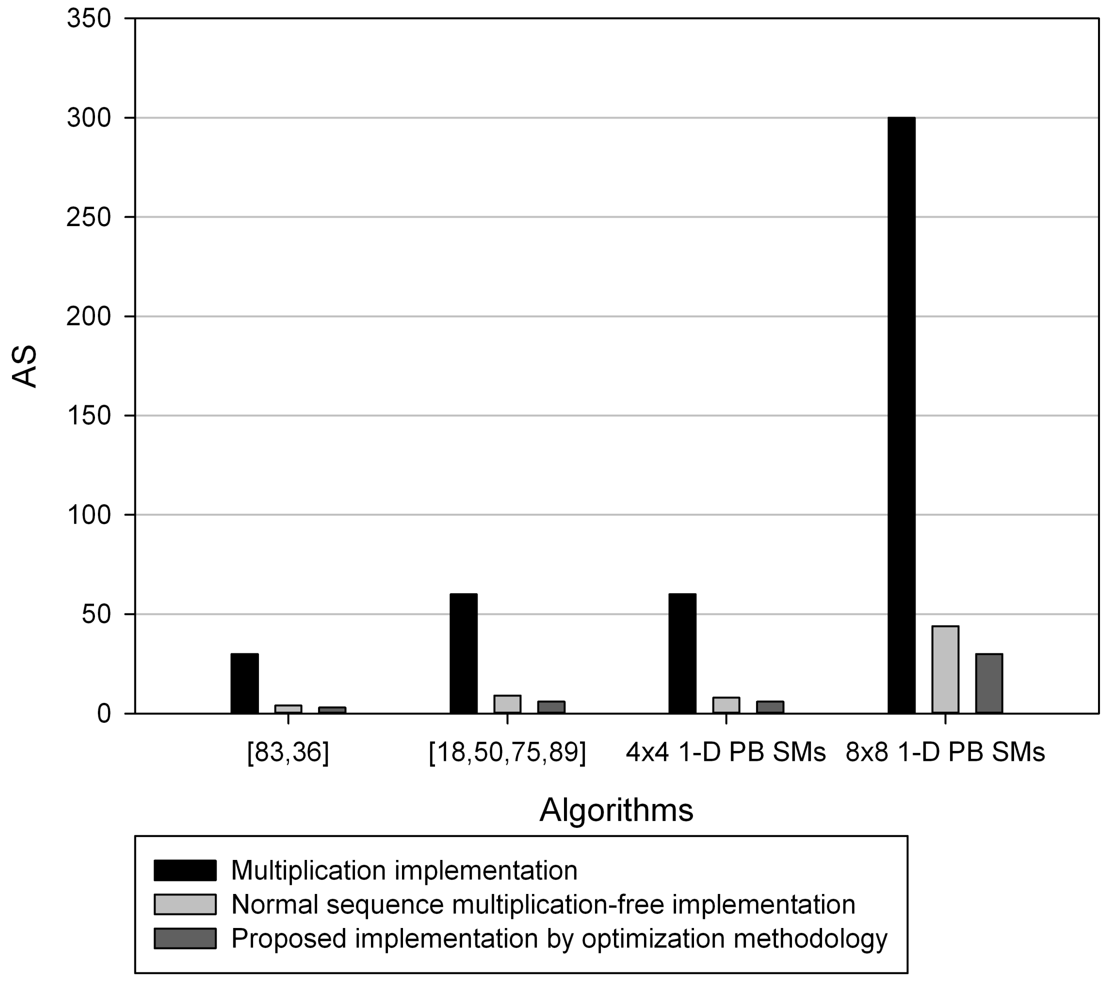


Figure 6.13: Resource consumptions of three different implementations for four scalar multiplications. The four scalar multiplications includes two scalar multiplications by [83, 36] and [18, 50, 75, 89]; the scalar multiplication portion in the 4×4 1-D Partial Butterfly algorithm; and the scalar multiplication portion in the 8×8 1-D Partial Butterfly algorithm. The three implementations includes the original implementation using multiplications in the 1-D Partial Butterfly algorithms, the conventional sequence / parallel multiplication-free implementation, and the proposed implementation.

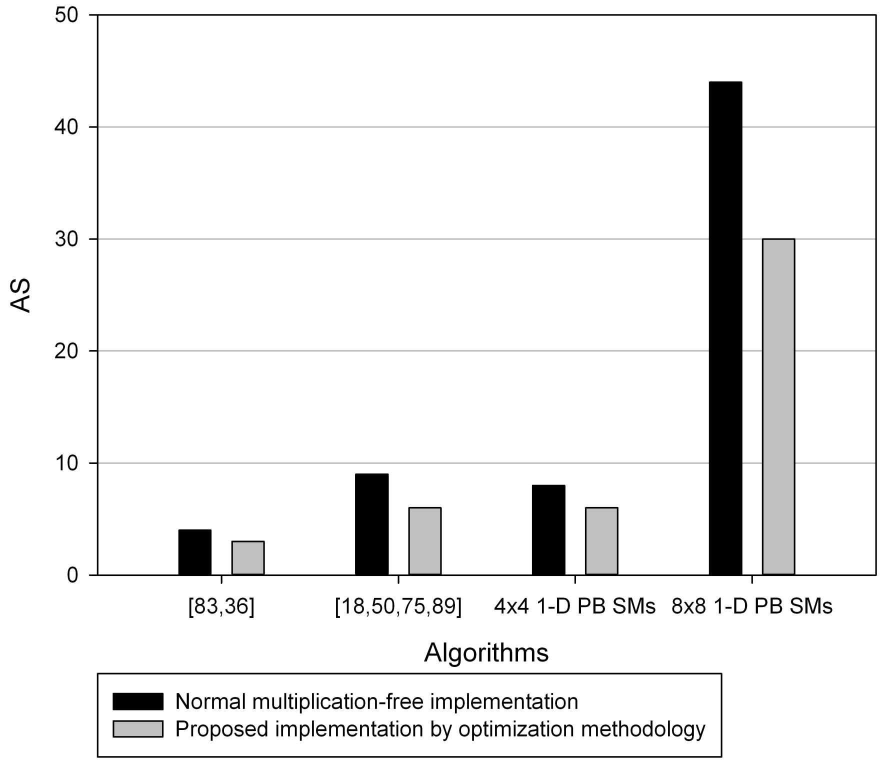


FIGURE 6.14: Resource consumptions of the conventional sequence / parallel multiplication-free and the proposed implementations for four scalar multiplications. The four scalar multiplications includes two scalar multiplications by $[83, 36]$ and $[18, 50, 75, 89]$; the scalar multiplication portion in the 4×4 1-D Partial Butterfly algorithm; and the scalar multiplication portion in the 8×8 1-D Partial Butterfly algorithm.

algorithm. The three implementations includes (1) the original implementation using multiplications in the 1-D Partial Butterfly algorithms; (2) the conventional sequence / parallel multiplication-free implementation; and (3) the proposed implementation. As can be seen, the numbers of ASs required in the implementation developed by the proposed method are three, six, six and thirty for the two scalar multiplications by $[83, 36]$ and $[18, 50, 75, 89]$ and the two scalar multiplication portions in the 4×4 and 8×8 1-D Partial Butterfly algorithms, respectively. These numbers of ASs are only 10% of those of implementation (1). The numbers of ASs required in the proposed implementation for the scalar multiplication by $[83, 36]$ and the scalar multiplication portion in the 4×4 1-D Partial Butterfly algorithm are 25% less than those of implementation (2). For the scalar multiplication by $[18, 50, 75, 89]$ and the scalar multiplication portion in the 8×8 1-D Partial Butterfly algorithm, the proposed implementation requires about 33% less ASs than implementation (2).

In the proposed optimization method, we observe that the complexity optimization algorithm at optimization level 1, with time complexity $O(n)$, can guarantee an optimal MACR using only additions and subtractions. It searches for all the possibilities to reduce the number of ASs for each $'1'$ chain without affecting the conversion of the other $'1'$ chains.

In the second level of optimization, the proposed adding tree guarantees the shortest running time for each multiplication because it utilizes the parallelism of ASs as much as possible. The only thing preventing the running time from being shorter is the operation dependency. However,this dependency cannot be further optimized.

The STOOS generation algorithm (Algorithm 4) guarantees a full set of permutations with different operation orders of the minimized MACR operands. The algorithm has a factorial time complexity.

In the third optimization level, the adding tree data generation and the searching in the STOOS space lead to a very competitive implementation. The implementation greatly reduces the resource consumption. However, the time complexity for the full search is large (factorial-exponential). The number of elements in the search space is

$$\mathrm{S}^{\mathrm{M}} = \left(\frac{N_I!}{\left\lfloor \frac{N_I}{2} \right\rfloor! \, (2!)^{\left\lfloor \frac{N_I}{2} \right\rfloor}} \right)^M , \tag{6.19}$$

TABLE 6.12: Sizes of the search spaces for the Partial Butterfly algorithms.

Partial Butterfly Algorithms	N	M	NI_{max}	S_{max}	Max no. of search cases
4×4	7	2	4	3	9
8×8	7	4	4	3	81
16×16	7	8	4	3	6 561
32×32	7	16	4	3	43 046 721

where each element is a set of operation orders and each operation order is selected for a multiplication among the STOOS of that multiplication. We name an element in the search space as permutation set (PS). The number of PSs is the size of the STOOS search space.

In the 4×4 to 64×64 Partial Butterfly algorithms, the multipliers of scalar multiplication are smaller than 128 or 7-bit wide. It can be proved that after the first optimization level, the number of non-zero elements, N_I, in array CA is

$$N_I \leq \frac{N}{2}, \tag{6.20}$$

where N is the number of bits used to represent the multipliers. Therefore, $N_I \leq \frac{N}{2} = 4$. Hence, the number of permutations in $STOOS$ of each multiplication (Equation (6.17)) is

$$S \leq \frac{N_I!}{\left\lfloor \frac{N_I}{2} \right\rfloor! \, (2!)^{\left\lfloor \frac{N_I}{2} \right\rfloor}} = 3. \tag{6.21}$$

The maximum number of permutation sets (PSs) among all multiplication is then 3^M, where M is the number of multiplications. In the 4×4, 8×8, 16×16 and 32×32 Partial Butterfly algorithms, $M = 2$, 4, 8 and 16, respectively. Table 6.12 shows the maximum number of PSs for all the Partial Butterfly algorithms. As can be seen, with the listed number of search cases, the search algorithm in the proposed method completely are able to deal with all the Partial Butterfly algorithms.

Table 6.13: Sizes of search spaces for different N_Is.

NI	2	3	4	5	6	7	8
S	1	3	3	15	15	105	105
No. of search cases	1	3^M	3^M	15^M	15^M	105^M	105^M

Table 6.13 shows the maximum number of search cases for different N_I.

Although for scalar-multiplication algorithms, the number of search cases is large and depends on the number of $'1'$ bits after the first optimization N_I and the number of multipliers M, all search cases are independent. Therefore, for the scalar-multiplication-containing algorithms which have multipliers greater than or equal to 65536 (16-bit wide) and have large number of multiplications in the scalar multiplications, searching and computing objective function of each case can be setup to be run in different computers or parallel computing. In addition, the search algorithm only needs to run once, then we can use the optimized algorithms to develop hardware architectures.

6.3.3.2 Discussion on the Proposed IT Algorithms

Table 6.14, Figure 6.15, Table 6.15 and Figure 6.16 illustrate the longest path, running time, number of ASs and resource consumption, respectively, of the Partial Butterfly algorithms, the conventional multiplication-free Partial Butterfly algorithms, and the proposed integer transform algorithms. As can be seen, the proposed algorithms can fully perform the 4×4 and 8×8 1-D transforms after 4 and 5 addition/subtractions, respectively. Compared to the original Partial Butterfly algorithms and the conventional sequence multiplication-free Partial Butterfly algorithms, the speed of the proposed algorithms increases by 75% and 20%, respectively. By using fourteen and fifty-eight ASs for the 4×4 and 8×8 transforms respectively, the proposed algorithms requires around 87% and 16% less resource

TABLE 6.14: The longest path lengths of the three integer transform algorithms: the Partial Butterfly, the conventional sequence multiplication-free Partial Butterfly and the proposed integer transform algorithms.

Transform	Partial Butterfly	Conventional sequence multiplication-free Partial Butterfly algorithm	Proposed integer transform algorithm
4 x 4 1-D	17	5	4
8 x 8 1-D	19	6	5

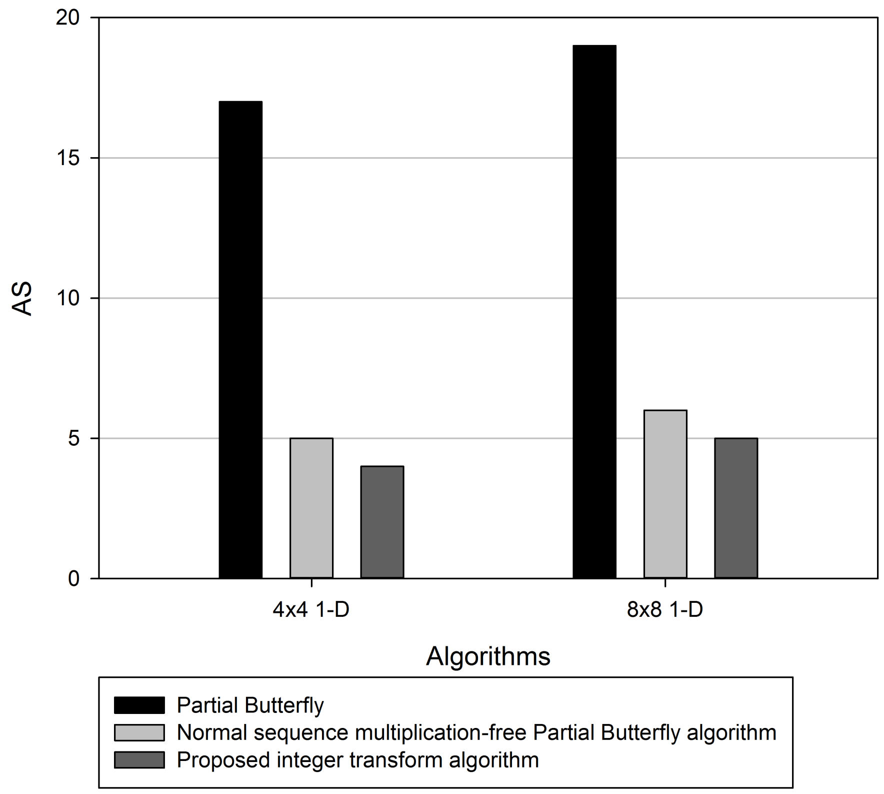


FIGURE 6.15: Running time of the three integer transform algorithms: the Partial Butterfly, the conventional sequence multiplication-free Partial Butterfly and the proposed integer transform algorithms.

than the original and the conventional sequence/parallel multiplication-free Partial Butterfly algorithms, respectively. Table 6.16 shows further details of the total number of ASs and shift operations (Ss) used in the conventional sequence/parallel multiplication-free algorithms and the proposed algorithms for the $4 \times 4/8 \times 8$ 1-D/2-D forward transforms.

TABLE 6.15: Number of ASs of the three integer transform algorithms: the Partial Butterfly, the conventional sequence / parallel multiplication-free Partial Butterfly and the proposed integer transform algorithms.

Transform	Partial Butterfly	Conventional sequence/parallel multiplication-free Partial Butterfly algorithm	Proposed integer transform algorithm
4 x 4 1-D	128	16	14
8 x 8 1-D	388	72	58
4 x 4 2-D	1024	128	112
8 x 8 2-D	6208	1152	928

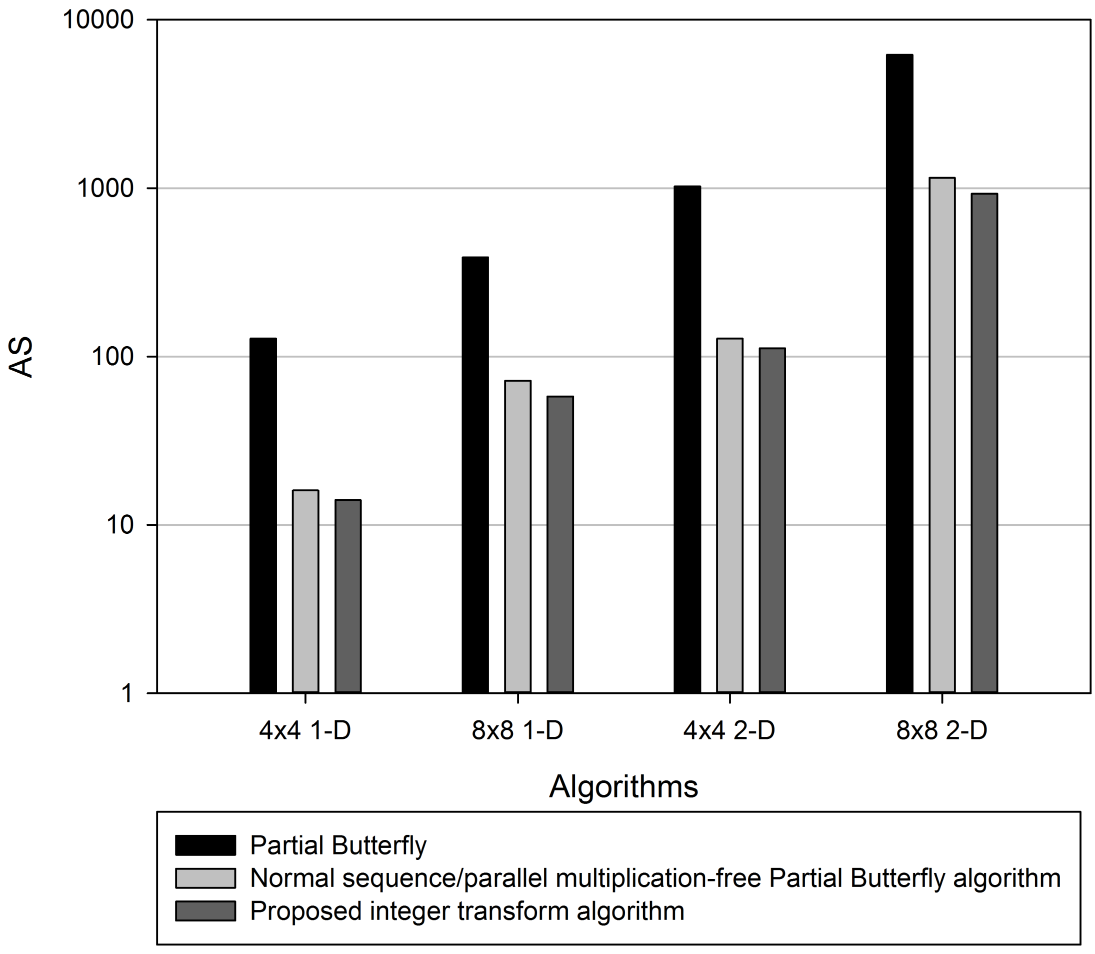


FIGURE 6.16: Resource consumption of the three integer transform algorithms: the Partial Butterfly, the conventional sequence / parallel multiplication-free Partial Butterfly and the proposed integer transform algorithms.

