การสร้างตัวแปลง 1 มิติสำหรับการเข้ารหัสวีดิทัศน์ประสิทธิภาพสูงบน FPGA

นางสาวปานชีวา อารยะชีพปรีชา

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรมหาบัณฑิต
สาขาวิชาวิศวกรรมไฟฟ้า ภาควิชาวิศวกรรมไฟฟ้า
คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย
ปีการศึกษา 2557
ลิขสิทธิ์ของจุฬาลงกรณ์มหาวิทยาลัย

IMPLEMENTATION OF 1D TRANSFORM FOR HIGH EFFICIENCY VIDEO CODING ON FPGA

Miss Pancheewa Arayacheeppreecha

A Thesis Submitted in Partial Fulfillment of the Requirements
for the Degree of Master of Engineering Program in Electrical Engineering
Department of Electrical Engineering
Faculty of Engineering
Chulalongkorn University
Academic Year 2014

Thesis Title	IMPLEMENTATION OF 1D TRANSFORM FOR HIGH EFFICIENCY VIDEO CODING ON FPGA
By	Miss Pancheewa Arayacheeppreecha
Field of Study	Electrical Engineering
Thesis Advisor	Assistant Professor Suree Pumrin, Ph.D.

Accepted by the Faculty of Engineering, Chulalongkorn University in Partial Fulfillment of the Requirements for the Master's Degree

ปานชีวา อารยะชีพปรีชา : การสร้างตัวแปลง 1 มิติสำหรับการเข้ารหัสวีดิทัศน์ประสิทธิภาพสูงบน FPGA (IMPLEMENTATION OF 1D TRANSFORM FOR HIGH EFFICIENCY VIDEO CODING ON FPGA) อ.ที่ปรึกษาวิทยานิพนธ์หลัก: ผศ. ดร. สุรีย์ พุ่มรินทร์, 95 หน้า.

วิทยานิพนธ์นี้นำเสนอสถาปัตยกรรมสำหรับขั้นตอนการแปลง 1 มิติ ของการเข้ารหัสวีดิทัศน์ประสิทธิภาพสูง (HEVC) ซึ่งเป็นมาตรฐานการเข้ารหัสวีดิทัศน์มาตรฐานใหม่สุดในปัจจุบัน งานออกแบบถูกสร้างขึ้นสำหรับอุปกรณ์ลอจิกแบบโปรแกรมได้ (FPGAs) ซึ่งเหมาะสำหรับการผลิตปริมาณน้อย ภาษาที่ใช้ในการออกแบบคือ ภาษา VHDL สถาปัตยกรรมนี้สามารถคำนวณการแปลงได้ทุกขนาด คือ ขนาด 4x4, 8x8, 16x16 และ 32x32 และให้ปริมาณงานต่อหน่วยเวลาที่สูงเท่ากัน โดยปริมาณงานต่อหน่วยเวลานี้มากพอที่จะรองรับการเข้ารหัสวีดิทัศน์ความละเอียด 8K (7680 พิกเซล x 4320 พิกเซล) ที่อัตรา 30 เฟรมต่อวินาที สถาปัตยกรรมนี้สามารถรับรูปแบบข้อมูลขาเข้าได้หลายรูปแบบตามการแบ่งโครงสร้างแบบต้นไม้แบ่งสี่ส่วน การใช้ทรัพยากรเฉพาะในงานที่อาจหน่วงระบบเช่นการคูณเป็นกลวิธีสำคัญในการออกแบบอุปกรณ์ลอจิกแบบโปรแกรมได้ ดังนั้นเราจึงใช้ตัวคูณเฉพาะในแผ่นการประมวลผลสัญญาณดิจิตอล เพื่อให้ได้งานออกแบบที่มีประสิทธิภาพสูง และเพื่อเป็นการประหยัดทรัพยากรทั่วไป ฮาร์ดแวร์สำหรับการแปลงขนาดเล็กถูกนำมาใช้ซ้ำในการแปลงขนาดใหญ่เพื่อประหยัดทรัพยากรโดยรวมในระบบ เนื่องจากตัวคูณเฉพาะเป็นทรัพยากรที่มีจำนวนจำกัด เราจึงออกแบบการใช้ตัวคูณเฉพาะร่วมกันระหว่างหลายการคูณ เพื่อลดจำนวนตัวคูณเฉพาะที่ต้องใช้ จนกระทั่งงานออกแบบนี้สามารถสร้างบนอุปกรณ์ลอจิกแบบโปรแกรมได้ขนาดเล็ก เช่น Spartan3A ได้ ในงานวิทยานิพนธ์นี้ ได้ออกแบบวิธีการเข้ารหัสรูปแบบข้อมูลขาเข้าให้มีประสิทธิภาพ โดยรูปแบบข้อมูลขาเข้าได้มาจากการแบ่งข้อมูลแบบโครงสร้างต้นไม้แบ่งสี่ส่วน ซึ่งเป็นวิธีการแบ่งเพื่อให้ได้หน่วยประมวลผลย่อยสำหรับขั้นตอนการแปลงของการเข้ารหัสวีดิทัศน์ประสิทธิภาพสูง สุดท้ายซอฟต์แวร์มาตรฐานของการเข้ารหัสวีดิทัศน์ประสิทธิภาพสูงได้ถูกนำมาใช้เข้ารหัสชุดวีดิทัศน์มาตรฐาน เพื่อนำข้อมูลในขั้นตอนการแปลงมาเปรียบเทียบกับผลลัพธ์การจำลองสถาปัตยกรรมเพื่อที่จะยืนยันความถูกต้อง

ภาควิชา	วิศวกรรมไฟฟ้า	ลายมือชื่อนิสิต	....
สาขาวิชา	วิศวกรรมไฟฟ้า	ลายมือชื่อ อ.ที่ปรึกษาหลัก	....
ปีการศึกษา	2557		

5770225421 : MAJOR ELECTRICAL ENGINEERING
KEYWORDS: HIGH EFFICIENCY VIDEO CODING (HEVC) / FIELD PROGRAMMABLE GATE ARRAY (FPGA) / INTEGER TRANSFORM / VIDEO CODING / ARCHITECTURE

PANCHEEWA ARAYACHEEPPREECHA: IMPLEMENTATION OF 1D TRANSFORM FOR HIGH EFFICIENCY VIDEO CODING ON FPGA. ADVISOR: ASST. PROF. SUREE PUMRIN, Ph.D., 95 pp.