TABLE 6.16: Number of additions and shift operations in two series of the Partial Butterfly-based integer transform algorithms: the conventional multiplication-free algorithms and the proposed integer transform algorithms for the $4\times4/8\times8$ 1-D/2-D forward transforms.

Transform	Number of operations in the conventional multiplication-free algorithms	Number of operations in the proposed algorithms
4×4 1-D	$4As+2As+1S+(4As+5Ss)\times2+2As=16As+11Ss$	$4As+2As+1S+(3As+4Ss)\times2+2As=14As+9Ss$
8×8 1-D	$8As+(16As+11Ss)+(9As+11Ss)\times4+8As+4As=72As+55Ss$	$8As+(14As+9Ss)+(6As+7Ss)\times4+8As+4As=58As+37Ss$
4×4 2-D	$128As+88Ss$	$112As+72Ss$
8×8 2-D	$1152As+880Ss$	$928As+592Ss$

TABLE 6.17: The longest path length and resource consumption of the proposed algorithms in comparison with those of other published HEVC integer transform algorithms.

1-D Algorithms	8×8 Martuza and Wahid (2012)	4×4 Rithe et al. (2012)	8×8 Rithe et al. (2012)	Proposed 4×4	Proposed 8×8
Length of the longest path	≥ 7	4	6	4	5
Resource consumption	72	18	60	14	58

Table 6.17 lists the longest path and resource consumption in AS unit of the proposed algorithms in comparison with other published algorithms for HEVC. Since Martuza and Wahid (2012) reported the number of adders used in the architecture but did not report the number of additions in the algorithm, the details in the table are calculated based on the authors' report. As can be seen, the speed of the proposed 8×8 algorithm increases by 29% and 17% compared to that of Martuza and Wahid (2012) and Rithe et al. (2012)'s 8×8 algorithms, respectively. Its cost reduces by 19% and 3% compared to that of Martuza and Wahid (2012) and Rithe et al. (2012)'s algorithms, respectively. The proposed 4×4 algorithm is as fast as the Rithe et al. (2012)'s algorithm, but consumes 22% less resource.

6.4 A Novel High-Throughput Area-Efficient Architecture

6.4.1 High Throughput Requirement under I/O Pin Constraints

In soft and hard IP designs, number of input/output (I/O) pins are normally limited due the cost of wire area and packaging. However, in video applications, more I/O pins/wires leads to less time needed for data transfer between IPs/ICs. Therefore, much more effort is put to design applications for high resolution video with a limited number of I/O pins.

In order to solve a series of design problems which requires high throughputs with a limited number of I/O pins, a big challenge has been set for this architecture design with a very tight constraint on number of I/O pins at sixteen pins. It should be noted that HEVC requires 16-bit depth for each pixel and high throughput to process ultra high resolution video. With HEVC 16-bit depth requirement, the constraint limits the system to inputting half-pixel and outputting half-pixel at a time. Dealing with ultra high resolution video under the half-pixel I/O condition is a tough work requiring a large effort to design. Once the architecture design can fulfill the HEVC requirement with this tight constraint, it guarantees that designs for looser constraints are feasible by applying the same techniques.

The tight constraint leads to sixteen cycles to only input or output a row/column of eight pixels, and eight cycles to input or output a row/column of four pixels. Running time of a system includes input time, processing time and output time. The processing time can vary depending on the system design, but the I/O time is fixed when the number of I/O pins is fixed. Therefore, the shortest running time is the input time or output time. This can only be achieved by applying pipelining mechanisms for input, output and processing operations, and at the same time, the processing time is shorter or equal to the I/O time. In order to do pipelining efficiently, time to process a 8-pixel and 4-pixel row/column should not exceed the I/O times, i.e., sixteen and eight cycles, respectively.

The I/O times are fixed at sixteen and eight cycles for the 8×8 and 4×4 transforms, respectively. Since the running time cannot be shorter than the I/O time, we cannot further optimize the running time in terms of number of cycles. We only can optimize it by shortening the period, i.e., increasing the operating frequency of the system. By minimizing the running time, the throughput is maximized. Therefore, in this design, the applied techniques can be classified into two categories: one is to design an efficient pipelining mechanism, and the other is to shorten the operating period. In addition to these two categories, another applied technique is to minimize resource consumption.

6.4.2 Architecture Design

The design requires sixteen input and output pins. Hence, eight pins are used for input and eight pins are used for output. As described in the HEVC standard, the transform inputs are 16-bit depth, while internal operations are 32-bit depth. Therefore, the proposed architecture has half-pixel input and half-pixel output. All the internal components are 32-bit depth. The following sub-sections describe the techniques used to achieve high-throughput and area-efficient architecture designs even under such very tight I/O constraints.

6.4.2.1 Multi-Cycle Adder Design for High Frequency

We select the 1-D algorithms proposed in Section 6.3.2 for this architecture design due to its low complexity, high throughput and low resource consumption (Figure 6.11). Based on the analysis in Section 6.4.1, we are given processing time constraints of eight and sixteen cycles for the 4×4 and 8×8 transforms.

As can be seen in Table 6.14, the lengths of the longest paths of the proposed 4×4 and 8×8 algorithms are four and five additions, respectively. If we design in such a way that each 1-D algorithm is performed in one cycle, then the period is equal to the running time of four additions for the 4×4 transforms and five additions for the 8×8 transforms. If we design in a way that each addition is performed in one cycle, then the period is equal to the running time of an addition; and it takes four and five cycles to complete the 4×4 and 8×8 transforms, respectively. The second way is feasible and better as we have eight and sixteen cycles for the 4×4 and 8×8 transforms, respectively.

In order to further reduce the operating period, we can implement a special multi-cycle adder, which can perform an addition in several cycles, so that the period can be a part of the running time of the addition. Since we have eight cycles to perform the 4×4 transform and the longest path in the proposed 4×4 algorithm includes four additions, we can implement each addition as a 2-cycle adder to utilize all eight cycles. Similarly, each addition can be implemented as a 3-cycle adder in the 8×8 transform to utilize all fifteen cycles.

In this architecture design, both the 4×4 and 8×8 transforms are included. Therefore, the optimal choice is implementing each addition as a 2-cycle adder. As a result, eight and ten cycles are required to complete the 4×4 and 8×8 transforms, respectively. This satisfies the initial requirement stated in Section 6.4.1.

Since each adder needs to perform both addition and subtraction, ripple carry adder/subtractor is often selected for this class of designs. Let us assume that we need to implement a n-bit addition by a m-cycle adder. Instead of using a n-bit adder, we use a $\lceil \frac{n}{m} \rceil$-bit adder and design a controller to control the adder to add different sets of bits in different cycles, from the least significant bit sets to the most significant bit sets. Since the running time of a ripple carry adder/subtractor is proportional to the binary length of the inputs to be added, by using $\lceil \frac{n}{m} \rceil$-bit adder, we can reduce the period about m times.

Not only reducing the period, but using multi-cycle adders also saves resource. Since an n-bit ripple carry adder/subtractor includes n 1-bit adder/subtractors, using $\lceil \frac{n}{m} \rceil$-cycle adders consumes m times less resource than using n-bit adder/-subtractors.

Therefore, by using 2-cycle adders, we can reduce the period and resource consumption by two times compared to using conventional adders if each adder is designed to perform in a cycle. If the entire 1-D 4×4 or 8×8 transform is designed to perform in a cycle, using 2-cycle adders can reduce the period by eight or ten times, respectively, and reduce the resource consumption by two times compared to using conventional adders.

6.4.2.2 Pipeline Scheduling for 1-D Transforms

The scheduled sequencing graphs with 2-cycle adders for the proposed 4×4 and 8×8 transform algorithms can be found in Figure 6.17 and Figure 6.18. Subsequently,

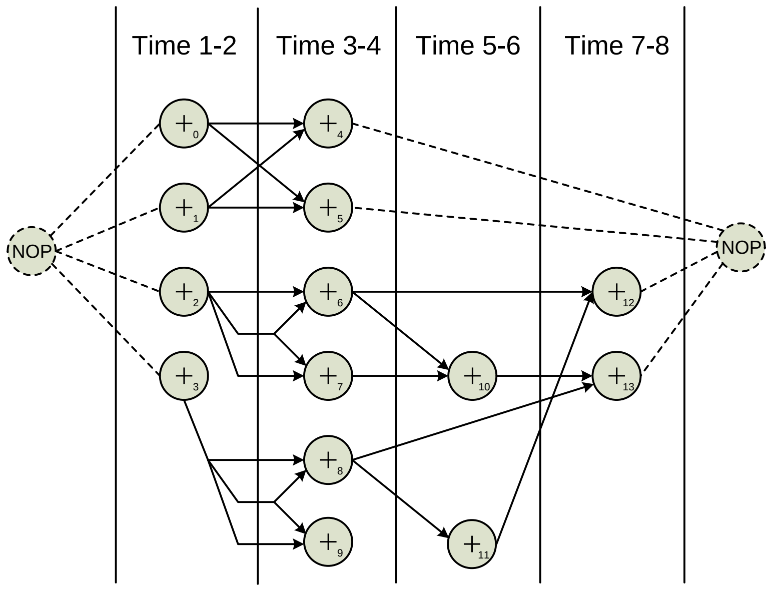


FIGURE 6.17: The proposed scheduled sequencing graph for the 4×4 1-D fast and low-cost forward transform algorithms.

TABLE 6.18: Resource scheduling for the proposed 4×4 1-D transform algorithm.

Operations	Start time
v_0, v_1, v_2, v_3	1
v_4, v_5, v_6, v_7, v_8, v_9	3
v_{10}, v_{11}	5
v_{12}, v_{13}	7

TABLE 6.19: Resource scheduling for the proposed 8×8 1-D transform algorithm.

Operations	Start time
v_0, v_1, v_2, v_3, v_{18}, v_{19}, v_{20}, v_{21}	1
v_4, v_5, v_6, v_7, v_{22}, v_{23}, v_{24}, v_{25}, v_{26}, v_{27}, v_{28}, v_{29}, v_{30}, v_{31}, v_{32}, v_{33}	3
v_8, v_9, v_{10}, v_{11}, v_{12}, v_{13}, v_{34}, v_{35}, v_{36}, v_{37}, v_{38}, v_{39}, v_{40}, v_{41}, v_{42}, v_{43}, v_{44}, v_{45}	5
v_{14}, v_{15}, v_{46}, v_{47}, v_{48}, v_{49}, v_{50}, v_{51}, v_{52}, v_{53}	7
v_{16}, v_{17}, v_{54}, v_{55}, v_{56}, v_{57}	9

we have the resource schedules for the proposed 4×4 and 8×8 1-D FITs in Table 6.18 and Table 6.19.

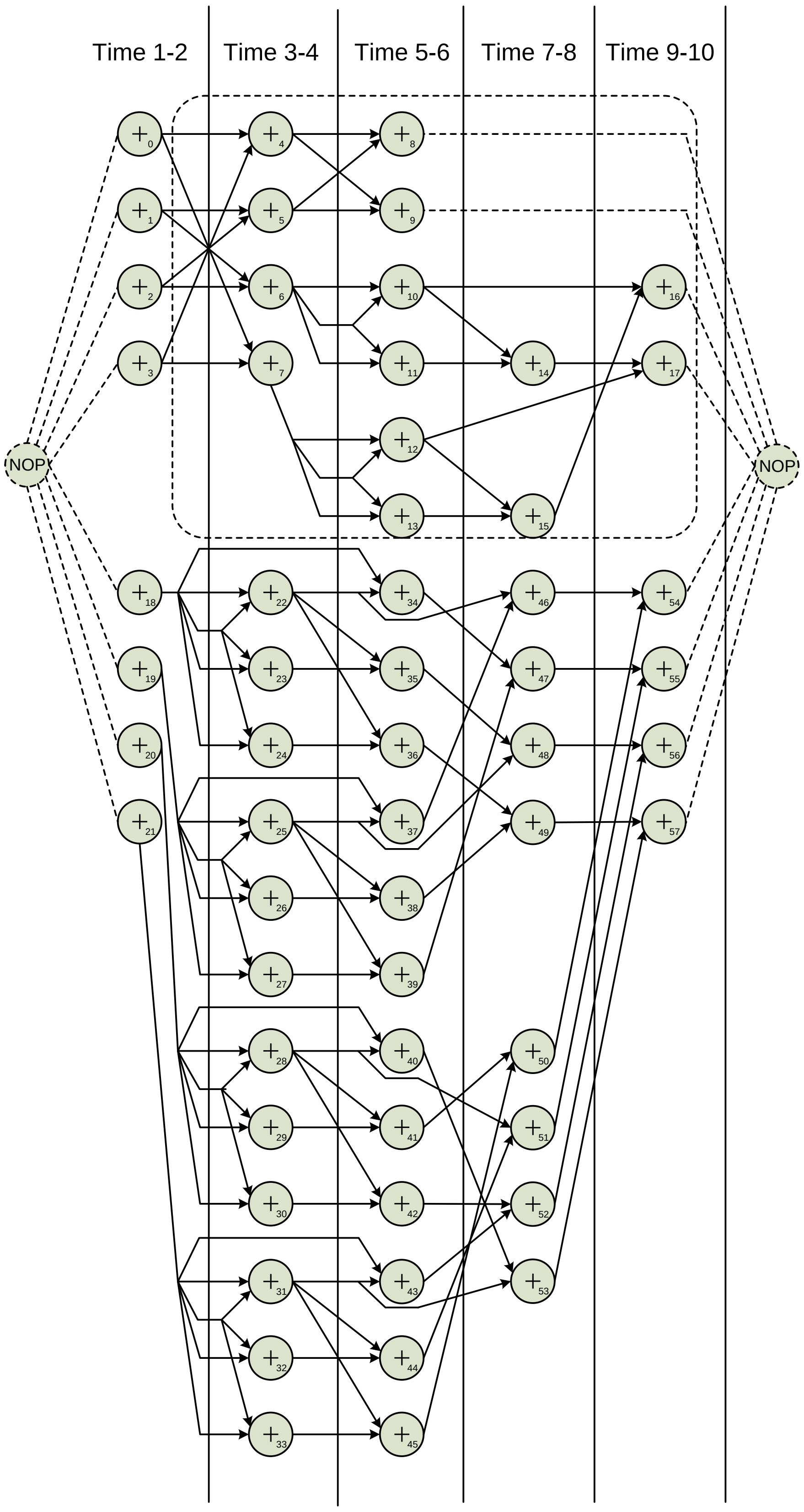


FIGURE 6.18: The proposed scheduled sequencing graph for the 8 × 8 1-D fast and low-cost forward transform algorithms.

6.4.2.3 Multi-Stage Register Design for Pipeline

As the system works in a pipelining mechanism, output of each addition/subtraction (AS) is required to be stored in a register, so that it can be used in the next stage or time slot. If several ASs in different stages are grouped to share hardware, i.e., we use an adder for these ASs and use a register to store the adder output, then in each stage, the register is loaded and restored with the output of the corresponding AS in the stage. Once it is restored, the previous value is overwritten as the register only can store one value at a time.

As can be seen in Figure 6.18, some outputs may be used in several stages or time slots. For example, the outputs of AS_{18}, AS_{19}, AS_{20} and AS_{21} are used as inputs in both time slots 3-4 and 5-6. The outputs of AS_{22}, AS_{25}, AS_{28} and AS_{31} are used as inputs in both time slots 5-6 and 7-8. If conventional registers are used and we want to keep the outputs in several stages, the registers storing these ASs' outputs cannot be shared. It means that if some of these ASs are grouped with other ASs to share hardware, separate registers are needed to store the outputs of these ASs. These separate registers cannot be shared or reused, leading to a resource waste. In addition, it also takes time and raises difficulties to select which register to store the output of the shared adder.

Therefore, a special register type, i.e., multi-stage register, is designed to store these outputs, which can hold the stored values in multiple stages. Assuming the output are n bits long, the length of a m-stage register is $n \times m$ bits. When the register stores a value, it shifts the current stored data to the left n bits and store the value into the n least significant bits. Therefore, by using this m-stage register, we can retrieve the m-bit value from the register not only in the next stage but also after m stages from the register. In addition, the writing operation of this register is the same as that of a conventional register. The advantages of using this m-stage register compared to using m separate registers includes (1) the m-stage register can be shared and reused, while m separate registers cannot be reused in other stages; and (2) storing ASs' outputs into the m-cycle register once the shared adder completes its computation is faster and more convenient than selecting which register to store.

6.4.2.4 Resource Binding for Low Resource Consumption

As addition/subtractions (ASs) are scheduled to run in different times, resource sharing can be enabled. By binding different ASs in different timing, we can use just one adder/subtractor to perform all these ASs. Therefore, the more ASs we can bind together, the more resource we can save. The optimal choice is using the number of adder/subtractors equal to the maximum number of ASs performing at the same time. As can be seen from Figure 6.18, Time 5-6 uses the maximum number of ASs, i.e., 18, among all the time slots; and this number of adder/subtractors is the best choice.

However, binding ASs also leads to the use of multiplexors (MUXs). Binding two ASs requires a 2-1 MUX. Binding three or four ASs needs a 4-1 MUX, while binding from four to seven ASs generates a 8-1 MUX. Therefore, during binding, we also need to optimize the number of MUXs and its sizes. Figure 6.19 shows our proposed scheduled sequencing graph with resource binding for the proposed 8×8 1-D FIT. A detail resource binding can be seen in Table 6.20. In Figure 6.20, the ASs are re-indexed following the corresponding binding resources. It should be noted that the dashed region is also the same as the 4×4 transform. Hence, in order to design both transforms, the ASs from number 4 to 7 have to select one

TABLE 6.20: Resource binding for the proposed 8×8 1-D transform algorithm.

Binding	Resource	Binding	Resource	Binding	Resource
B(v_0)	0	B(v_{20})	8	B(v_{40})	8
B(v_1)	1	B(v_{21})	9	B(v_{41})	14
B(v_2)	2	B(v_{22})	6	B(v_{42})	15
B(v_3)	3	B(v_{23})	10	B(v_{43})	9
B(v_4)	0	B(v_{24})	11	B(v_{44})	16
B(v_5)	1	B(v_{25})	7	B(v_{45})	17
B(v_6)	2	B(v_{26})	12	B(v_{46})	7
B(v_7)	3	B(v_{27})	13	B(v_{47})	6
B(v_8)	0	B(v_{28})	8	B(v_{48})	10
B(v_9)	1	B(v_{29})	14	B(v_{49})	12
B(v_{10})	4	B(v_{30})	15	B(v_{50})	14
B(v_{11})	2	B(v_{31})	9	B(v_{51})	16
B(v_{12})	5	B(v_{32})	16	B(v_{52})	9
B(v_{13})	3	B(v_{33})	17	B(v_{53})	8
B(v_{14})	2	B(v_{34})	6	B(v_{54})	14
B(v_{15})	3	B(v_{35})	10	B(v_{55})	16
B(v_{16})	3	B(v_{36})	11	B(v_{56})	10
B(v_{17})	2	B(v_{37})	7	B(v_{57})	12
B(v_{18})	6	B(v_{38})	12		
B(v_{19})	7	B(v_{39})	13		

between two different input sets: one is from the source for the 4×4 transform, and the other is from ASs number 0 to 3 for the 8×8 transform. This is the same as generating four more ASs to bind and may increase the size of four corresponding MUXs or may increase the number of MUXs.

As HEVC requires 16-bit depth pixels and 32-bit internal operations, we use 32-bit adder/subtractors. Let us assume that outputs of 32-bit Adder/Subtractors AS_0 to AS_5 (Figure 6.20) are stored in 32-bit registers R_0 to R_5, and those of 32-bit AS_{10} to AS_{17} are store in 32-bit registers R_6 to R_{13}. As outputs of AS_6 to AS_9 need to be held to use in two different stages or time slots, their outputs are designed to connect to special 64-bit long registers LR_0 to LR_3. Each long register can be divided into two parts, i.e., 32-bit LRL (Least Significant Part of LR) and

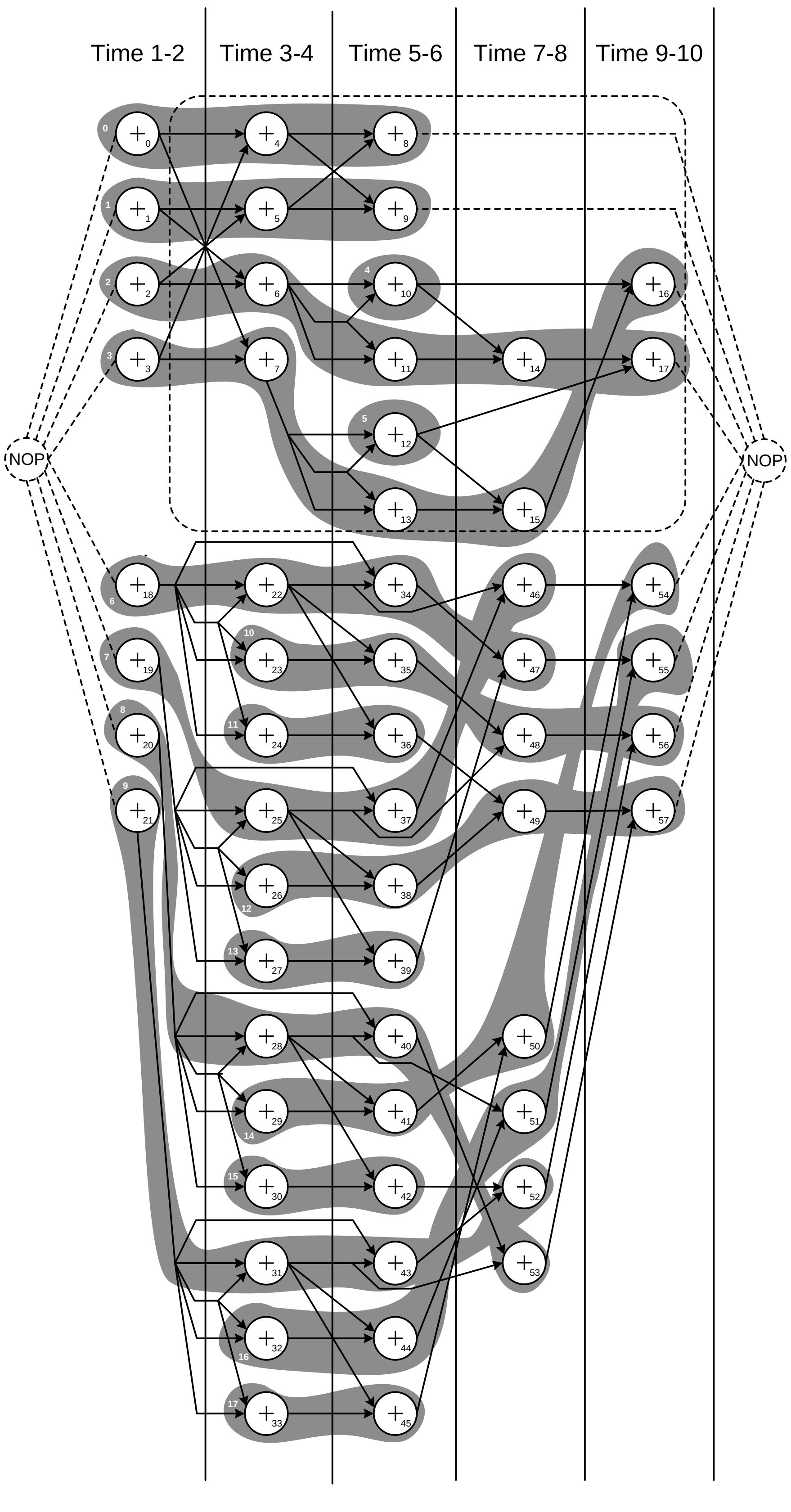


FIGURE 6.19: The proposed scheduled sequencing graph with resource binding for the proposed 8 × 8 1-D FIT.

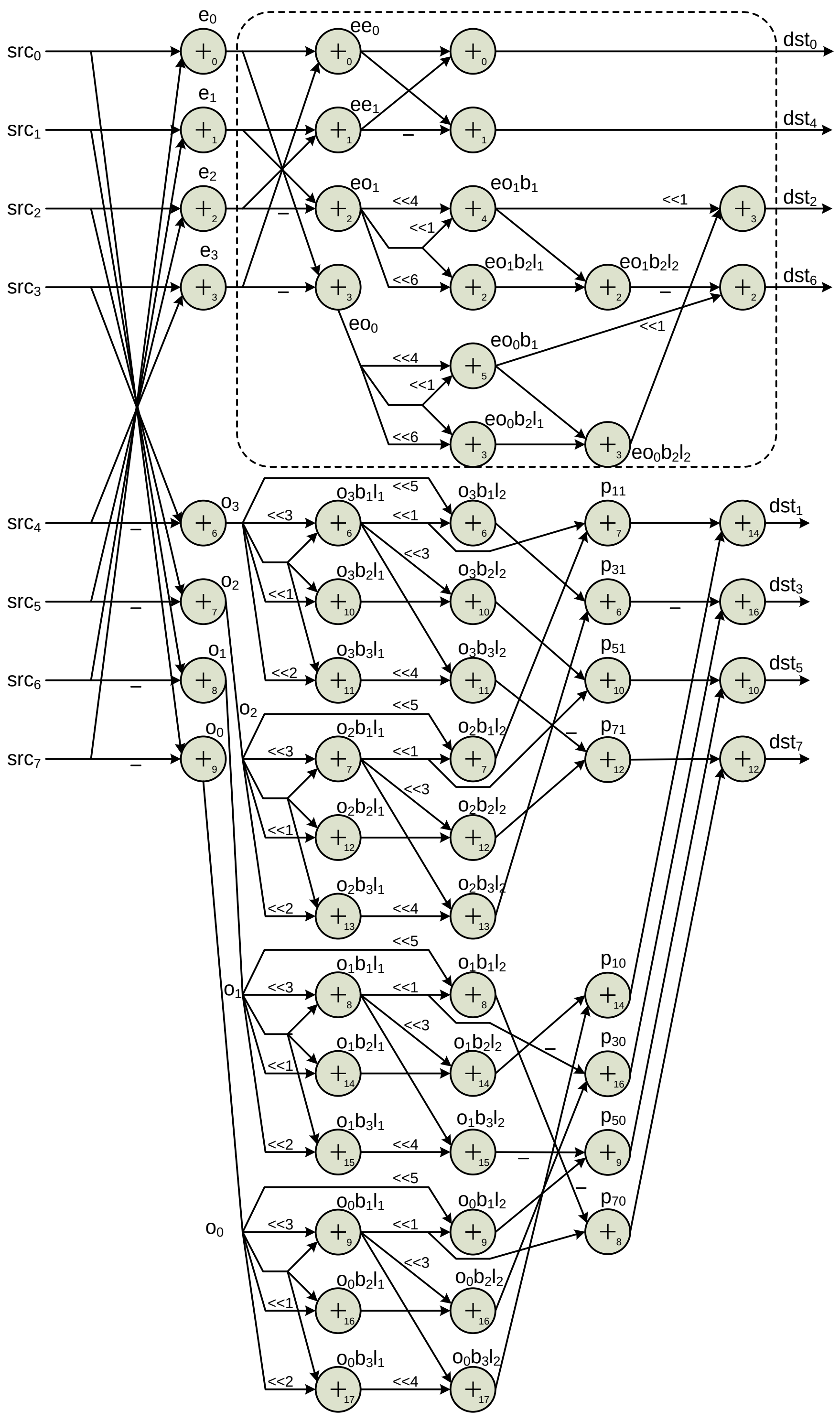


FIGURE 6.20: The proposed 8×8 1-D FIT with resource binding.

TABLE 6.21: Inputs and outputs of each AS through different time slots when performing the proposed 8 × 8 1-D forward transform algorithm. Each cell includes $input_1$ (from register), $input_2$ (from register) and $output$ (to register).

AS	Time 1-2	Time 3-4	Time 5-6	Time 7-8	Time 9-10
AS_0 (R_0)	src_0, src_7, e_0 (R_0)	e_0 (R_0), e_3 (R_3), ee_0 (R_0)	ee_0 (R_0), ee_1 (R_1), dst_0 (R_0)		
AS_1 (R_1)	src_1, src_6, e_1 (R_1)	e_1 (R_1), e_2 (R_2), ee_1 (R_1)	ee_0 (R_0), $-ee_1$ (R_1), $dst_4^{\&}$ (R_1)		
AS_2 (R_2)	src_2, src_5, e_2 (R_2)	e_1 (R_1), $-e_2$ (R_2), eo_1 (R_2)	eo_1 (R_2), $eo_1 \ll 6$ (R_2), $eo_1b_2l_1$ (R_2)	$eo_1b_1 \ll 1$ (R_4), $eo_1b_2l_1$ (R_2), $eo_1b_2l_2$ (R_2)	$eo_0b_1 \ll 1$ (R_5), $-eo_1b_2l_2$ (R_2), dst_6§ (R_2)
AS_3 (R_3)	src_3, src_4, e_3 (R_3)	e_0 (R_0), $-e_3$ (R_3), eo_0 (R_3)	eo_0 (R_3), $eo_0 \ll 6$ (R_3), $eo_0b_2l_1$ (R_3)	$eo_0b_1 \ll 1$ (R_5), $eo_0b_2l_1$ (R_3), $eo_0b_2l_2$ (R_3)	$eo_1b_1 \ll 1$ (R_5), $eo_0b_2l_2$ (R_3), dst_2# (R_2)
AS_4 (R_4)			$eo_1 \ll 4$ (R_2), $eo_1 \ll 1$ (R_2), eo_1b_1 (R_4)		
AS_5 (R_5)			$eo_0 \ll 4$ (R_3), $eo_0 \ll 1$ (R_3), eo_0b_1 (R_5)		
AS_6 (LR_0)	src_3, $-src_4$, o_3 (LRL_0)	$o_3 \ll 3$ (LRL_0), o_3 (LRL_0), $o_3b_1l_1$ (LRL_0)	$o_3 \ll 5$ (LRM_0), $o_3b_1l_1 \ll 1$ (LRL_0), $o_3b_1l_2$ (LRL_0)	$o_3b_1l_2$ (LRL_0), $o_2b_3l_2$ (R_9), P_{31} (LRL_0)	
AS_7 (LR_1)	src_2, $-src_5$, o_2 (LRL_1)	$o_2 \ll 3$ (LRL_1), o_2 (LRL_1), $o_2b_1l_1$ (LRL_1)	$o_2 \ll 5$ (LRM_1), $o_2b_1l_1 \ll 1$ (LRL_1), $o_2b_1l_2$ (LRL_1)	$o_3b_1l_1 \ll 1$ (LRM_0), $o_2b_1l_2$ (LRL_1), P_{11} (LRL_1)	
AS_8 (LR_2)	src_1, $-src_6$, o_1 (LRL_2)	$o_1 \ll 3$ (LRL_2), o_1 (LRL_2), $o_1b_1l_1$ (LRL_2)	$o_1 \ll 5$ (LRM_2), $o_1b_1l_1 \ll 4$ (LRL_2), $o_1b_1l_2$ (LRL_2)	$o_0b_1l_1 \ll 1$ (LRM_3), $-o_0b_1l_2$ (LRL_2), P_{70} (LRL_2)	

32-bit LRM (Most Significant Part of LR). These two parts can be written and read independently to each other. Table 6.21, Table 6.22 and Table 6.23 show inputs and outputs of the ASs through different time slots when performing the proposed 8 × 8 and 4 × 4 1-D forward transform algorithms.

Table 6.22: Inputs and outputs of each AS through different time slots when performing the proposed 8×8 1-D forward transform algorithm. Each cell includes $input_1$ (from register), $input_2$ (from register) and $output$ (to register).

AS	Time 1-2	Time 3-4	Time 5-6	Time 7-8	Time 9-10
AS_9 (LR_3)	src_0, $-src_7$, o_0 (LRL_3)	$o_0 \ll 3$ (LRL_3), o_0 (LRL_3), $o_0b_1l_1$ (LRL_3)	$o_0 \ll 5$ (LRM_3), $o_0b_1l_1 \ll 1$ (LRL_3), $o_0b_1l_2$ (LRL_3)	$o_0b_1l_2$ (LRL_3), $-o_1b_3l_2$ (R_{11}), P_{50} (LRL_3)	
AS_{10} (R_6)		$o_3 \ll 1$ (LRL_0), o_3 (LRL_0), $o_3b_2l_1$ (R_6)	$o_3b_1l_1 \ll 3$ (LRL_0), $o_3b_2l_1$ (R_6), $o_3b_2l_2$ (R_6)	$o_3b_2l_2$ (R_6), $o_2b_1l_1 \ll 1$ (LRM_1), P_{51} (R_6)	P_{50} (LRL_3), P_{51} (R_6), dst_5 (R_6)
AS_{11} (R_7)		$o_3 \ll 2$ (LRL_0), o_3 (LRL_0), $o_3b_3l_1$ (R_7)	$o_3b_1l_1$ (LRL_0), $o_3b_3l_1 \ll 4$ (R_7), $o_3b_3l_2$ (R_7)		
AS_{12} (R_8)		$o_2 \ll 1$ (LRL_1), o_2 (LRL_1), $o_2b_2l_1$ (R_8)	$o_2b_1l_1 \ll 3$ (LRL_1), $o_2b_2l_1$ (R_8), $o_2b_2l_2$ (R_8)	$o_2b_2l_2$ (R_8), $-o_3b_3l_2$ (R_7), P_{71} (R_8)	P_{70} (LRL_2), P_{71} (R_8), dst_7 (R_8)
AS_{13} (R_9)		$o_2 \ll 2$ (LRL_1), o_2 (LRL_1), $o_2b_3l_1$ (R_9)	$o_2b_1l_1$ (LRL_1), $o_2b_3l_1 \ll 4$ (R_9), $o_2b_3l_2$ (R_9)		
AS_{14} (R_{10})		$o_1 \ll 1$ (LRL_2), o_1 (LRL_2), $o_1b_2l_1$ (R_{10})	$o_1b_1l_1 \ll 3$ (LRL_2), $o_1b_2l_1$ (R_{10}), $o_1b_2l_2$ (R_{10})	$o_1b_2l_2$ (R_{10}), $o_0b_3l_2$ (R_{13}), P_{10} (R_{10})	P_{11} (LRL_1), P_{10} (R_{10}), dst_1 (R_{10})
AS_{15} (R_{11})		$o_1 \ll 2$ (LRL_2), o_1 (LRL_2), $o_1b_3l_1$ (R_{11})	$o_1b_1l_1$ (LRL_2), $o_1b_3l_1 \ll 4$ (R_{11}), $o_1b_3l_2$ (R_{11})		
AS_{16} (R_{12})		$o_0 \ll 1$ (LRL_3), o_0 (LRL_3), $o_0b_2l_1$ (R_{12})	$o_0b_1l_1 \ll 3$ (LRL_3), $o_0b_2l_1$ (R_{12}), $o_0b_2l_2$ (R_{12})	$o_0b_2l_2$ (R_{12}), $-o_1b_1l_1 \ll 1$ (LRM_2), P_{30} (R_{12})	P_{30} (R_{12}), $-P_{31}$ (LRL_0), dst_3 (R_{12})
AS_{17} (R_{13})		$o_0 \ll 2$ (LRL_3), o_0 (LRL_3), $o_0b_3l_1$ (R_{13})	$o_0b_1l_1$ (LRL_3), $o_0b_3l_1 \ll 4$ (R_{13}), $o_0b_3l_2$ (R_{13})		

TABLE 6.23: Inputs and outputs of each AS through different time slots when performing the proposed 4×4 1-D forward transform algorithm. Each cell includes $input_1$ (from register), $input_2$ (from register) and $output$ (to register).