This thesis proposes a 1-D transform architecture for the latest video coding standard, the High Efficiency Video Coding (HEVC). The design is described in VHDL, and aimed for Field Programmable Gate Arrays (FPGAs), which are suitable for low volume productions. All transform sizes, which are 4x4, 8x8, 16x16, and 32x32, can be computed by the proposed architecture with equally high throughput. The throughput is high enough to encode 8K(7680 pixels x 4320 pixels) videos at 30 frames/s. The proposed architecture can receive flexible input combinations resulting from a quad-tree partitioning. Using dedicated resources in critical tasks such as multiplications is an important strategy for FPGA designs, so dedicated multipliers in the DSP slices are extensively employed to gain high performance design and save general purpose resources. Hardware of a small size transform is completely reused in a larger size to further save the overall resources. Since the dedicated multipliers are usually expensive resources, a multiplier sharing scheme is invented in this thesis. The total number of dedicated multipliers required is reduced such that the design can be implemented on small size FPGA such as the Spartan3A. A scheme called the configuration encoding scheme is created to efficiently represent 1-D transform input combinations resulting from a quad-tree partitioning, which is the partitioning used to get basic processing units of the transform step in the HEVC. Finally, the HEVC reference software is used to encode a set of standard test sequences, then data of the transform step are recorded and compared with simulation results of the architecture to ensure correctness.

Department:	Electrical Engineering	Student's Signature	....
Field of Study:	Electrical Engineering	Advisor's Signature	....
Academic Year:	2014		

ACKNOWLEDGEMENTS

This thesis could not be possible without my advisor. So, first thank is to Asst. Prof. Suree Pumrin, Ph.D. Another sincerest appreciation is for Boonchuay Supmonchai. Thanks for their time and effort contribution they give to me throughout my research. Their basic knowledge tutorials and guidance enable me to start this project. Their useful suggestions make me see several new ideas to follow. Thanks also for teaching me how to effectively express my academic work.

I would like to express my appreciation to Asst. Prof. Wanchalerm Pora, Ph.D., and Asst. Prof. Kittiphan Techakittiroj, Ph.D., who are my thesis committee. Their constructive suggestions during my thesis proposal and final defense examination help completing my thesis. Thanks all of the teachers in the Department of Electrical Engineering who gave me necessary fundamental knowledge during the time I studied here.

I also would like to express my appreciation to the Embedded System and IC Design (ESID) laboratory, Department of Electrical Engineering for resources and financial supports. I would like to thank ESID laboratory junior and senior members for every of their helps.

I owe my deep gratitude to Patheera Uthaichana, Design Gateway Co., Ltd. for her answers of my technical questions about FPGAs. This work is also received supports from the Human Resource Development for Eletronics Design Industry (HRD for EDI) Project, supported by Board of Investment (BOI) Thailand and coordinated by Thai Embedded Systems Association (TESA).

Last but very important, I would like to express my sincerest appreciation to my parents. Thanks for their financial support, and most importantly their strong encouragement during my Master Degree study.

CONTENTS

List of Figures

List of Tables

Chapter 1
Introduction

1.1 Motivation and Significant of the Research Problems

Video information is continually increased in both resolutions and frame rates. Modern video codecs will need to support higher resolutions such as 4K (3840x2160) and faster frame rates, as high as 120 fps. Higher resolutions and frame rates mean more information needed to be processed, as well as higher network bandwidth needed for transmitting video data and larger storage spaces are necessary to store them.

Video compression techniques reduce redundancy information, which make it possible to represent the original video data using fewer bits. The High Efficiency Video Coding (HEVC) is the latest video coding standard jointly developed by the ITU-T Video Coding Experts Group (VCEG) and the ISO/IEC Moving Picture Experts Group (MPEG). The HEVC is able to deliver same video quality at 50% bit-rate compared with its predecessor, the H.264/AVC [1].

To achieve high coding efficiency, several new or improved coding techniques have been introduced in the HEVC, which cause the complexity of the codec to increase significantly. The HEVC decoders are slightly more complex than the H.264 decoders, while the HEVC encoders are remarkably more complex, and expected to be a subject of research in years to come [2].

Due to the HEVC complexity and its focus on increased use of parallel processing architectures [1], implementation of the HEVC encoders on hardware, instead of software, is an interesting option. Hardware designs can process several tasks concurrently, which make its more feasible to build an HEVC real-time encoder.

High volume hardware productions are usually fabricated using Application Specific Integrated Circuits (ASICs) technology. However, for low volume productions, Field Programmable Gate Arrays (FPGAs) are more suitable alternative. FPGAs are reconfigurable hardware, which can be configured according to circuit designers' need. FPGAs design development also has cheaper non-recurring engineering cost, compared with ASICs, and faster time to market [3]. Furthermore, FPGA designs can be upgraded as the technology changes.

In this thesis, FPGA architectures for the HEVC transform will be designed and synthesized. HEVC is a hybrid video coding. The coding process is composed of four main steps, which are the prediction, the transform, the entropy encoding, and the filter. The HEVC transform, which is an integer transform, is an important component in the HEVC coding standard. To achieve the targeted bit-rate, HEVC employs variable and larger block size transform, ranging from 4x4 to 32x32, which increases complexity of its transform significantly. The design IP (intellectual property) cores in this research can potentially be used as part of a HEVC encoder design on FPGA.

1.2 Literature Reviews

Kim et al. [4] have proposed a high performance hardware architecture for 1-D forward transform of the HEVC. The architecture can support transform unit (TU) sizes of 4x4, 8x8, 16x16, and 32x32. The design can compute 1-D transform within 38 cycles, irrespective of transform sizes. The throughput can reach 10 Gsamples/s on TSMC 180 nm CMOS technology, with an operating frequency of 400 MHz. The proposed architecture is claimed to support 4K (3840x2160) at 30 fps.

The design in [4] is based on the partial butterfly algorithm, which is the standard algorithm used in the HEVC reference software [5]. The multiplication step is decomposed into binary shifts and additions. Outputs from the first step, the odd and even components, is shifted up 0, 1, ..., 6 position, then added to generate product terms of desired coefficients. The fact that small-size transform coefficients are sub-sampled of large-size transform coefficients is employed to optimize common operator inside the design.

Another architecture for a 16-point 1-D transform of the HEVC is proposed by Jeske et al. [6]. The architecture is aimed to be implemented on FPGAs, which are reconfigurable hardware. The targeted families are Cyclone II and Stratix III of Altera.

The design used both algorithmic optimization as well as architectural optimization to achieve its goals, which are high throughput and low resource usage. Results of the proposed architecture are compared with a direct implementation architecture, using the standard algorithm. Maximum resource usage gain of 73.7% and maximum frequency gain of 445.6% was achieved. To visualize the throughput gain of the design, it is shown that the optimized architecture can support QFHD (3840x2160) at 30 fps but the direct implementation architecture cannot.