AS	Time 1-2	Time 3-4	Time 5-6	Time 7-8
AS_0 (R_0)	src_0, src_3, e_0 (R_0)	e_0 (R_0), e_1 (R_1), dst_0 (R_0)		
AS_1 (R_1)	src_1, src_2, e_1 (R_1)	e_0 (R_0), $-e_1$ (R_1), dst_2 (R_1)		
AS_2 (R_2)	src_1, $-src_2$, e_2 (R_2)	o_1 (R_2), $o_1 \ll 6$ (R_2), $o_1b_2l_1$ (R_2)	$o_1b_1 \ll 1$ (R_4), $o_1b_2l_1$ (R_2), $o_1b_2l_2$ (R_2)	$o_0b_1 \ll 1$ (R_5), $-o_1b_2l_2$ (R_2), dst_1 (R_2)
AS_3 (R_3)	src_0, $-src_3$, e_3 (R_3)	o_0 (R_3), $o_0 \ll 6$ (R_3), $o_0b_2l_1$ (R_3)	$o_0b_1 \ll 1$ (R_5), $o_0b_2l_1$ (R_3), $o_0b_2l_2$ (R_3)	$o_1b_1 \ll 1$ (R_5), $o_0b_2l_2$ (R_3), dst_3 (R_2)
AS_4 (R_4)		$o_1 \ll 4$ (R_2), $o_1 \ll 1$ (R_2), o_1b_1 (R_4)		
AS_5 (R_5)		$o_0 \ll 4$ (R_3), $o_0 \ll 1$ (R_3), o_0b_1 (R_5)		

For example, in Table 6.21, Adder/Subtractor AS_0 (connected to R_0), at time 1-2 when running the proposed 8×8 algorithm, has two inputs src_0 and src_7 as shown in Figure 6.20. Its output after time 1-2 is e_0 (Figure 6.20), which is then stored back in R_0 to be used in next time slots. At time 3-4, it has two inputs e_0 and e_3 (Figure 6.20) and one output ee_0 (Figure 6.20). At this time, e_0 and e_3 values have been already stored in R_0 and R_3, respectively, since time 1-2. After that, the addition result ee_0 (Figure 6.20) is also stored back in R_0 to be used in next time slots.

As can be seen from Table 6.21, Table 6.22 and Table 6.23, each time AS_0 can choose one of four input sets, including (1) src_0 and src_7 in time 1-2 of the 8×8 algorithm; (2) R_0 and R_3 in time 3-4 of the 8×8 algorithm; (3) R_0 and R_1 in time 5-6 of the 8×8 algorithm; or (4) src_0 and src_3 in time 1-2 of the 4×4 algorithm. Hence, a 32-bit MUX 4-1 can be used at the two inputs of AS_0. The numbers

TABLE 6.24: Number of possible input sets for adder/subtractors in the proposed architecture.

AS	Input sets	AS	Input sets	AS	Input sets	AS	Input sets	AS	Input sets
AS_0	4	AS_4	1	AS_8	4	AS_{12}	4	AS_{16}	4
AS_1	4	AS_5	1	AS_9	4	AS_{13}	2	AS_{17}	2
AS_2	6	AS_6	4	AS_{10}	4	AS_{14}	4		
AS_3	6	AS_7	4	AS_{11}	2	AS_{15}	2		

TABLE 6.25: MUX types and quantities for the ASs in the proposed architecture.

MUX type	Quantity (each AS has 2 inputs)
32-bit 6-1 MUX	$2 \times 2 = 4$
32-bit 4-1 MUX	$10 \times 2 = 20$
32-bit 2-1 MUX	$4 \times 2 = 8$

of input sets for all the Adder/Subtractors from AS_0 to AS_{17} can be found in Table 6.24.

Based on Table 6.24, it is clear that two 32-bit 6-1 MUXs, ten 32-bit 4-1 MUXs and two 32-bit 2-1 MUXs are needed for all the eighteen ASs in the proposed architecture (Table 6.25).

6.4.2.5 1-D Transform Data Path Design

The proposed 1-D design inputs, processes and outputs four and eight pixels simultaneously in the 4×4 and 8×8 transforms, respectively. Each input is 16-bit deep. Therefore, the proposed shared 1-D architecture needs $8 \times 16 = 128$ input pins and 128 output pins. However, as all internal operations of the standard requires 32-bit depth, the 128 I/O pins are converted to 256 pins to facilitate internal processing.

Figure 6.21 shows the proposed architecture design for the proposed 1-D 4×4 and 8×8 forward transform. On the right side of the figure, registers R_0 to R_{13} and LR_0 to LR_3 together with source register from src_0 to src_7 are shown. Adder/Subtractors from AS_0 to AS_{17} can be found at the bottom of the figure. Above the ASs, there are thirty-two multiplexors, which are listed in Table 6.25. The inputs of those multiplexors are connected to the above registers, while their outputs are connected to the inputs of the ASs. In addition, the outputs of those ASs are connected to the above registers in order to store addition/subtraction results. All the ASs, multiplexors and registers are controlled through control signals generated by a Control Unit.

6.4.2.6 Micro-Code and Control Signal Optimization

The Control Unit is an FSM generating control signals to all components in the Data Path. The Adder/Subtractors need ENable signals $enas$, and function indication signal sub to indicate addition or subtraction function needed to perform. The Multiplexors require Select signals ms. Long registers LR_0 to LR_3 also needs a control signal to know when to shift and store data.

Table 6.26 and Table 6.27 show the values of the enable signals for eighteen ASs in the Data Path design. They are determined based on Table 6.21, Table 6.22 and Table 6.23. Table 6.28 and Table 6.29 show the values of sub signals for eighteen ASs to determine whether addition or subtraction is required. Table 6.30 and Table 6.31 shows the control signal to the MUXs.

As some of the control signals for the Control Unit are similar, or some signals are always equal to '1' or '0', optimization can be done to reduce the redundancy. As can be seen in Table 6.26, $enas_0 = enas_1$, $enas_2 = enas_3$, $enas_4 = enas_5$,

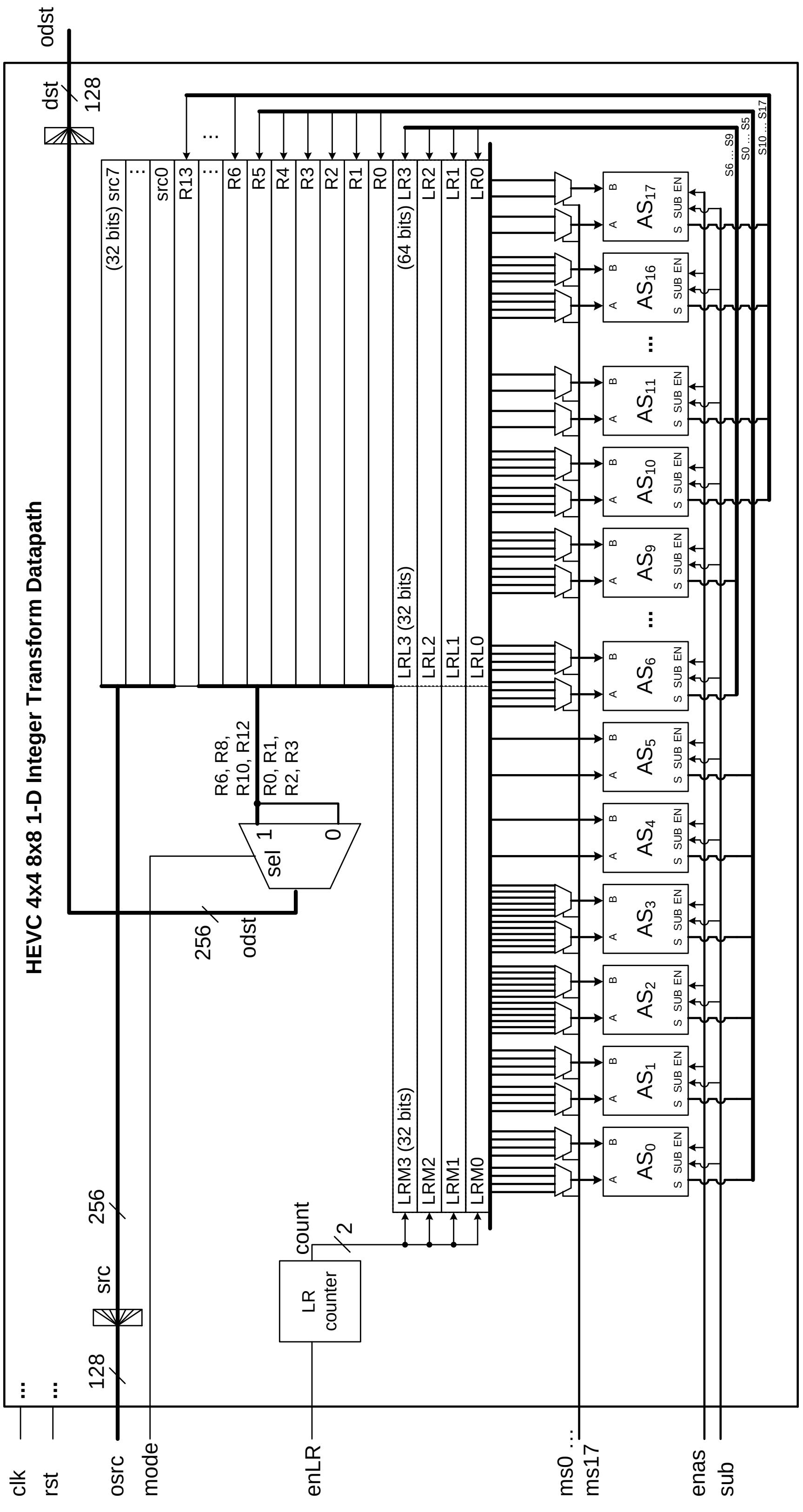


FIGURE 6.21: The proposed data path design for the proposed 1-D 4 × 4 and 8 × 8 forward fransform algorithms.

TABLE 6.26: Enable signals of the ASs in the proposed 8 × 8 FIT algorithms, *enas*, through different system stages.

enas at	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Time 1	0	0	0	0	0	0	0	0	1	1	1	1	0	0	1	1	1	1
Time 2	0	0	0	0	0	0	0	0	1	1	1	1	0	0	1	1	1	1
Time 3	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	1	1	1
Time 4	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	1	1	1
Time 5	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Time 6	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Time 7	0	1	0	1	0	1	0	1	1	1	1	1	0	0	1	1	0	0
Time 8	0	1	0	1	0	1	0	1	1	1	1	1	0	0	1	1	0	0
Time 9	0	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0
Time 10	0	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0

TABLE 6.27: Enable signals of the ASs in the proposed 4 × 4 FIT algorithms, *enas*, through different system stages.

enas at	5	4	3	2	1	0
Time 1	0	0	1	1	1	1
Time 2	0	0	1	1	1	1
Time 3	1	1	1	1	1	1
Time 4	1	1	1	1	1	1
Time 5	0	0	1	1	0	0
Time 6	0	0	1	1	0	0
Time 7	0	0	1	1	0	0
Time 8	0	0	1	1	0	0

TABLE 6.28: Function indication signals of the ASs in the proposed 8 × 8 FIT algorithms, *sub*, through different system stages. If *sub* equals to 0, the corresponding AS performs addition. Otherwise, it performs subtraction.

sub at	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Time 1	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0
Time 2	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0
Time 3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0
Time 4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0
Time 5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
Time 6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
Time 7	0	1	0	0	0	1	0	0	1	1	0	0	0	0	0	0	0	0
Time 8	0	1	0	0	0	1	0	0	1	1	0	0	0	0	0	0	0	0
Time 9	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
Time 10	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0

Table 6.29: Function indication signals of the ASs in the proposed 4 × 4 FIT algorithms, *sub*, through different system stages. If *sub* equals to 0, the corresponding AS performs addition. Otherwise, it performs subtraction.

sub at	5	4	3	2	1	0
Time 1	0	0	1	1	0	0
Time 2	0	0	1	1	0	0
Time 3	0	0	0	0	1	0
Time 4	0	0	0	0	1	0
Time 5	0	0	0	0	0	0
Time 6	0	0	0	0	0	0
Time 7	0	0	0	1	0	0
Time 8	0	0	0	1	0	0

Table 6.30: Select signals of the MUXs in the proposed 8 × 8 FIT algorithms, *ms*, thought different system stages (ms_i is used as the select signal for two multiplexors at the two inputs of AS_i).

ms at	17	16	15	14	13	12	11	10	9	8	7	6	3	2	1	0
Time 1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Time 2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Time 3	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
Time 4	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
Time 5	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2
Time 6	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2
Time 7	0	2	0	2	0	2	0	2	3	3	3	3	3	3	0	0
Time 8	0	2	0	2	0	2	0	2	3	3	3	3	3	3	0	0
Time 9	0	3	0	3	0	3	0	3	0	0	0	0	4	4	0	0
Time 10	0	3	0	3	0	3	0	3	0	0	0	0	4	4	0	0

Table 6.31: Select signals of the MUXs in the proposed 4 × 4 FIT algorithms, *ms*, thought different system stages (ms_i is used as the select signal for two multiplexors at the two inputs of AS_i).

ms at	3	2	1	0
Time 1	1	1	1	1
Time 2	1	1	1	1
Time 3	2	2	2	2
Time 4	2	2	2	2
Time 5	3	3	0	0
Time 6	3	3	0	0
Time 7	4	4	0	0
Time 8	4	4	0	0

$enas_6 = enas_7$, $enas_8 = enas_9$, $enas_{10} = enas_{12} = enas_{14} = enas_{16}$, and $enas_{11} = enas_{13} = enas_{15} = enas_{17}$. Therefore, we only need to keep *enas* at positions 0, 2, 4, 6, 8, 10 and 11 for the 8×8 transform. Similarly, from Table 6.27, we only need to keep *enas* at positions 0, 2 and 4 for the 4×4 transform. For *sub* signals (Table 6.28 and Table 6.29), we only need to keep values at positions 1, 2, 3, 6, 8, 12, 16 for 8×8 and positions 1, 2, 3 for the 4×4 transform. *sub* signals at positions 0, 4, 5, 10, 11, 13, 14, 15 and 17 can be omitted since they are always equal to '0'. For MUX select signals (Table 6.30 and Table 6.31), we only need to keep values at positions 0, 2, 6, 10 and 11 for the 8×8, and 0 and 2 for the 4×4 transform (Figure 6.22). This microcode and control signal optimization can reduce the circuit area due to the less number of wires connected between Control Unit and Data Path blocks.

6.4.2.7 Proposed 1-D Transform Architecture

Figure 6.22 shows the proposed 1-D shared transform architecture for the 4×4 and 8×8 transforms with the proposed data path in Section 6.4.2.5 and a Control Unit. The microcode and control signals (*ms*, *enas* and *sub*) are minimized to reduce the circuit area.

6.4.2.8 Pipeline Scheduling for 2-D Transforms

A pipeline mechanism is deployed for the 2-D transforms among the input process, the output process and the two 1-D transforms (Figure 6.23). The system starts at t_0 to input the first eight pixels (row 1.0) to an input buffer block (IB) in sixteen cycles. The IB collects, merges all half-pixels and sends them to 1-D integer transform block 1 (IT1) at t_1. At t_1, the IB collects the second eight pixels (row

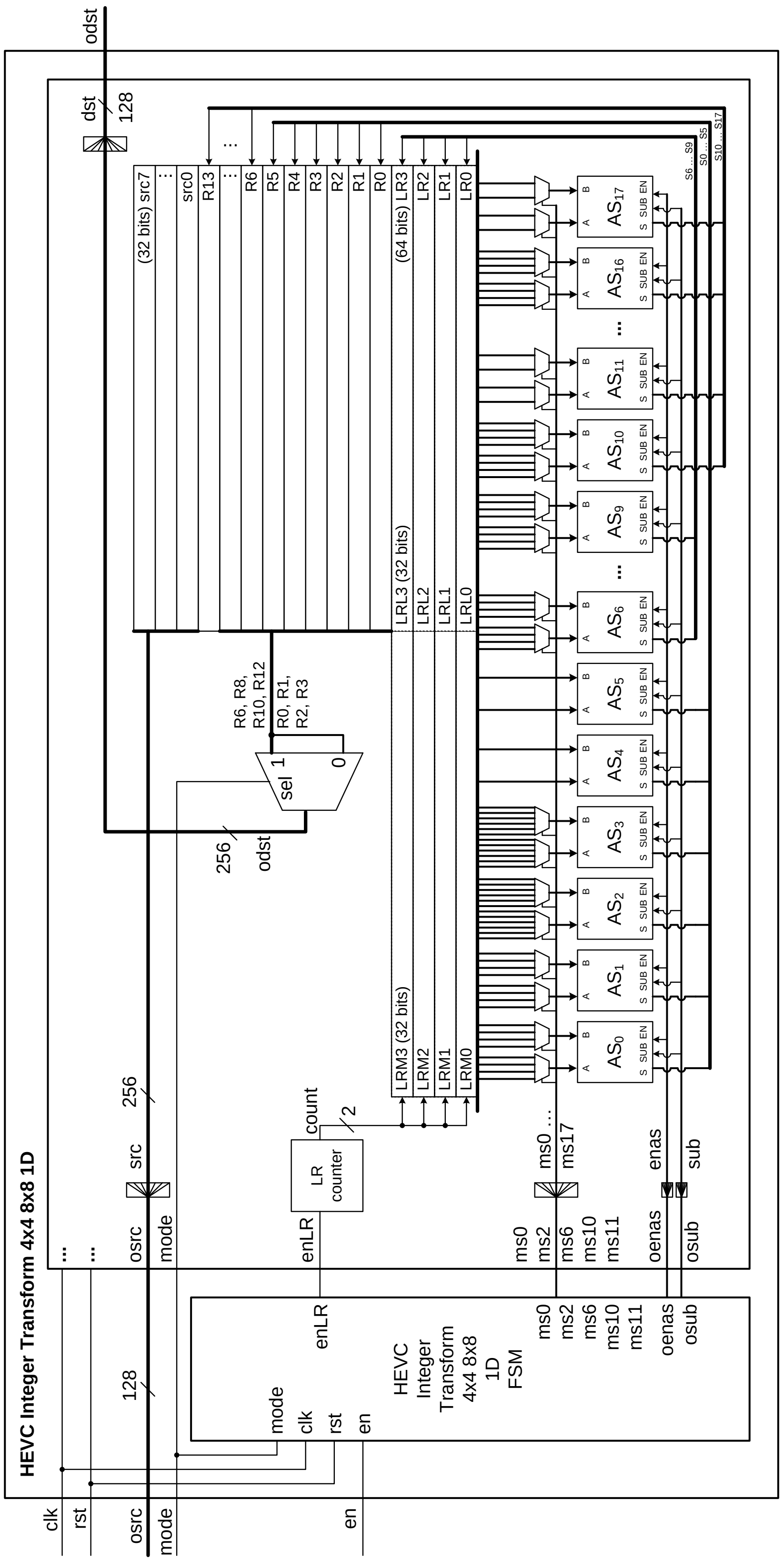


FIGURE 6.22: The proposed architecture design for the proposed 1-D 4×4 and 8×8 forward fransform algorithms.