In [6], fully combinational strategy is employed to reduce hardware overhead. The design is based on the reference partial butterfly algorithm, with several optimization techniques. The techniques include factorizing computation equations in the third step of the algorithm, the step which sums together multiplied terms, to reduce bit width of the operators; decomposing the multiplication steps into binary shifts and additions; postponing binary shifts in the multiplication step to the end to save the number of bits in adders. The third step computation equations are further factorized to facilitate sharing of sub-expression between outputs.

To cut down more bits in adder/subtractor operators, outputs from the third step of the algorithm are shifted down before being added with offset constant in the rounding step. Finally, zero concatenations, which represent the binary shifts, are carefully implemented to minimize the number of bits necessary to implement each adder.

Fig.1 depicts a partial diagram of the proposed architecture in [6]. This hardware part computes an output x_1 from the 16 inputs, $w_i; i = 0,1, \dots ,16$, which enter on the left of the figure.

Zhao [7] proposed a 2-D forward transform architecture for the HEVC. The architecture can support all transform sizes, i.e. 4x4, 8x8, 16x16, and 32x32. The design is focused on high throughput. Weighted average throughput is used for measuring performance of the design, since different transform size yields different throughput [7].

Synthesis results on the Cyclone IV E FPGA show average throughput of 238.13 Msamples/s, which is enough to process 2560x1600 video at 30 fps. The design is further synthesized under Application Specific Integrated Circuits (ASICs) 45

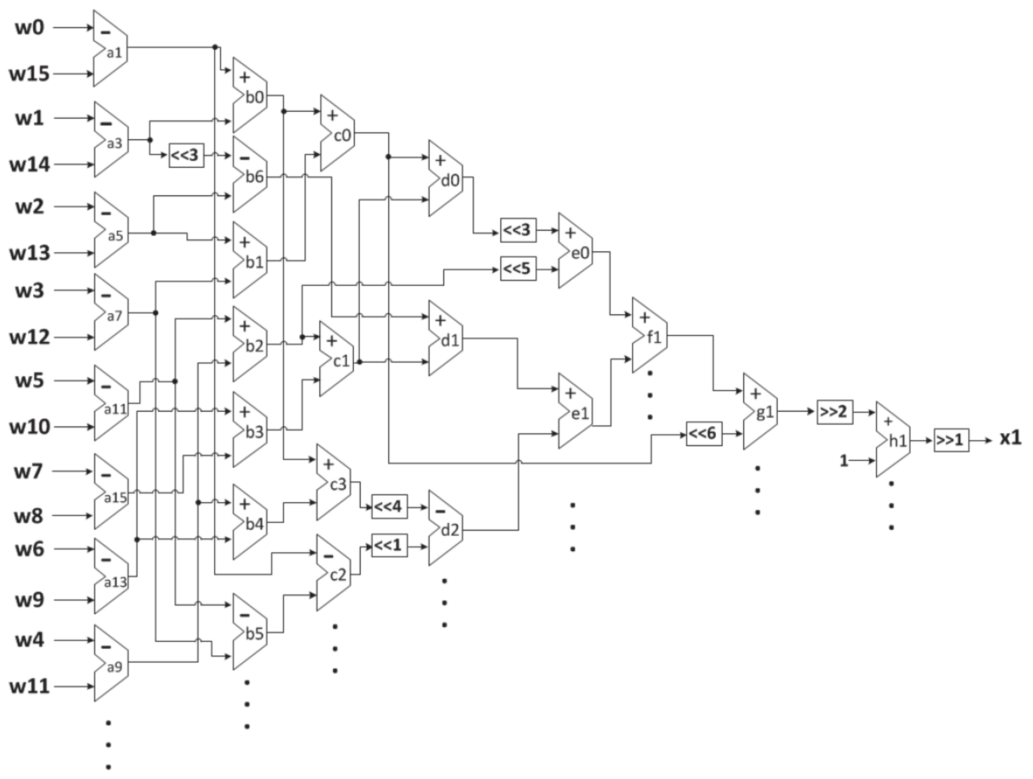


Fig. 1 Partial diagram of the proposed architecture in [6].

nm technology, which yields a throughput of 634.35 Msamples/s. This throughput is enough to support 4Kx2K (4096 pixels x 2048 pixels) video at 30 fps.

Partial butterfly algorithm is also used as the basis of the work in [7]. A modification is made to the standard algorithm by adding the result from the first step in the algorithm together before multiplying with 64. For example, Computation of

$$Z_0 = 64 \cdot EEE_0 + 64 \cdot EEE_1 \tag{1}$$

is carried out by adding EEE_0 with EEE_1 first, to generate $EEEE_0$. Then, Z_0 is computed from multiplying $EEEE_0$ with 64.

Since parallel implementation of multipliers on ASICs lead to relatively slow and costly design, multiplication steps are instead replaced with a series of binary shifts and additions. Another optimization technique in this design is reusing hardware utilized by small-size transform in computing other large-size transform.

Three architectures for the 1-D integer transform for the HEVC are proposed in Park et al. [8]. The architectures have advantage in area, delay, and power respectively. All the designs in [8] are based on the partial butterfly algorithm. The designs can support 4x4, 8x8, 16x16, and 32x32 transforms.

Fundamental block diagram for the design in [8] is depicted in Fig.2. Three modifications are made to this fundamental architecture to generate 3 variants called Flexible-1, Flexible-2, and Flexible-3 architecture.

The fundamental block is composed of the INPUT-ADDER-UNIT (IAU) which compute the first step in the partial butterfly algorithm; the SHIFT-ADD-UNIT (SAU) which compute the second step; and OUTPUT-ADDER-UNIT (OAU) which compute the third step. Some part of the transform computation is reused from smaller-point transform, which is shown as (N/2)-POINT FIXED INTEGER DCT UNIT in Fig.2.

The multiplication step, the second step of the partial butterfly algorithm, is implemented as a set of multiple constant multiplications (MCM). Example of an MCM unit is shown in Fig.3. This MCM unit receives $b(i)$ as the input, and generate $18 \cdot b(i), 50 \cdot b(i), 75 \cdot b(i),$ and $89 \cdot b(i)$, which is $t_{i,18}, t_{i,50}, t_{i,75},$ and $t_{i,89}$ in Fig.3 respectively. Basically, MCM technique decomposes multiplications into binary shifts and additions and optimizes sharing of operations between outputs.