1.1), while IT1 transforms the first eight pixels (row 1.0). Then, it transfers the result to Transpose RAM 1. The process is repeated. At t_8, eight pixels of row 2.0 are started to input to the IB. At the same time, IT1 starts to process row 1.7. At t_9, the transformation of eight rows of block 1 from row 1.0 to 1.7, have completed and the result is in Transpose RAM 1. At the same time, eight pixels of row 2.1 are started to input to the IB. IT1 starts to process row 2.0. The IT2 starts to transform column 1.0 from Transpose RAM 1 and transfer the results to the output buffer (OB). As RAM 1 is now storing transformed data of block 1, when IT1 finishes processing row 2.0, it will store data into Transpose RAM 2. From t_{10} onwards, all hardware blocks work simultaneously. The OB receives a package of eight pixels from IT2 and output half-pixel at a cycle. The number of cycles needed to transform each 8×8 block is $16 \times 8 = 128$ cycles, where each cycle is just 16-bit addition time long.

6.4.2.9 Proposed 2-D Transform Architecture

Figure 6.24 shows the architecture of the proposed 2-D 4×4 and 8×8 transforms. The architecture consists of one input buffer (IB), one output buffer (OB), two 1-D transform blocks (IT1 and IT2) and two RAMs for data transpose between two times of 1-D transform.

Figure 6.25 shows the detailed timing diagram of the proposed system. Figure 6.26 and Figure 6.27 are the enlarged parts of Figure 6.25.

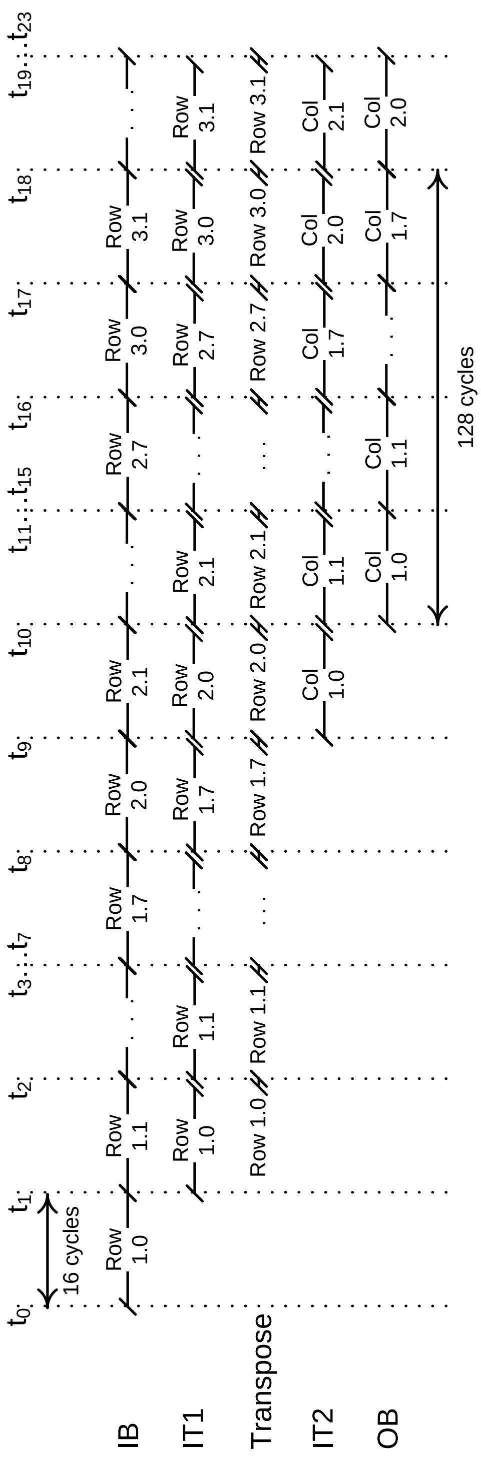


FIGURE 6.23: Pipeline scheduling between input buffer (OB), output buffer (OB), integer transform (IT) blocks and transpose RAMs.

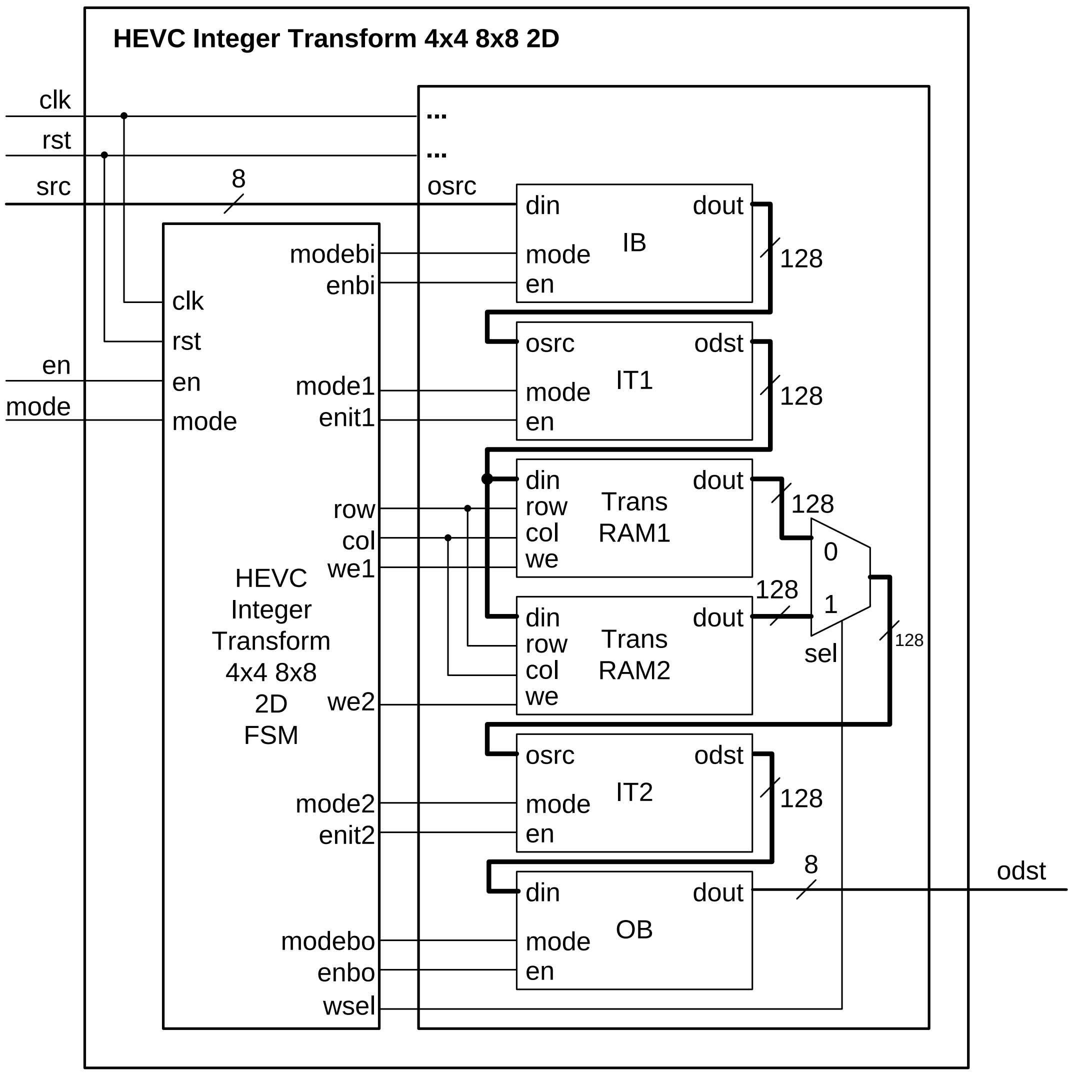


Figure 6.24: The proposed 2-D integer transform.

6.4.3 Experimental Results and Discussion

6.4.3.1 Experimental Results

The architecture design is implemented in Verilog using UMC 65nm Low Leakage (LL) RVT technology at 1.2V. It is functionally verified using ModelSim, synthesized using Design Compiler, logically verified using NCVerilog, floor-planned and place-and-routed using Encounter, post-place-and-route-verified with *gds* file

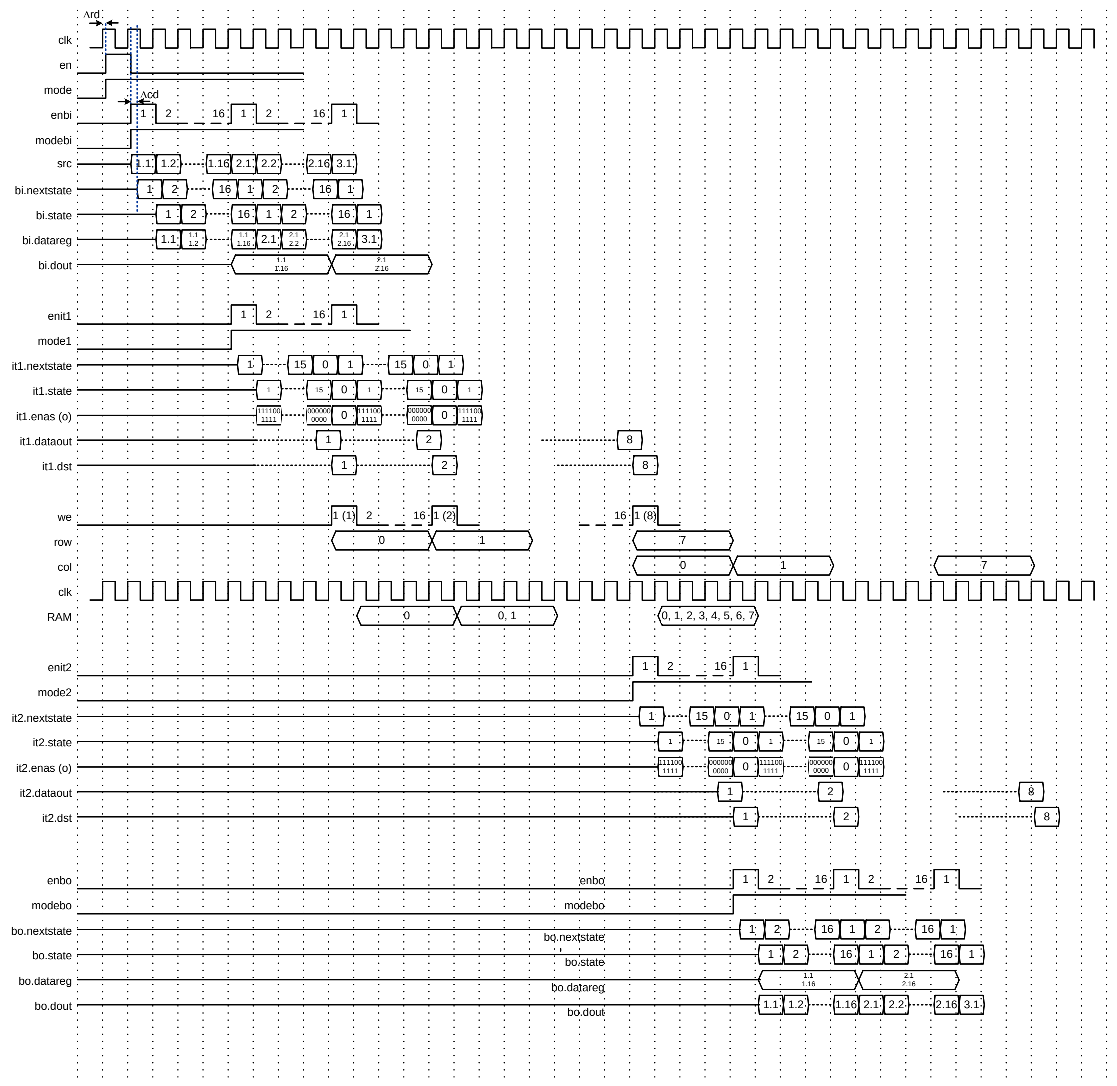


FIGURE 6.25: Timing diagram of the proposed architecture.

using NCVerilog. The design has been sent for fabrication. The testing PCB has also been designed and fabricated.

Figure 6.28 shows the layout implementation of the architecture. Figure 6.29 and Figure 6.30 shows the physical layout with I/O pads and die photograph of the IC fabricated using UMC 65nm Low Leakage (LL) technology, respectively. The number of I/O pads is sixteen and the gate counts include an Input Buffer, an Output Buffer, two Transpose RAMs and two integer transform blocks are 54.7K.

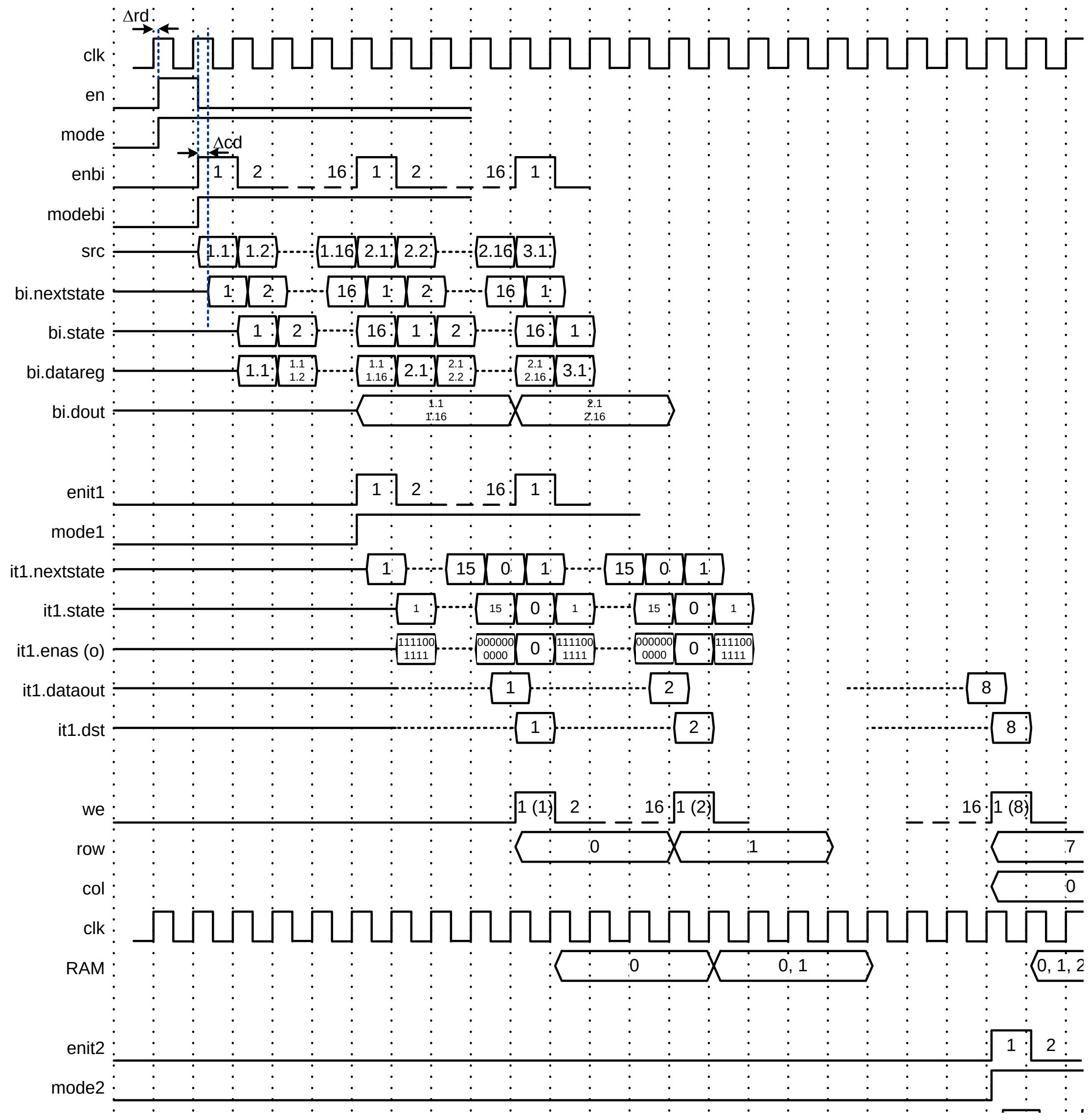


FIGURE 6.26: IT1-enlarged timing diagram of the proposed architecture.

The IC total power consumption is 8.94 mW. The maximum operating frequency that the implementation can achieve is 500 MHz. The architecture can support the transforms for up to Quad-Full High Definition (QFHD) videos, i.e., resolution of 3840 $\times$ 2160 pels, at the progressive scan frequency of 30 Hz (30 frames per second).

Table 6.32, Figure 6.31 and Figure 6.32 show the area and power breakdown of the proposed architecture. As can be seen, the 1-D transforms only consumes 17.2

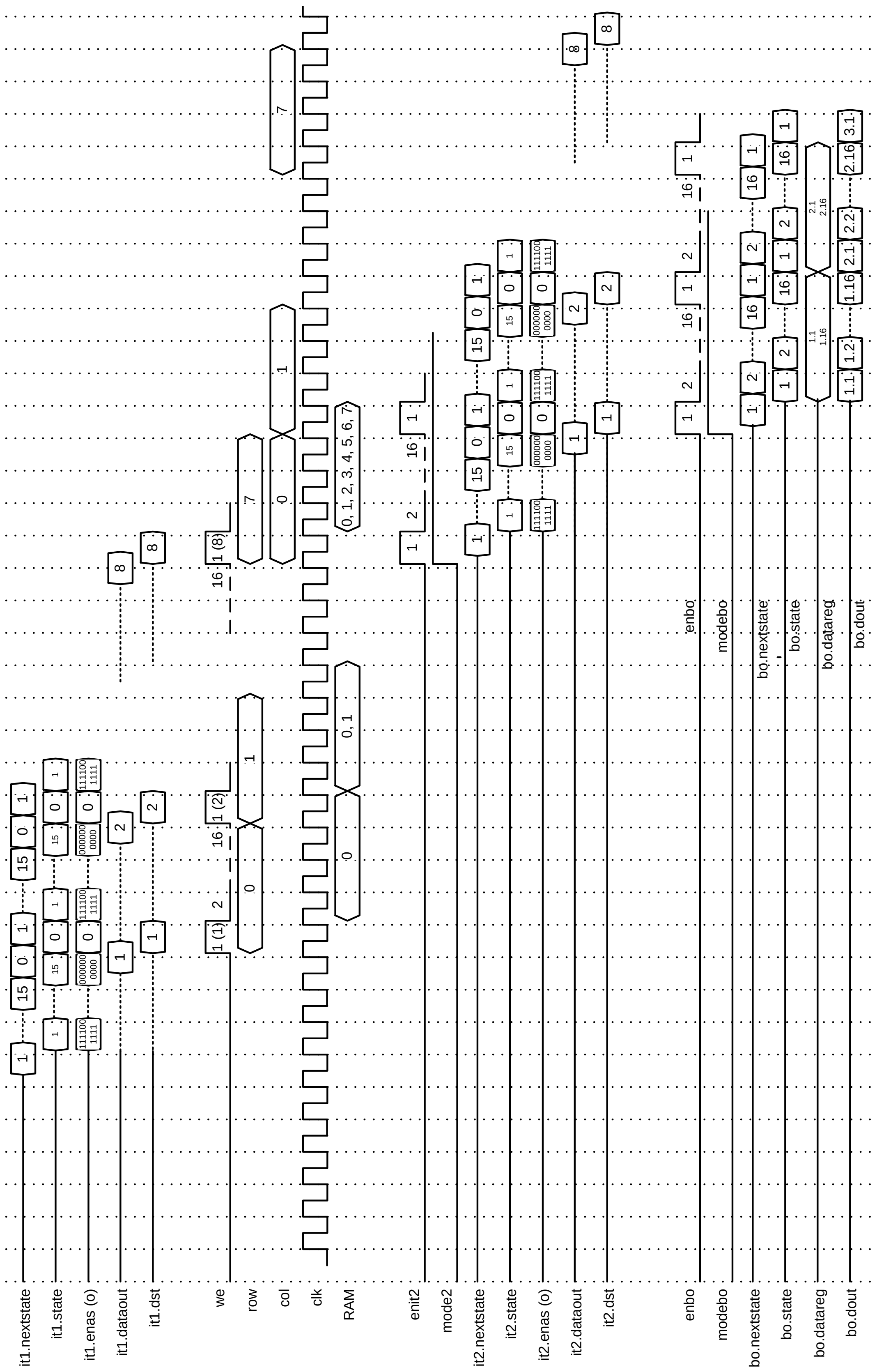


FIGURE 6.27: IT2-enlarged timing diagram of the proposed architecture.

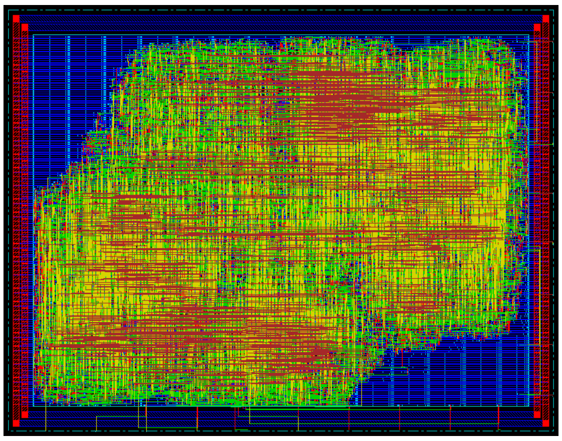


Figure 6.28: Layout of HEVC 4×4 and 8×8 transform architecture.

Table 6.32: Area and power consumption breakdown of the proposed architecture.

Modules	Gate count (gates)	Area (%)	Power (mW)	Power (%)
IB	1643.75	3	0.212	2.4
OB	1203.75	2.2	0.433	4.8
IT1	17203	31.5	2.512	28.1
IT2	17211.5	31.5	2.181	24.4
TRANSRAM1	8549.75	15.6	1.773	19.8
TRANSRAM2	8545.61	15.6	1.769	19.8
IT-FSM	342	0.6	$6.18E-02$	0.7
IT	54699.36	100	8.9418	100

Kgates and 2.3 mW in average, which is 31% and 26% in average of the total area and power consumption. The transpose RAMs occupy a large portion, 32%, of the total area and consumes 40% of the IC power. The input and output buffer area and power consumption are rather small, at 5% and 7% in total, respectively.

Figure 6.33 shows the PCB design for the fabricated IC.

6.4.3.2 Discussion

Table 6.33 shows the details of the proposed implementation in comparison to the implementation of Martuza and Wahid (2012). Gate count of a design is the

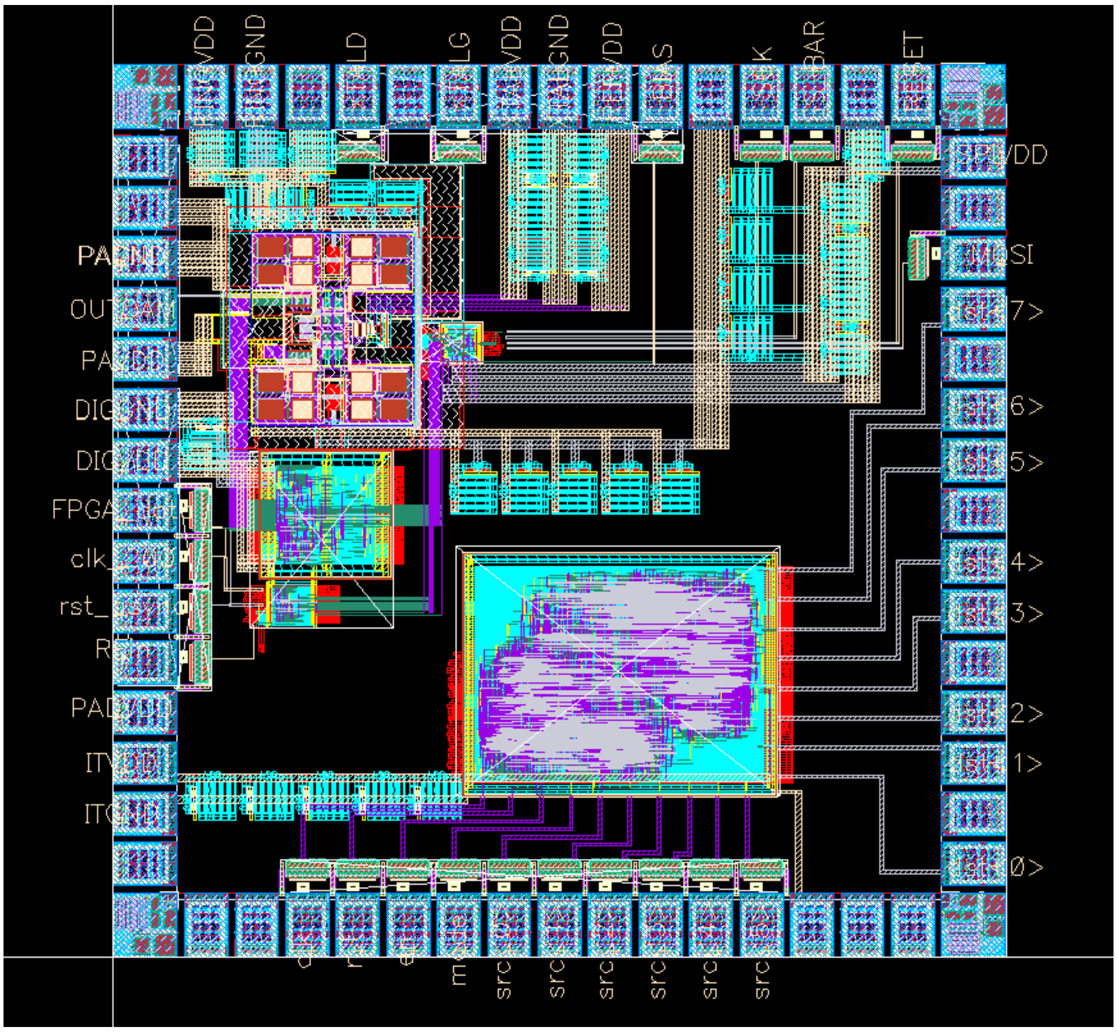


FIGURE 6.29: Layout with I/O pads of HEVC 4 × 4 and 8 × 8 transform architecture.

design area normalized by the area of the smallest 2-input NAND gate. In the UMC 65nm technology, the size of the smallest 2-input NAND gate is 0.8μm width $\times$1.8μm height. I/O throughput is the number of pixels that the architecture can input or output at the same time. Throughput is the number of pixels processed in an unit of time, where the time includes the I/O time and processing time. The throughput unit is pixels per cycle or pixels per second. When comparing the designs whose I/O pin counts are small, i.e., I/O time is longer than processing

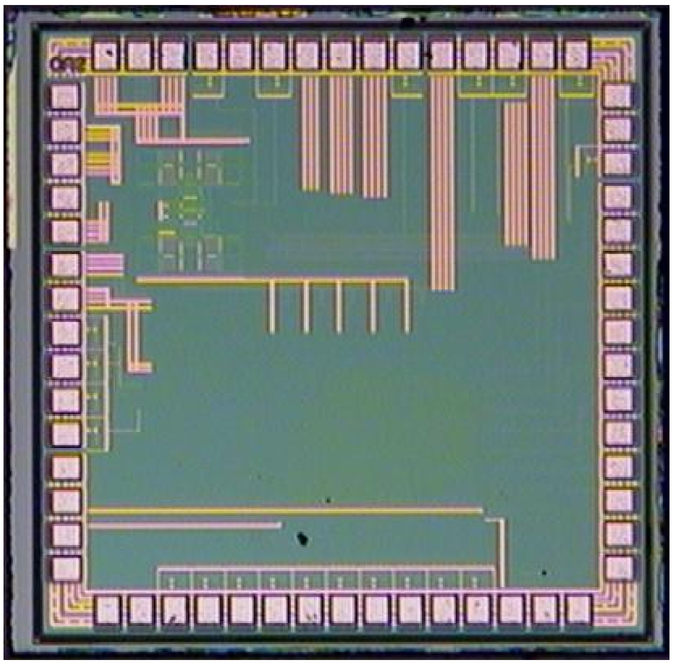


Figure 6.30: Die photograph of HEVC 4×4 and 8×8 transform architecture.

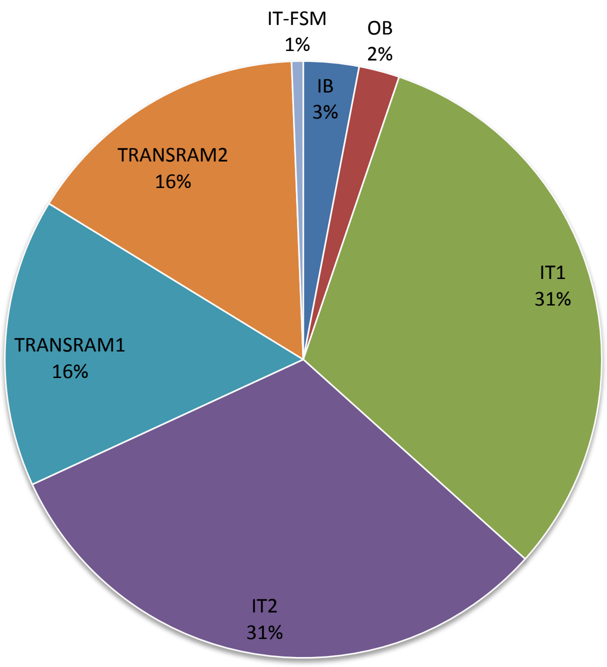


Figure 6.31: Area breakdown.

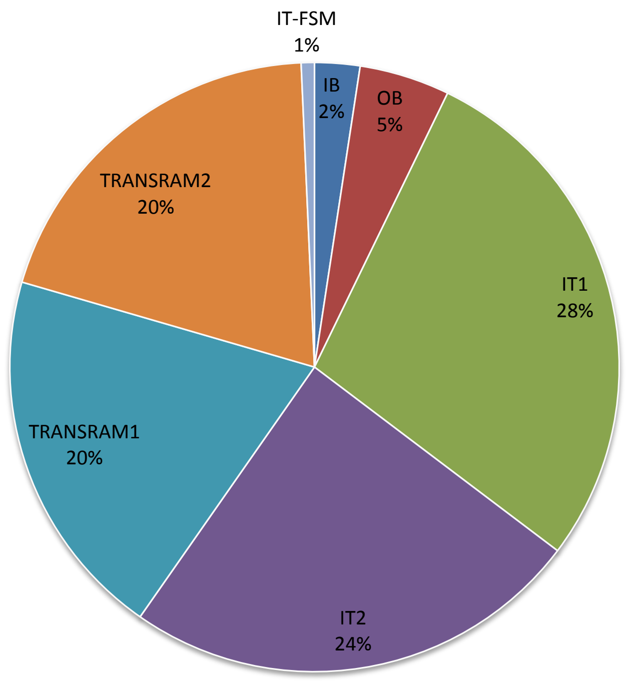
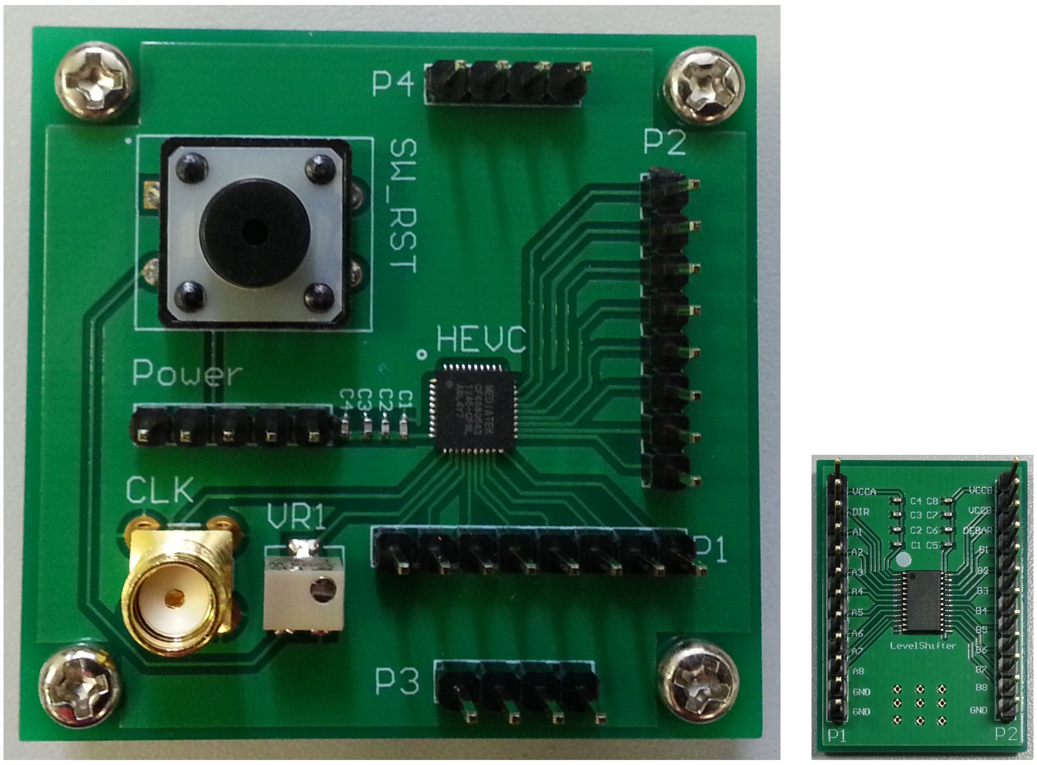


Figure 6.32: Power breakdown.

time, the throughput need to be normalized by the number of I/O pins, since the number of I/O pins plays a crucial role in limiting the design throughput. As power of a design depends on the process technology and voltage, it is also normalized by voltage square for a fair comparison. Therefore, although the Martuza and Wahid (2012) mapped the design to CMOS 0.18μm, which is different with the technology that the proposed design is implemented, fair comparisons still can be achieved because throughput, area and power are all normalized.

As can be seen from the table, Martuza and Wahid (2012) proposed an 1-D 8×8 inverse transform architecture for HEVC, while the proposed architecture is a 2-D 4 and 8×8 forward transform architecture. We use our proposed 1-D architecture including its controller to compare to the design of Martuza and Wahid (2012)(MD). Thanks to the period optimization using 2-cycle adders, the proposed 1-D architecture implementation can achieve a normalized throughput



(a) (b)

FIGURE 6.33: PCB design for the fabricated HEVC 4 × 4 and 8 × 8 IC. (a) PCB and (b) Level shifter.

of 500 Mpps, which is 2.4 times as much as that of MD. Even in terms of direct throughput, the proposed design throughput is 18% more than that of the MD. It should be noted that the reported frequency of the MD achieved only when synthesizing the design to the technology without any post-synthesized verification or post-place-and-route verification.