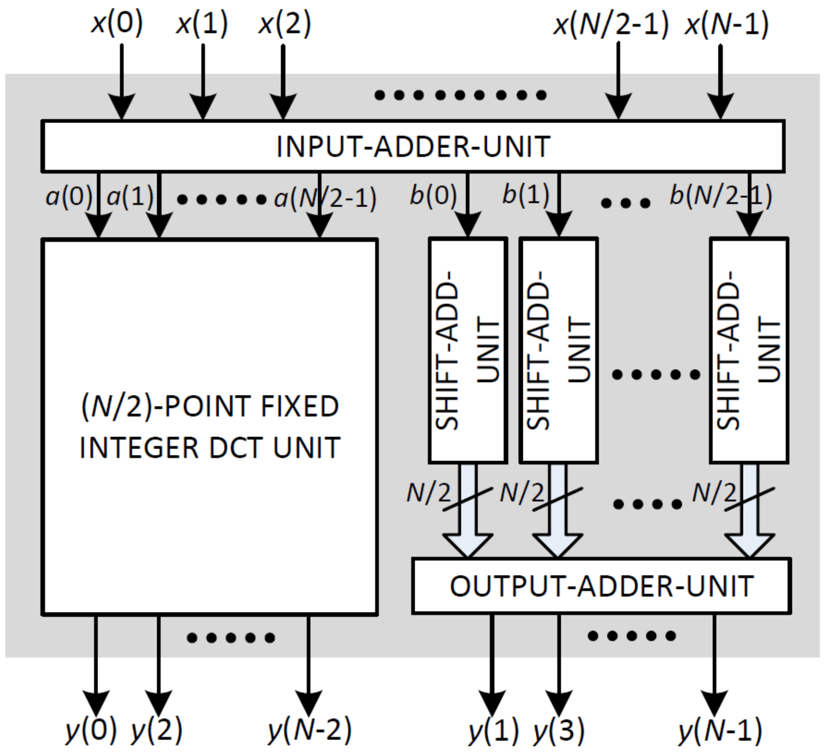


Fig. 2 Fundamental block diagram for the design in [8].

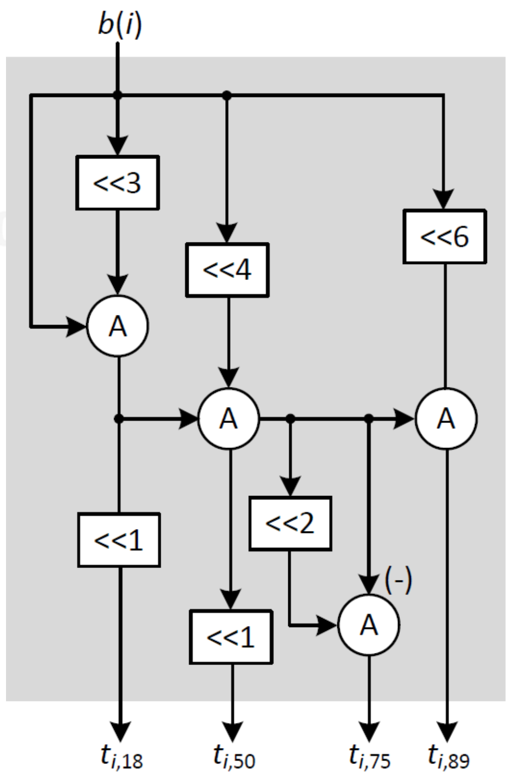


Fig. 3 An MCM unit, which compute $18 \cdot b(i), 50 \cdot b(i), 75 \cdot b(i),$ and $89 \cdot b(i)$ [8].

Block diagrams in Fig.4 are architecture variants modified from the fundamental block diagram in Fig.2. Flexible-1 has advantage in area; Flexible-2 has advantage in delay; and Flexible-3 has advantage in power. Flexible-2 and Flexible-3 can support 7680x4320 video at 30 fps with operating frequency only 94 MHz, which can save power.

Fundamental block diagram in Fig.2 is modified by adding MUX-UNIT in to the design to generate Flexible-1 architecture. The MUX-UNIT as highlighted in Fig.4 (a), is used for selecting inputs for the (N/2)-POINT REUSABLE INTEGER DCT UNIT. If the architecture is configured to compute a (N/2)-point transform, the inputs are selected as $x(0), x(1), \ldots, x(N/2-1)$. Otherwise, the inputs are selected as $a(0), a(1), \ldots, a(N/2-1)$, which is computed from $x(0), x(1), \ldots x(N-1)$ of a N-point transform.

Flexible-2 architecture is generated by inserting another (N/2)-POINT REUSABLE INTEGER DCT UNIT to the Flexible-1 architecture. The added block enables the architecture to compute 2 sets of (N/2)-point transform simultaneously. Thus, N samples of data are processed in every cycle irrespective of transform size, which results in equal throughput for all transform size.

Flexible-3 architecture is further modified from Flexible-2 by merging SHIFT-ADD-UNITs (SAUs) with the added (N/2)-POINT REUSABLE INTEGER DCT UNIT. The merging result units are called CONFIGURABLE SHIFT-ADD-UNIT, which are simply extended version of the MCM units. Notice 2-to-1 MUXes inside the CONFIGURABLE SHIFT-ADD-UNIT, this unit can compute $36 \cdot x(5), 83 \cdot x(5), 64 \cdot x(5),$ and $64 \cdot x(5)$, or $18 \cdot b(1), 50 \cdot b(1), 75 \cdot b(1),$ and $89 \cdot b(1)$.

Meher [9] extended the work in [8] by employing Flexible-2 architecture as the base unit of their design. The overall block diagram of the 1-D transform design is depicted in Fig.5. The structure can be used for implementation of a set of N-point transform or two sets of (N/2)-point transform, for any even integer N.