Although the proposed design requires only eighteen ASs compared to thirty-two adders of the MD, the proposed 1-D architecture consumes 17.2 Kgates, 1.4 times of the MD gate count. Table 6.34 and Figure 6.34 show an area breakdown of the proposed 1-D transform implementation. As can be seen, the ASs occupy 10.4 Kgates (60% of the 1-D area), while the registers and MUXs occupy 6.6 Kgates (38% of the 1-D area). Therefore, the overhead area is basically due to the added

TABLE 6.33: The proposed architecture implementation achievement in comparison to that of other architectures.

Architectures	Martuza and Wahid (2012)	The proposed 1-D architecture	The proposed 2-D architecture
Technonogy	CMOS 0.18μm	UMC 65nm	UMC 65nm
Voltage	1.8 V	1.2 V	1.2 V
Total I/O pins	32 pins	16 pins	16 pins
Transform type	1-D 8×8 Inverse, HEVC	1-D 4×4, 8×8 Forward, HEVC	2-D 4×4, 8×8 Forward, HEVC
Number of adders in 1-D	32 adders	18 ASs	18 ASs
Frequency	211.4 MHz	500 MHz	500 MHz
Gate count (Gates)	12.3K + 4K standard cells	17.2K	54.7K
Power	11.7 mW	2.3 mW	8.94 mW
I/O throughput (pels/cycle)	1 ppc	0.5 ppc	0.5 ppc
Throughput (pels/cycle)	1 ppc	0.5 ppc	0.5 ppc
Throughput (pels/s)	211.4 Mpps	250 Mpps	250 Mpps
Normalized Throughput(pels/s) (32 pins)	211.4 Mpps	500 Mpps	500 Mpps
Normalized Power (1.2V)	5.2 mW	2.3 mW	8.94 mW
Max resolution	Full HD (1920×1080) 30p	QFHD (3840×2160) 30p	QFHD (3840×2160) 30p

registers and MUXs for the pipelining mechanisms. It also should be noted that Martuza and Wahid (2012) reported their design gate count in two different places: at one place in the script, it is 12.3K and 4K standard cells, and in a table, it is 12.3K in a table. We use 12.3 Kgates to compare.

From Table 6.33, we also see that the proposed 1-D architecture implementation consumes a power of 2.3mW, which is only 44% of that of the MD. The maximum resolution that the proposed 2-D architecture can deal is quad-full high definition with the resolution of 3840×2160 pels at the progressive scanning frequency of 30

Table 6.34: Area breakdown of the proposed 1-D transform architecture.

	Gate count (Kgates)	%
ASs	10.4	60.5
Registers	3.7	21.5
FSM	0.2	1.5
MUXs	2.9	16.5

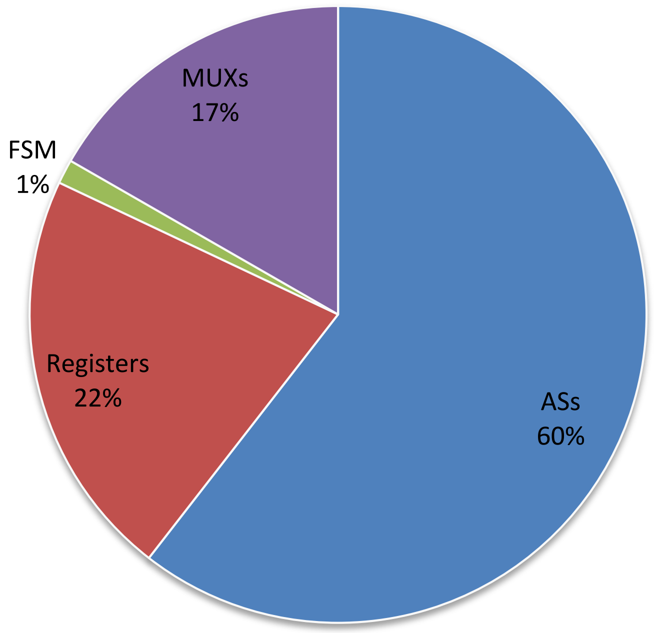


Figure 6.34: Area breakdown of the proposed 1-D transform architecture.

frames per second. This is four times of the resolution that Martuza and Wahid (2012) reported to be able to process. However, their design is 1-D transform. Therefore, for 2-D transforms, Martuza and Wahid (2012)'s design only can process up to 1280 × 800 pels. Hence, the maximum resolution that the proposed 2-D architecture can process is eight times as large as that of the Martuza and Wahid (2012)'s design.

6.5 Summary

In this chapter, we have proposed (1) a novel optimization method to automatically minimize complexity, running time and resource consumption for scalar-multiplication-containing algorithms; (2) novel 4×4 and 8×8 hardware-oriented fast and low-cost integer transform algorithms for HEVC; and (3) a high-throughput and area-efficient transform architecture for the HEVC 4×4 and 8×8 transforms.

The proposed optimization method can be applied to algorithms which contain scalar multiplications and the algorithms' operations can be performed in parallel. The method includes three optimization levels which are corresponding to two algorithms and one scheme to sequently optimize the scalar multiplications' implementation. In the first level of optimization, the proposed complexity optimization algorithm is applied to find the least complex multiplication-to-addition conversion result (MACR) for the multiplications. In the second optimization level, the proposed timing optimization scheme generates the shortest adding tree for the minimized running time and lists all possible operation orders corresponding to the tree (STOOS). While the first two optimization levels are for one by one multiplication, the last optimization level is for all multiplications in the scalar multiplications. In this level, the proposed resource optimization algorithm is applied to find the best operation order, which provides the largest shared regions among the multiplications to enable hardware utilization. The experimental results show that the speed of the automatically generated implementation by the method for the scalar multiplications increases by about 90%, while the cost reduces by 90% compared to the implementation by multiplications. Compared to the conventional multiplication-free implementation, the speed of the generated implementation increases by 33% while the cost reduces from 25 to 33%. It is also proved that the method is computationally feasible to be applied to all the Partial Butterfly algorithms from 4×4 to 32×32.

By applying the proposed optimization method for the scalar multiplications in the Partial Butterfly algorithms and based on the idea of sharing resource among different transform sizes, we have proposed novel 4×4 and 8×8 hardware-oriented fast and low-cost integer transform algorithms for HEVC. The implemental results show that the proposed algorithms can fully perform the 4×4 and 8×8 1-D transforms after 4 and 5 addition/subtractions, respectively. Compared to the original Partial Butterfly algorithms and the conventional sequence multiplication-free Partial Butterfly algorithms, the speed of the proposed algorithms increases by 75% and 20%, respectively. By using fourteen and fifty-eight ASs for the 4×4 and 8×8 transforms respectively, the proposed algorithms requires around 87% and 16% less resource than the original and the conventional sequence/parallel multiplication-free Partial Butterfly algorithms, respectively. Compared to the reported algorithms in the literature for HEVC, the speed of the proposed 8×8 algorithm is 29% and 17% faster than that of Martuza and Wahid (2012) and Rithe et al. (2012)'s 8×8 algorithms, respectively. Its cost reduces by 19% and 3% compared to that of Martuza and Wahid (2012) and Rithe et al. (2012)'s algorithms, respectively. The proposed 4×4 algorithm is as fast as the Rithe et al. (2012)'s algorithm, but consumes 22% less resource.

We have developed a high-throughput and area-efficient transform architecture for the HEVC 4×4 and 8×8 transforms in this chapter. The architecture is under a very tight constraint on the I/O pin count of sixteen pixels. We have also introduced several techniques to achieve the requirement of high throughput for HEVC under such constraint. They are (1) multi-cycle adders to reduce system period, consequently, increase throughput; (2) multi-stage registers for pipelining mechanism; (3) pipelining mechanism with two levels for the 1-D transforms and 2-D transforms; (4) resource binding to reduce cost; and (5) micro-code and control signals optimization to reduce circuit area. The architecture design is implemented in Verilog using UMC 65nm Low Leakage (LL) RVT technology at 1.2V with multi-level of verifications till post-place-and-route-verification. Then the design is sent for Chip fabrication. Testing PCB is also designed and fabricated. Thanks to the techniques, the proposed architecture can support the transforms for up to Quad-Full High Definition (QFHD) videos, i.e., resolution of 3840×2160 pels, at the progressive scan frequency of 30 Hz. This is eight times as large as that of the Martuza and Wahid (2012)'s design with a trade-off of the increment in circuit area of 40% for 1-D architecture. The proposed architecture consumes only 44% of that of the Martuza and Wahid (2012)'s design.

Chapter 7

Conclusions and Future Works

7.1 Conclusions

Integer transform, which is based on the Discrete Cosine Transform (DCT) with the avoidance of the mismatch problem and the reduction of the computational complexity, is one of the most important coding tools in both H.264/AVC, the latest video compression standard, and an emerging High Efficiency Video Coding (HEVC), which will be finalized in early 2013.

While the H.264/AVC high profiles support higher-fidelity videos for application areas like professional film production, video post production, or high-definition TV/DVD, HEVC supports beyond-full high definition videos or 3-D, multi-view applications. Therefore, designs of all the modules including the integer transforms in H.264/AVC or HEVC are challenging to achieve high throughputs. For hardware implementation, area efficiency is needed to reduce design cost. In addition, integration, reusability and testability of hardware designs should be also considered.

In this thesis, we have addressed the above challenges in integer transform designs for H.264/AVC and HEVC through four integer transform designs with two design levels: *algorithm* and *architecture*. While the first three designs address the challenges of H.264/AVC at the *architecture* level, the forth design addresses the challenges of HEVC at both *algorithm* and *architecture* levels. The first two designs are portable inverse integer transform designs for H.264/AVC. While the first design addresses portability and area efficiency, the second design focuses on portability and high throughput. The third design includes a forward and an inverse integer transform architectures for H.264/AVC, which targets both the high throughput and area efficiency goals. The forth design is an algorithm and architecture design of the forward integer transforms for HEVC, also targeting high throughput and area efficiency.

In the first portable design, we have proposed the most functionally complete inverse integer transform unit. This unit consists of only one 1-D 8×8 inverse transform unit with three levels of hardware sharing, including (1) embedding two 4×4 inverse transform units into the 1-D 8×8 inverse transform unit; (2) sharing one 4×4 inverse Hadamard transform with one 4×4 inverse transform unit; and (3) embedding a 2-D 2×2 inverse Hadamard transform into the 1-D 4×4 inverse transform unit. Then, an inverse integer transform block is proposed deploying the proposed shared transform unit and an additional quantization module. The inverse integer transform design is implemented on an ASIP-controlled SoC platform conforming to the Wishbone shared bus standard for portability and testability. The design can perform a 2×2, 4×4, and 8×8 data block in one, eleven and nineteen cycles, respectively. The design can support the transforms for videos with the resolutions of up to 16.4 Mpixel^2 with a progressive scan frequency of 30 Hz (30 frames per sec). The resulting circuit area is considerably minimal compared to other designs in its class thanks to the proposed transform unit, which has the embodiment of 4×4 circuits in the 8×8 circuit and the embodiment of 2×2 circuit in the 4×4 circuits.

The second portable design deploys a fully pipelining mechanism supported by an ASIP, two DMACs and two inverse integer transform blocks. The inverse transform block is reused from the first portable design after rescheduling its operation timing. As a result, it can perform a 2×2, 4×4, and 8×8 data block in one, four cycles, and eight cycles, respectively, compared to one, eleven and nineteen cycles in the first portable design, respectively. The second architecture can deliver a high normalized throughput of sixty-four ppc and 15.6 Gpps at 144 MHz using 0.18 μm technology, and can support the transforms for videos with the resolutions of up to 38.4 Mpixel^2 with a progressive scan frequency of 30Hz. These are higher than those of other reported designs in its class. Thanks to the proposed functionally complete inverse integer transform unit, the resulting circuit area is considerably reasonable compared to that of other designs in the same class.

The third design is a high throughput and area-efficient SoC-based FIT/IIT design. Thanks to the newly proposed forward transform unit and the proposed inverse transform unit from the first portable design, our proposed design can perform both 4×4 and 8×8 transforms with additional supports for 2×2 and 4×4 Hadamard transforms of DC coefficients with a reasonable area. Thanks to the proposed pipelining mechanism among input, output, transform and quantization processes, and thanks to the special input and output buffer designs to balance the data load in data input and output buses, our FIT/IIT design achieves a higher DTUA compared to other reported designs. In the 0.18 μm technology, the FIT and IIT modules can be operated at 162.1 and 230.9 MHz, respectively. The inverse and forward transform architecture can support for videos with the resolution of up to 61.6 and 43.2 Mpixel^2, respectively, with progressive scan frequency of 30 Hz.

Besides the three H.264/AVC transform designs, a novel metric for performance-cost comparison among the reported designs is also proposed in this thesis. $DTUA$ has been the metric in the literature for comparisons among high throughput and area-efficient FIT/IIT designs. However, due to the incomprehensiveness of $DTUA$ as well as the current metrics for comparison, some designs use very large bus widths but their authors were still able to claim their area efficiencies. In this thesis, we have proposed a novel performance-cost metric, *PCM*, for the H.264 forward/inverse integer transforms. PCM is defined as the ratio of data throughput over the design cost, which is a product of power, area, and delay. Compared to $DTUA$, *PCM* facilitates more comprehensive comparisons among FIT/IIT designs. Performance-cost analysis software is subsequently proposed to automatically compute values of different metrics, including *PCM*s and $DTUA$, facilitating a comprehensive comparison among inputted designs.

For the emerging HEVC, due to the higher order of complexity compared to H.264, high throughput is even more challenging in integer transform designs. In this thesis, we high throughput and area efficiency for HEVC integer transform designs are addressed at both *algorithm* and *architecture* levels. At the *algorithm* level, we explore fast and low-cost integer transform algorithms, while at the *architecture* level, we explore the high-throughput and area-efficient architectures. We have proposed (1) a novel optimization method to automatically minimize complexity, running time and resource consumption for scalar-multiplication-containing algorithms, (2) a series of novel 4×4 and 8×8 hardware-oriented fast and low-cost integer transform algorithms for HEVC, and (3) a high-throughput and area-efficient transform architecture for the HEVC 4×4 and 8×8 transforms.

The novel optimization method is to achieve fast and low-cost algorithm implementations for scalar-multiplication-containing algorithms in general and the Partial Butterfly algorithms in particular. The method includes a multiplication-to-addition conversion step and three levels of optimization: (1) Complexity Optimization, (2) Timing Optimization and (3) Resource Optimization. Experimental results show that the speed of the automatically generated implementation by the method for the scalar multiplications increases by about 90%, while the cost reduces by 90% compared to the implementation by multiplications. Compared to the conventional multiplication-free implementation, the speed of the generated implementation increases by 33% while the cost reduces from 25 to 33%. It is also proved that the method is computationally feasible to be applied to all the Partial Butterfly algorithms from 4×4 to 32×32.

By applying the proposed optimization method for the scalar multiplications in the Partial Butterfly algorithms and based on the idea of sharing resource among the different transform sizes, we have proposed a series of novel 4×4 and 8×8 hardware-oriented fast and low-cost integer transform algorithms for HEVC. The number of addition/subtractions of the proposed implementations for the 4×4 and 8×8 transforms is fourteen and fifty-eight, respectively. It it can fully perform the 4×4 and 8×8 1-D transforms after four and five addition/subtractions, respectively. Compared to the original Partial Butterfly algorithms and the conventional sequence multiplication-free Partial Butterfly algorithms, the speed of the proposed algorithms increases by 75% and 20%, respectively. The proposed algorithms require around 87% and 16% less resource than the original and the conventional sequence/parallel multiplication-free Partial Butterfly algorithms, respectively. Compared to the reported algorithms in the literature for HEVC, the speed of the proposed algorithm is 17-29% faster and its cost is less.

Finally, we have developed, implemented and fabricated a high-throughput and area-efficient transform architecture for the HEVC 4×4 and 8×8 transforms based on the proposed algorithms. The architecture has designed under a very tight constraint on the I/O pin count of sixteen pixels. We have also introduced several techniques to achieve a high throughput for HEVC transform designs under such a tight constraint in this thesis. The architecture design is implemented in Verilog using UMC 65nm Low Leakage (LL) RVT technology at 1.2V with multi-level of verifications till post-place-and-route-verification. Then the design is sent for Chip fabrication. Testing PCB is also designed and fabricated. Thanks to the above techniques, the proposed architecture can support the transforms for up to Quad-Full High Definition (QFHD) videos, i.e., resolution of 3840×2160 pixel^2, at the progressive scan frequency of 30 Hz. This throughput is much higher and the design's power is much less than those of the reported designs in the literature.

7.2 Future Works

As predicted by Ohm and Sullivan (2013), HEVC encoders may require years of research to be realized due to its complexity. There still has a huge room to explore and reduce the complexity of the HEVC encoders and decoders in general and of the forward and inverse integer transforms in particular.

We have already proposed the fast and low-cost 4×4 and 8×8 forward transforms for HEVC. HEVC also supports the 4×4 and 8×8 inverse integer transforms as well as the 16×16 and 32×32 forward and inverse integer transforms with an additional Discrete Sine Transform for 4×4 luma residual blocks. Therefore, fast and low-cost algorithms are needed for all these transforms. As the proposed optimization method can generate optimized implementations for scalar multiplications. One direction of future work is to develop fast and low-cost 4×4 and 8×8 inverse transform algorithms, 16×16 and 32×32 forward and inverse integer transform algorithms and 4×4 DST algorithms using the proposed optimization method. Based on the fast and low-cost algorithms, high-throughput and area-efficient architectures for these transforms could be subsequently developed.

Another future direction is to continue to improve the optimization method. The current optimization method has a large time complexity for its full search, which is factorial-exponential. Heuristics could be put into the optimization algorithms to reduce the search space or to localize the best candidate. The optimization method could be extended to deal with different running time, not only the shortest running time.