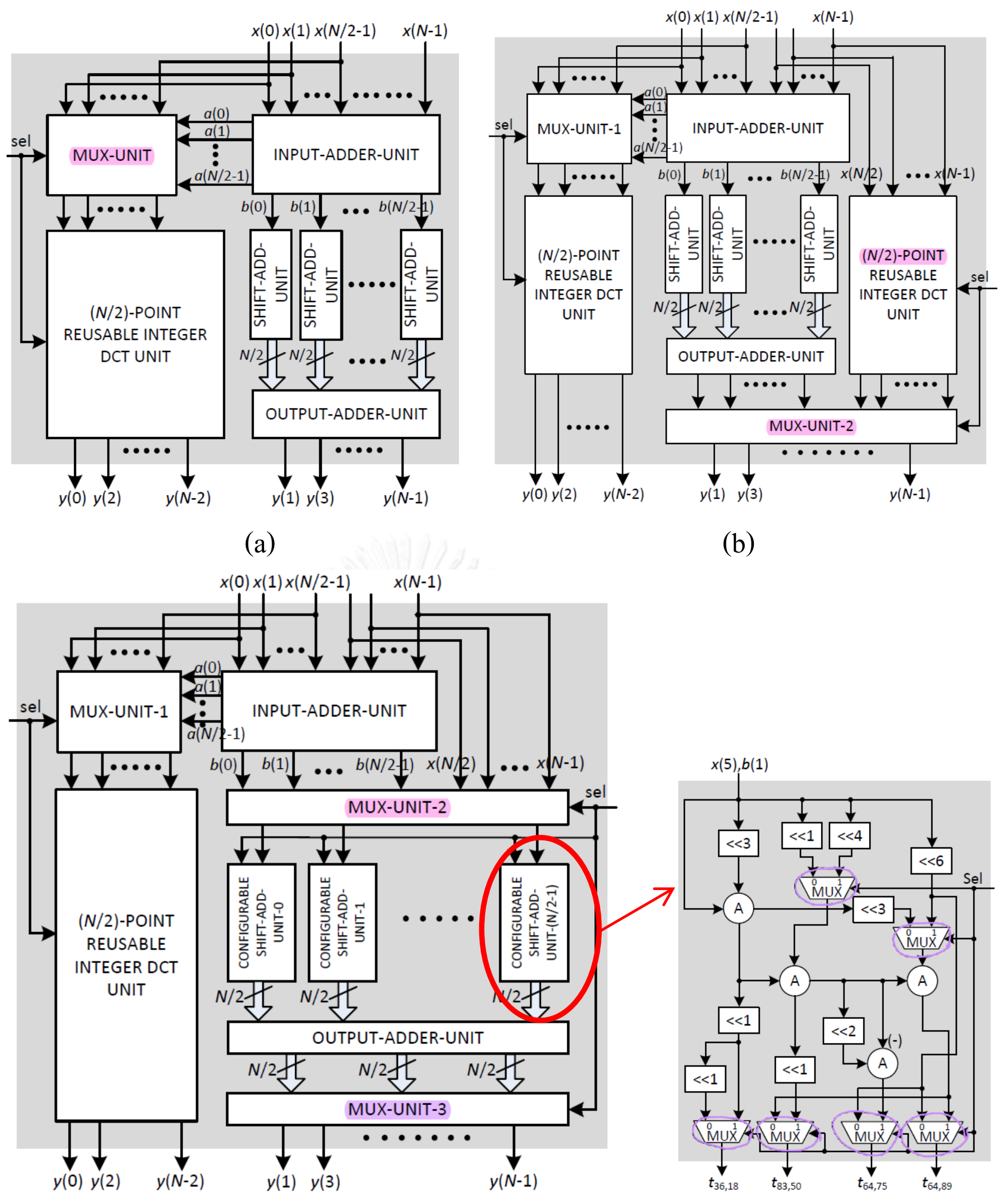

x(0) x(1) x(N/2-1) x(N-1)
sel
MUX-UNIT
INPUT-ADDER-UNIT
a(0) a(1) a(N/2-1)
b(0) b(1) b(N/2-1)
(N/2)-POINT REUSABLE INTEGER DCT UNIT
SHIFT-ADD-UNIT
N/2
OUTPUT-ADDER-UNIT
y(0) y(2) y(N-2) y(1) y(3) y(N-1)
(a)
MUX-UNIT-1
MUX-UNIT-2
x(N/2) x(N-1)
(b)
MUX-UNIT-3
CONFIGURABLE SHIFT-ADD-UNIT-0
CONFIGURABLE SHIFT-ADD-UNIT-1
CONFIGURABLE SHIFT-ADD-UNIT-(N/2-1)
x(5),b(1)
<<3 <<1 <<4 <<6 <<3 <<1 <<2 <<1 <<1
Sel
MUX
A
(-)
t36,18 t83,50 t64,75 t64,89
(c)

Fig. 4 Modified architectures (a) Flexible-1 (b) Flexible-2 (c) Flexible-3 [9].

To use this design as a transform engine for the HEVC, consider the case when N is equal to 32. Depend on the control unit, the architecture can compute a set of 32-point transform, or two sets of 16-point transform. When the architecture is configured to compute a set 32-point transform, the upper (N/2)-point reusable integer DCT unit, the IAU, SAU, and OAU, are used. Otherwise, both of the (N/2)-point reusable integer DCT units are used for computing 2 sets of 16-point transform.

The design inside the (N/2)-point reusable integer DCT units that is employed in computing 16-point transform uses the same structure in Fig.5. Each design module can compute a set of 16-point transform or two sets of 8-point transform. The architecture is recursively designed in this fashion until it reaches the 4-point transform.

In summary, the hardware can compute a set of 32-point transform, two sets of 16-point transform, four sets of 8-point transform, or eight sets of 4-point transform. The architecture yields equal throughput irrespective of transform size.

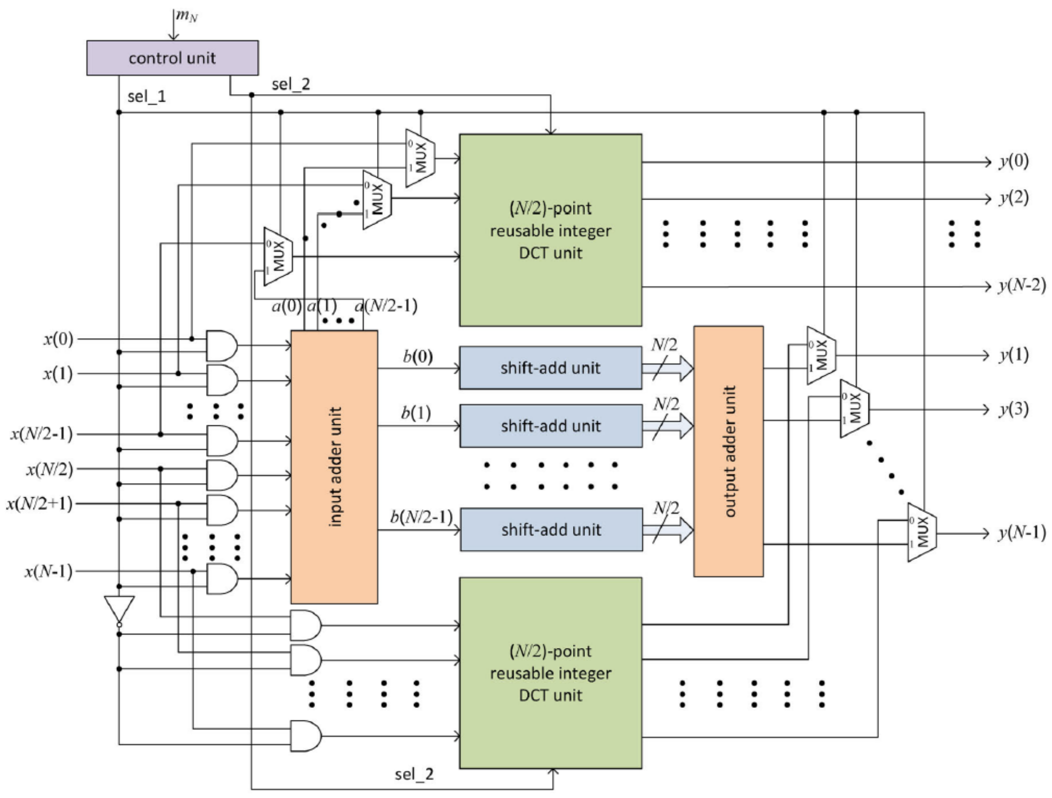


Fig. 5 Overall block diagram of the 1-D transform design in [10].

The architecture is further pruned by integrating the rounding step in the reference partial butterfly algorithm into the SAU. The integration makes the transform results of the architecture not exactly correct, i.e. the results are only approximation, but the effect to the coding performance is only marginal. Pruning strategy can reduce resource usage in the design.

Both the original and pruned architectures can support 8K (7680x4320) video at 60 fps on TSMC 90 nm technology at operating frequency of 187 MHz.

Table.1 summarizes the performance and resource usage of the reviewed architectures.

Table. 1 Design summary of architectures in the literature.

	Kim et al. [5]	Jeske et al. [6]	Zhao et al. [7]	Park et al. [8]	Meher et al. [9]
Technology	ASIC (180 nm)	Cyclone II (90 nm)	Cyclone IV (60 nm)	ASIC (150 nm)	ASIC (90 nm)
Function	1D all sizes	1D 16x16	2D all sizes	1D all sizes	1D all sizes
Throughput (MSamples/s)	10,000	376.2	238.13	1,504	2,990
Max Freq (MHz)	400	23.51	125	94	187
LUTs	-	5,343 (ALUTs)	-	-	-
FFs	-	-	-	-	-
Slices	-	-	40,541 (LEs)	-	-

1.3 Thesis Objectives

1.3.1 Propose a 1D forward integer transform architecture of the High Efficiency Video Coding (HEVC) to be used as a part of an HEVC encoder on FPGA platform. The architecture will,

- have high throughput

- be flexible.

1.3.2 Design a test program to verify the correctness of the proposed architecture.

1.4 Thesis Scope

1.4.1 Propose a 1D forward transform architecture of the HEVC on Xilinx FPGA which

- have high throughput (support at least 4K @ 30 fps)

- flexible (can support all transform size, 4-point; 8-point; 16-point; 32-point, and all possible input combinations resulted from quad-tree partitioning with equal throughput)

1.4.2 Design a test program to verify the correctness of the proposed architecture.

1.5 Expected Results

1.5.1 A high throughput and flexible FPGA IP core for the 1D forward integer transform of the HEVC. The architecture is aimed to be used as part of a real-time HEVC encoder on FPGA platform.

1.5.2 A test program for verifying the correctness of the proposed architecture.

1.6 Research Procedures

1.6.1 Review literatures related to hardware designs for 1D integer transform of the HEVC.

1.6.2 Study the HEVC integer transform and its standard algorithm, the partial butterfly algorithm, employed in the HEVC reference software.

1.6.3 Design and simulate an FPGA architecture for the core 16-point 1D forward transform of the HEVC.

1.6.4 Design and simulate a reconfigurable FPGA architecture for all integer transform size, which are 4-point, 8-point, 16-point, and 32-point.

1.6.5 Evaluate and improve the first proposed design to be more flexible, i.e. able to compute 1D transform of every possible input combination with equal throughput.

1.6.6 Design a test program.

1.6.7 Verify the proposed architectures using the test program.

1.6.8 Summarize the results, analyze and compare performance and resource usages with other previous works in literature.

1.6.9 Write the thesis report.

1.7 Thesis Outlines

Following the introduction in this chapter, related backgrounds are presented in **chapter.2**. The chapter includes six subtopics. First, the High Efficiency Video Coding (HEVC), which is the video coding standard whose transform step will be implemented in this thesis, is reviewed. Basic processing units of the transform step called transform units (TUs) are discussed. The third subtopic is details about the transform step of the HEVC, which can be viewed as matrix multiplication. Next, a standard algorithm suitable for hardware implementation of the transform step is illustrated. The algorithm is called the partial butterfly algorithm.

The last two subtopics of chapter.2 is about standard measurements for comparisons of video coding quality. There are two main aspects to compare. The first aspect is the quality of videos after reconstruction by a video encoder. This quality is measure by Peak Signal-to-Noise Ratio (PSNR) indicator. Another aspect measures compression performance of video coding techniques. The indicator for the latter aspect is Bjontegaard Delta bitrate (BD-rate).

Chapter.3 and chapter.4 concern with the two proposed architectures in this thesis. Both architectures are aimed to be implemented on FPGAs. The first architecture, proposed in **chapter.3**, utilizes dedicated multipliers in FPGAs to gain a high throughput design. The dedicated multipliers are shared among different operations because they are scarce and therefore expensive resources. Another architecture, proposed in **chapter.4**, can receive flexible input combinations, computes transform of those inputs, and produces outputs with uniform throughput regardless of input combinations.

Test procedure and a test program for verifying the correctness of the proposed architectures are discussed in **chapter.5**. Transform inputs and outputs data are retrieved from the HEVC reference software [5]. Automated testbench is written in VHDL to inject transform input data into the proposed architecture, retrieve output data, and automatically compare transform output results with the reference data.

Finally, **chapter.6** concludes the thesis. Possible future works are also suggested in this chapter.

Chapter 2
Related Background

2.1 High Efficiency Video Coding (HEVC)

The High Efficiency Video Coding (HEVC) is the latest video coding standard jointly developed by the ITU-T Video Coding Experts Group (VCEG) and the ISO/IEC Moving Picture Experts Group (MPEG) standardization organizations. The standard is the successor of the well-known H.262/MPEG-2 and H.264/MPEG-4 video coding standard. The HEVC focuses on ever increasing resolution in modern video data as well as the use of parallel processing architectures available presently.