Bibliography

Ahmed, N., Natarajan, T., Rao, K. R., Jan 1974. Discrete Cosine Transform. IEEE Transactions on Computers C-23 (1), 90–93.

ALTERA, Jan 2003. Avalon bus specification – reference manual.
URL http://www.altera.com.cn/literature/manual/mnl_avalon_bus.pdf

Amer, I., Badawy, W., Jullien, G., Mar 2005. A high-performance hardware implementation of the H.264 simplified 8×8 transformation and quantization [video coding]. In: Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing 2005 (ICASSP 2005). Vol. 2. Philadelphia, Pennsylvania, USA, pp. ii/1137 – ii/1140 Vol. 2.

AMS, 2006. Standard cell library of $0.35 \mu m$ CMOS (version 3.72).

ARM, May 1999. AMBA specification (rev 2.0).
URL http://infocenter.arm.com/help/topic/com.arm.doc.ihi0011a/index.html

Bliss, W. G., Julien, A. W., Apr 1990. Efficient and reliable VLSI algorithms and architectures for the discrete fourier transform. In: Proceedings of the International Conference on Acoustics, Speech and Signal Processing 1990 (ICASSP 1990). Vol. 2. Albuquerque, New Mexico, USA, pp. 901–904.

Bossen, F., Bross, B., Sühring, K., Flynn, D., 2012. HEVC complexity and implementation analysis.

Bross, B., Han, W.-J., Sullivan, G. J., Ohm, J.-R., Wiegand, T., Oct 2012. High Efficiency Video Coding (HEVC) text specification draft 9.

Celik, M., Pileggi, L., Odabasioglu, A., 2002. IC Interconnect Analysis, 1st Edition. Springer.

Chao, Y.-C., Tsai, H.-H., Lin, Y.-H., Yang, J.-F., Liu, B.-D., Jul 2007. A novel design for computation of all transforms in H.264/AVC decoders. In: Proceedings of the IEEE International Conference on Multimedia and Expo 2007 (ICME 2007). Beijing, China, pp. 1914–1917.

Chen, K.-H., Guo, J.-I., Wang, J.-S., May 2005. An efficient direct 2-D transform coding IP design for MPEG-4 AVC H.264. In: Proceedings of the IEEE International Symposium on Circuits and Systems 2005 (ISCAS 2005). Kobe, Japan, pp. 4517–4520.

Chen, K.-H., Guo, J.-I., Wang, J.-S., Apr 2006. A high-performance direct 2-D transform coding IP design for MPEG-4 AVC/H.264. IEEE Transactions on Circuits and Systems for Video Technology 16 (4), 472–483.

Chen, W. H., Smith, C. H., Fralick, S. C., Sep 1977. A fast computational algorithm for the Discrete Cosine Transform. IEEE Transactions on Acoustics, Speech, and Signal Processing COM–25 (9), 1004–1009.

Cho, N. I., Lee, S. U., Mar 1991. Fast algorithm and implementation of 2-D DCT. IEEE Transactions on Circuits and Systems 38 (3), 297–305.

Choi, W., Park, J., Lee, S., Nov 2008. A high-performance & low-power unified 4×4 / 8×8 transform architecture for the H.264/AVC codec. In: Proceedings of 23rd International Conference Image and Vision Computing New Zealand (IVCNZ 2008). Lincoln University, Christchurch, New Zealand, pp. 1–6.

Ciletti, M. D., 2003. Advanced Digital Design with the Verilog HDL. Prentice Hall.

Do, T. T., Le, T. M., Dec 2009. A high normalized aggregate throughput soc-based inverse integer transform design for H.264/AVC. In: Proceedings of 12th International Symposium on Integrated Circuits (ISIC 2009). Singapore, pp. 453–456.

Do, T. T., Le, T. M., May 2010. High throughput area-efficient soc-based forward/inverse integer transforms for H.264/AVC. In: Proceedings of the IEEE International Symposium on Circuits and Systems 2010 (ISCAS 2010). Paris, France, pp. 4113–4116.

Gordon, S., Marpe, D., Wiegand, T., Mar 2004. Simplified use of 8×8 transforms – updated proposal & results.
URL http://wftp3.itu.int/av-arch/jvt-site/2004_03_Munich/JVT-K028.doc

Hewlitt, R. M., Swartzlander, E. S., Oct 2000. Canonical signed digit representation for fir digital filters. In: Proceedings of IEEE Workshop on Signal Processing Systems 2000 (SISP 2000). Lafayette, Louisiana, USA, pp. 416–426.

Hu, X., Liu, B., Zhang, C., Jul 2009. A high performance parallel transform and quantization architecture for H.264 decoder. In: Proceedings of the IEEE International Conference on Communications, Circuits and Systems 2009 (ICCCAS 2009). Milpitas, California, USA, pp. 1059–1060.

Hwangbo, W., Kim, J., Kyung, C.-M., May 2007. A high-performance 2-D inverse transform architecture for the H.264/AVC decoder. In: Proceedings of the IEEE International Symposium on Circuits and Systems 2007 (ISCAS 2007). New Orleans,USA, pp. 1614–1616.

IBM, Sep 1999. The CoreConnect bus architecture.
URL https://www-01.ibm.com/chips/techlib/techlib.nsf/products/CoreConnect_Bus_Architecture

IBM, 2008. Standard cell library of 0.13μm CMOS CMRF8SF (version 1.4.0.12 LM).

ISO/IEC 11172, Nov 1993. Coding of moving pictures and associated audio for digital storage media at up to about 1,5 mbit/s – part 2: Video.

ISO/IEC 13818-2, Nov 1994. Generic coding of moving pictures and associated audio information - part 2: Video.

ISO/IEC 14496-2, Jan 1999. Coding of audio-visual objects - part 2: Visual.

ITU-T, Mar 1993. H.261: Video codec for audiovisual services at p $\times$ 384 kbit/s.
URL http://www.itu.int/rec/T-REC-H.261-199303-I/en

ITU-T, Feb 1998. H.263: Video coding for low bit rate communication.
URL http://www.itu.int/rec/T-REC-H.263-199802-S/en

ITU-T, JTC1, I., Nov 2012. HEVC Test Model: HM 9.
URL https://hevc.hhi.fraunhofer.de/

ITU-T and ISO/IEC JTC1, Jan 2010. Joint call for proposals on video compression technology.

Kordasiewicz, R. C., Shirani, S., Sep 2005. ASIC and FPGA implementations of H.264 DCT and quantization blocks. In: Proceedings of the IEEE International Conference on Image Processing 2005 (ICIP 2005). Vol. 3. Genoa, Italy, pp. 1020–1023.

Landman, B. S., Russo, R. L., Dec 1971. On a pin versus block relationship for partitions of logic graphs. IEEE Transactions on Computers c-20 (12), 1469–1479.

Lee, B. G., Dec 1984. A new algorithm to compute Discrete Cosine Transform. IEEE Transactions on Acoustics, Speech, and Signal Processing ASSP–32 (6), 1243–1245.

Malvar, H. S., Hallapuro, A., Karczewicz, M., Kerofsky, L., Jul 2003. Low-complexity transform and quantization in H.264/AVC. IEEE Transactions on Circuits and Systems for Video Technology 13 (7), 598–603.

Marpe, D., Wiegand, T., Gordon, S., Sep 2005. H.264/MPEG4-AVC fidelity range axtensions: Tools, profiles, performance, and application areas. In: Proceedings of the IEEE Conference on Image Processing 2005 (ICIP 2005). Vol. 1. Genoa, Italy, pp. 593–596.

Martuza, M., Wahid, K., May 2012. A cost effective implementation of 8×8 transform of HEVC from H.264/AVC. In: Proceedings of 25th IEEE Canadian Conference on Electrical and Computer Engineering (CCECE '12). Montreal, Quebec, Canada, pp. 1–4.

Nebel, W., Mermet, J., 1997. Low Power Design in Deep Submicron Electronics, 1st Edition. Springer.

Ngo, N., Do, T., Le, T., Kadam, Y., Bermak, A., Jun 2008. ASIP-controlled inverse integer transform for H.264/AVC compression. In: Proceedings of 19th IEEE/IFIP International Symposium on Rapid System Prototyping (RSP '08). Monterey, California, USA, pp. 158–164.

Ohm, J.-R., Sullivan, G. J., Jan 2013. High Efficiency Video Coding: the next frontier in video compression. IEEE Signal Processing Magazine 30 (1), 152–158.

Ohm, J.-R., Sullivan, G. J., Schwarz, H., Tan, T. K., Wiegand, T., 2012. Comparison of the coding efficiency of video coding standards: Including high efficiency video coding (HEVC).

OpenCores, Sep 2002. WISHBONE System-on-Chip (SoC) interconnection architecture for portable IP cores.
URL http://cdn.opencores.org/downloads/wbspec_b3.pdf

Ostermann, J., Bormans, J., List, P., Marpe, D., Narroschke, M., Pereira, F., Stockhammer, T., Wedi, T., Mar 2004. Video coding with H.264/AVC: tools, performance, and complexity. IEEE Circuits and Systems Magazine 4 (1), 7–28.

Park, I.-C., Kang, H.-J., June 2001. Digital filter synthesis based on minimal signed digit representation. In: Proceedings of Design Automation Conference 2001 (DAC 2001). Las Vegas, Nevada, USA, pp. 486–473.

Park, J. S., Ogunfunmi, T., Apr 2009. A new hardware implementation of the H.264 8×8 transform and quantization. In: Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing 2009 (ICASSP 2009). Taipei, Taiwan, pp. 585–588.

Pastuszak, G., Apr 2008. Transforms and quantization in the high-throughput H.264/AVC encoder based on advanced mode selection. In: Proceedings of the IEEE Computer Society Annual Symposium on VLSI 2008 (ISVLSI '08). Le Corum, Montpellier, France, pp. 203–208.

Patterson, D. A., Hennessy, J. L., 1998. Computer Organization and Design: Hardware and Software Interface, 2nd Edition. Morgan Kauffman.

Rabaey, J. M., Chandrakasan, A., Nikolic, B., 2003. Digital Integrated Circuits: A Design Perspective, 2nd Edition. Prentice Hall.

Raja, G., Khan, S., Mirza, M. J., Dec 2005. VLSI architecture & implementation of H.264 integer transform. In: Proceedings of 17th International Conference on Microelectronics (ICM 2005). Islamabad, Pakistan, pp. 218–223.

Rao, K. R., Yip, P., 1990. Discrete Cosine Transform: Algorithms, Advantages, Applications. Academic Press.

Richardson, I. E. G., 2003. H.264 and MPEG-4 Video Compression: Video Coding for Next Generation Multimedia, 1st Edition. Wiley.

Richardson, I. E. G., 2010. The H.264 Advanced Video Compression Standard, 2nd Edition. Wiley.

Rithe, R., Cheng, C.-C., Chandrakasan, A. P., Nov 2012. Quad full-HD transform engine for dual-standard low-power video coding. IEEE Journal of Solid-State Circuits 47 (11), 2724–2736.

Saponara, S., Blanch, C., Denolf, K., Bormans, J., Apr 2003. The JVT advanced video coding standard: Complexity and performance analysis on a tool-by-tool basis. In: Proceedings of the International Packet Video Workshop 2003 (PV 2003). Nantes, France, pp. 1–12.

Savidis, L., Friedman, E. G., 2008. Physical Design Trends for Interconnect, 1st Edition. Morgan Kaufmann.

Sharma, M., Kumar, D., Apr 2012. Wishbone bus architecture – a survey and comparison. International journal of VLSI Design & Communication Systems 3 (2), 107–124.

Shi, B., Zheng, W., Li, D., Zhang, M., Jul 2007. Fast algorithm and architecture design for H.264/AVC multiple transforms. In: Proceedings of the IEEE International Conference on Multimedia and Expo 2007 (ICME 2007). Beijing, China, pp. 2086–2089.

Su, G.-A., Fan, C.-P., Dec 2008. Low-cost hardware-sharing architecture of fast 1-D inverse transforms for H.264/AVC and AVS applications. IEEE Transactions on Circuits and Systems II: Express Briefs 55 (12), 1249–1253.

Suhring, K., Sep 2008. H.264/AVC reference software: JM 14.2.
URL http://iphome.hhi.de/suehring/tml/download/old_jm/jm14.2.zip

Sullivan, G. J., Ohm, J.-R., Han, W.-J., Wiegand, T., 2012. Overview of the High Efficiency Video Coding (HEVC) standard.

Sullivan, G. J., Wiegand, T., Nov 1998. Rate-distortion optimization for video compression. IEEE Signal Processing Magazine 15 (6), 74–90.

Sylvester, D., Keutzer, K., Apr 1999. System-level performance modeling with BACPAC – Berkeley advanced chip performance calculator. In: Proceedings of the International Workshop on System-Level Interconnect Prediction 1999 (SLIP 1999). Monterey, California, USA, pp. 109–114.

Wang, T.-C., Huang, Y.-W., Fang, H.-C., Chen, L.-G., May 2003. Parallel 4×4 2D transform and inverse transform architecture for MPEG-4 AVC/H.264. In:

Proceedings of the IEEE International Conference on Circuits and Systems 2003 (ISCAS 2003). Vol. 2. Bangkok, Thailand, pp. 800–803.

Wenger, S., Hannuksela, M., Stockhammer, T., Westerlund, M., Singer, D., Feb 2005. RFC 3984: RTP Payload format for H.264 video.
URL `http://tools.ietf.org/html/rfc3984#page-2`

Weste, N., Harris, D., 2004. CMOS VLSI Design: A Circuits and Systems Perspective, 3rd Edition. Addison Wesley.

Wiegand, T., Girod, B., Oct 2001. Lagrange multiplier selection in hybrid video coder control. In: Proceedings of the IEEE Conference on Image Processing 2001 (ICIP 2001). Vol. 3. Thessaloniki, Greece, pp. 542–545.

Wiegand, T., Ohm, J.-R., Sullivan, G. J., Han, W.-J., Joshi, R., Tan, T. K., Ugur, K., Dec 2010. Special section on the joint call for proposals on high efficiency video coding (HEVC) standardization. IEEE Transactions on Circuits and Systems for Video Technology 20 (12), 1661–1666.

Wiegand, T., Schwarz, H., Joch, A., Kossentini, F., Sullivan, G. J., Jul 2003a. Rate-constraint coder control and comparison of video coding standards. IEEE Transactions on Circuits and Systems for Video Technology 13 (7), 608–703.

Wiegand, T., Sullivan, G., Luthra, A., Mar 2003b. Draft ITU-T Recommendation and Final Draft International Standard of Joint Video Specification (ITU–T Rec. H.264 — ISO/IEC 14496–10 AVC).
URL `http://wftp3.itu.int/av-arch/jvt-site/2003_03_Pattaya/JVT-G050r1.zip`

Wiegand, T., Sullivan, G. J., Bjøntegaard, G., Luthra, A., Jul 2003c. Overview of the H.264/AVC video coding standard. IEEE Transactions on Circuits and Systems for Video Technology 13 (7), 560–576.

Xilinx, Aug 2010. Virtex-4 family overview.
URL http://www.xilinx.com/support/documentation/data_sheets/ds112.pdf

List of Publications

Journal Papers

1. Trang T.T. Do and Yajun Ha. A Novel Speed and Resource Optimization Method for Scalar-Multiplication Implementations and Novel Fast and Low-Cost Integer Transform Algorithms for HEVC. *IEEE Transactions on Circuits and Systems for Video Technology*, to be submitted.

2. Trang T.T. Do and Yajun Ha. A High-Throughput and Area-Efficient Integer Transform Design for HEVC. *IEEE Transactions on Circuits and Systems for Video Technology*, to be submitted.

Conference Papers

1. Binh P. Nguyen, Trang T. T. Do, Chee-Kong Chui, and Sim-Heng Ong. Prediction-based Directional Search for Fast Block-Matching Motion Estimation. In *Proceedings of the Symposium on Information and Communication Technology (SoICT 2010)*, ACM ICPS 449, pages 86–91, Hanoi, Vietnam, 27-28 August 2010.

2. Trang T. T. Do, Thinh M. Le, Binh P. Nguyen, and Yajun Ha. Performance-Cost Analyses Software for H.264 Forward/Inverse Integer Transform. In

Proceedings of 21st *IEEE International Symposium on Rapid System Prototyping (RSP 2010)*, pages 1–7, Fairfax, Virginia, USA, 8-11 June 2010.

3. Trang T. T. Do and Thinh M. Le. High Throughput Area-Efficient SoC-Based Forward/Inverse Integer Transforms for H.264/AVC. In *Proceedings of the IEEE International Symposium on Circuits and Systems 2010 (ISCAS 2010)*, pages 4113–4116, Paris, France, 30 May - 2 June 2010.

4. Trang T. T. Do and Binh P. Nguyen. A High-Accuracy and High-Speed 2-D 8×8 Discrete Cosine Transform Design. In *Proceedings of* 1st *International Conference on Green Computing and* 2nd *AUN/SEED-Net Regional Conference on ICT (ICGC-RCICT 2010)*, pages 135–139, Yogyakarta, Indonesia, 2-3 March 2010.

5. Binh P. Nguyen and Trang T. T. Do. Cross Directional Rectangle Search for Fast Block-Matching Motion Estimation. In *Proceedings of* 1st *International Conference on Green Computing and* 2nd *AUN/SEED-Net Regional Conference on ICT (ICGC-RCICT 2010)*, pages 132–134, Yogyakarta, Indonesia, 2-3 March 2010.

6. Trang T. T. Do and Thinh M. Le. A High Normalized Aggregate Throughput SoC-based Inverse Integer Transform Design for H.264/AVC. In *Proceedings of* 12th *International Symposium on Integrated Circuits (ISIC 2009)*, pages 453–456, Singapore, 14-16 December 2009.

7. Nghia T. Ngo, Trang T.T. Do, Thinh M. Le, Y.S. Kadam, and A. Bermak. ASIP-controlled Inverse Integer Transform for H.264/AVC Compression. In *Proceedings of* 19th *IEEE/IFIP International Workshop on Rapid System Prototyping (RSP 2008)*, pages 158–164, Monterey, California, USA, 2-5 June 2008.