The HEVC is a hybrid video coding as in the case of all prior video coding standard since H.261. The HEVC video encoder block diagram is depicted in Fig.6 [1]. Hybrid video coding approaches are usually composed of four main steps.

The first step is the prediction step, which consists of Intra-Picture Estimation, Intra-Picture Prediction, Motion Estimation, and Motion Compensation blocks in the block diagram. The second step is the transform step, which consists of Transform, Scaling & Quantization, and Scaling & Inverse Transform blocks. The third step is the entropy coding, which is the Header Formatting & CABAC block. The final step is the loop filter, which consists of the Filter Control Analysis, and Deblocking & SAO Filters blocks.

Many new or improved coding techniques have been adopted in HEVC to achieve the targeted coding efficiency, around 50% bit-rate of its predecessor, H.264/AVC, at the same video quality. For example, the concept of Coding Tree Unit (CTU) is introduced. The HEVC is a block-based video compression, each video frame is divided into square blocks. HEVC employs variable size square blocks, up to 64x64 samples, instead of fixed size macroblocks in prior coding standard.

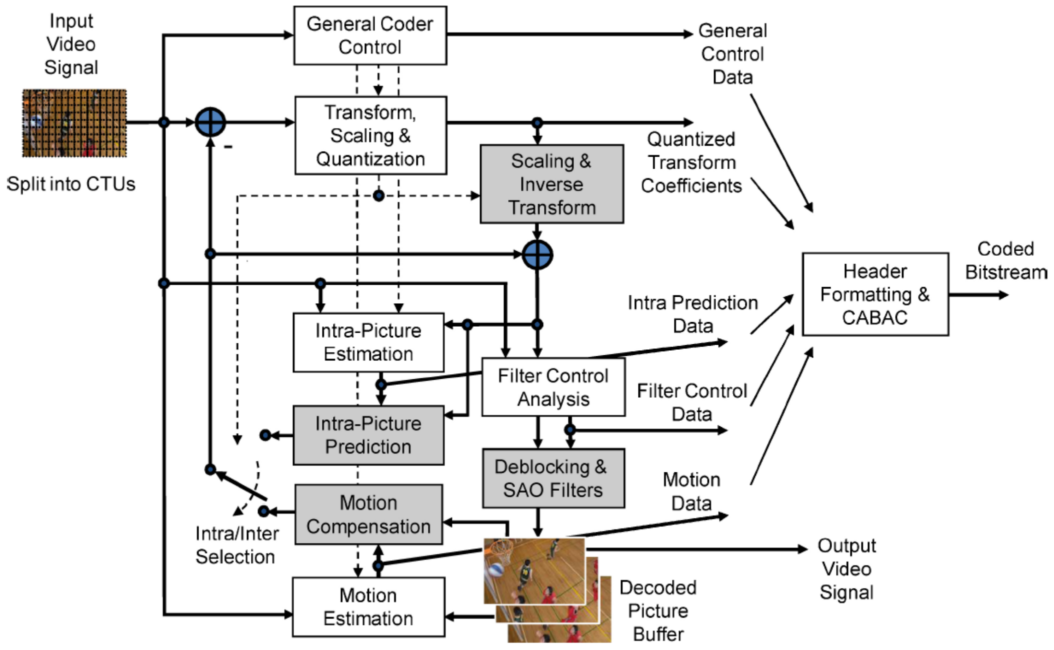


Fig. 6 The HEVC encoder block diagram [1].

Intra prediction in HEVC supports more prediction modes, 33 directional prediction mode in all, than H.264/AVC, which provides the encoder with a large set of prediction choices. Integer transforms with 4-point, 8-point, 16-point, and 32-point are employed in HEVC. Use of large block size transform make the coding efficiency higher, however, the complexity of the encoder is also increased.

It is worthy to note that only the syntax elements and the decoding processes are defined in the standard, which provides the designer a great flexibility to optimize the encoder. The standard comes with compliant reference software called the HM reference software [5]. The reference software is just an illustrated implementation of the HEVC codec, which is suitable for research and experiment. The software is not intended for real usage, and is not optimized for any specific application.

2.2 Transform Unit (TU)

The basic unit of the coding layer of the HEVC is the coding tree unit (CTU) [1]. A coding tree unit is composed of a luma coding tree block (CTB), corresponding chroma CTBs, and associated syntax elements. Luma CTBs can have a size of 16x16, 32x32, or 64x64. Larger CTBs lead to better compression.

The coding tree unit (CTU) can be partitioned into smaller blocks called coding units (CUs), each of which is composed of a luma coding block (CB), corresponding chroma CB, and associated syntax elements. Fig.7 shows CTUs partitioning examples. The partitioning is signaled by quadtree-like signaling in the syntax elements. If a NxN block is signaled to be partitioned, it will be divided into four blocks of size (N/2)x(N/2). The coding block (CU) is the root of a tree of transform blocks (TUs), which will be explained next.

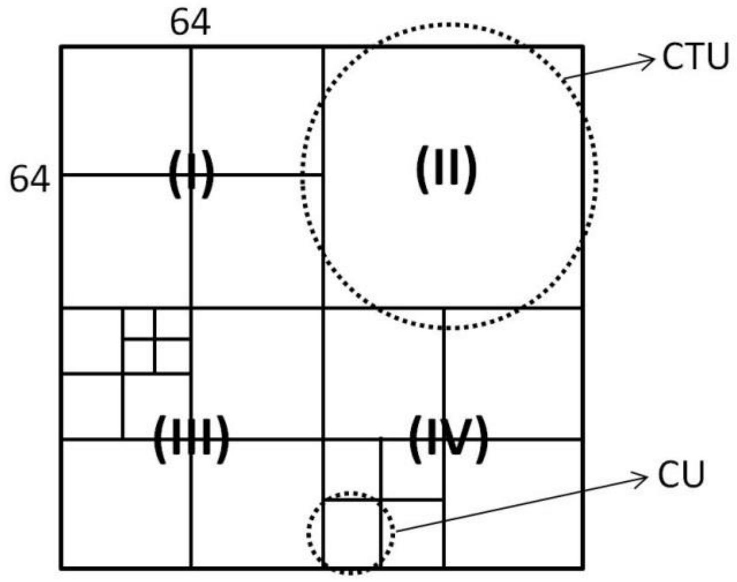


Fig. 7 examples of CTUs partitioning using quadtree-like signaling.

The coding block (CU) can be further partitioned into transform units (TUs) and prediction units (PUs) using another level of quadtree-like signaling. Prediction units (PUs) are consisted of a luma prediction block (PB), corresponding chroma PB, and associated syntax elements. The PBs are basic block for prediction steps in the HEVC hybrid video coding.

Transform units (TUs) are consisted of a luma transform block (TB), corresponding chroma TB, and associated syntax elements. The TBs are basic block for the transform step in the HEVC. Same quadtree structure is applied to partition both chroma and luma components of a CB to generate luma TBs and corresponding chroma TBs. HEVC support square transform block of size 4x4, 8x8, 16x16, and 32x32.

2.3 HEVC Transform

In transform step of a hybrid video coding, residual data from the prediction step are transformed into spectral domain which is more amenable to be used by the entropy coding step. HEVC adopts integer transforms which are approximated scaled versions of the Discrete Cosine Transform (DCT) and the Discrete Sine Transform (DST). The core transforms are derived from the DCT, while an alternative 4x4 transform is derived from the DST.

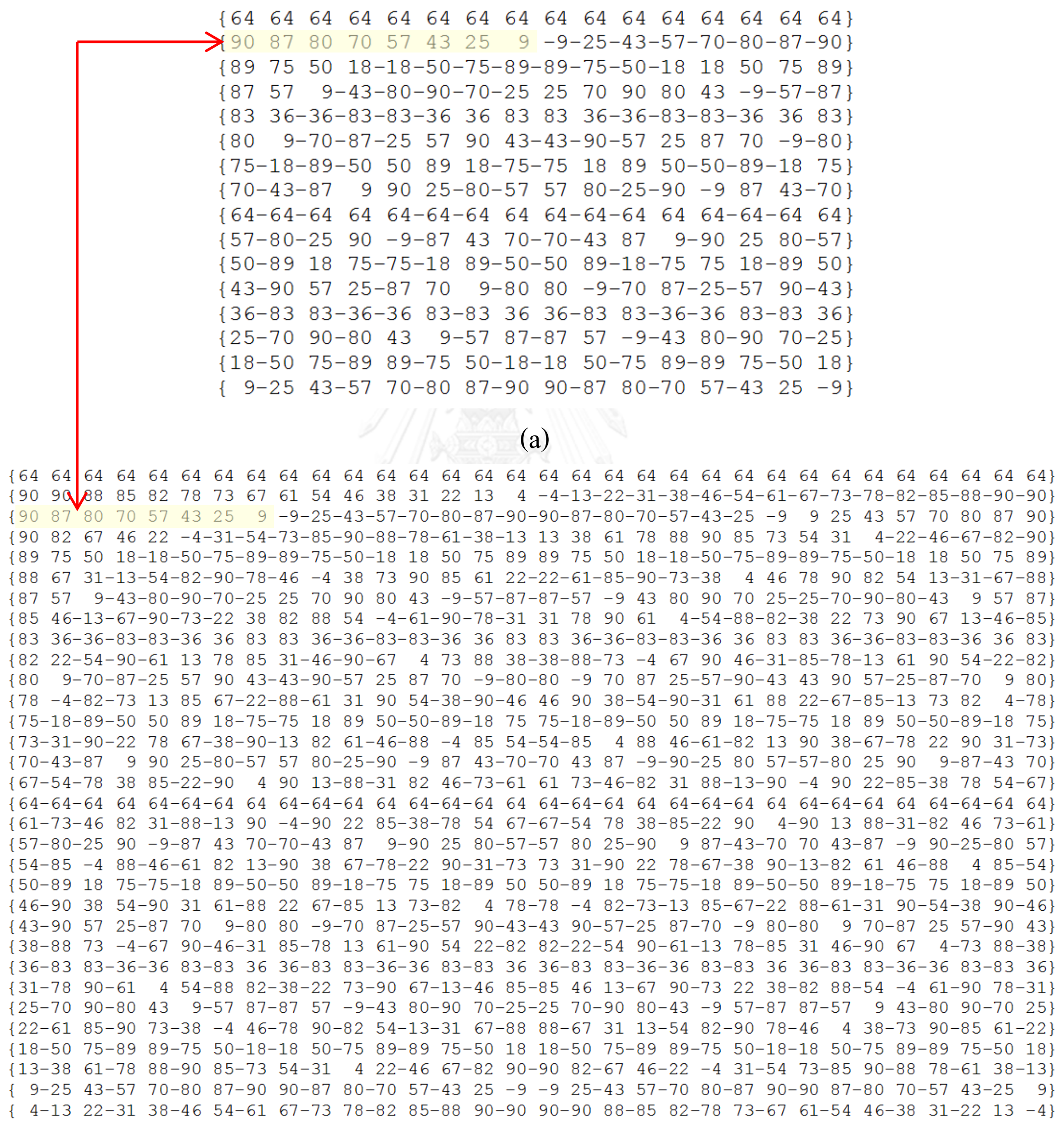
The standard specifies all the elements of the 1-D inverse transform coefficient matrices of the 32-point core transform, as well as the 4-point alternative transform. Coefficients for other lower point, 4-point, 8-point, and 16-point, core transform are just sub-samples of the 32-point coefficient matrix.

The 2-D transform is carried out by computing 1-D transform of each columns of the input data, followed by computing another 1-D transform of each rows of the intermediate result. The 2-D transform computation can be represented in matrix form by,

$$Y = A \cdot X \cdot A^T, \tag{2}$$

where X is an input data vector which is a 9-bit residual data from the prediction step of the HEVC encoder, A is the 1-D transform matrix, and Y is a transform result vector. The intermediate results after each 1-D transform are rounded by adding with an offset and shifting down, to preserve a bit width of 16 after each transform stage [10].

The 1-D forward transform matrix can be obtained by simply transposing its corresponding 1-D inverse transform matrix. The 16-point and 32-point forward transform matrices are shown in Fig.8. Note that the 16-point forward transform matrix is a sub-sampled version of the 32-point forward transform matrix. The elements in the 16-point forward transform matrix are simply the first sixteenth elements of the even rows, i.e. the zeroth row, the second row, and so forth, of the 32-point forward transform matrix.



```
{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}
```

(a)

```
{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}
```

(b)

Fig. 8 (a) 16-point 1-D forward transform matrix

(b) 32-point 1-D forward transform matrix [10].

Since the focus of this thesis is on the core transform of the HEVC, detail about approximation method used for deriving the integer transform from the DCT will be further explained.

The main reason to do transform is to de-correlate residual data to make entropy encoding task more efficient. The best transform for the entropy encoding is actually the Karhunen-Loeve transform (KLT), but DCT is more widely used in image and video compression due to its favorable properties [11], such as implementation friendlyness.

An N-point 1D DCT can be expressed by the following equation,

$$v_i = \sum_{j=0}^{N-1} u_j c_{ij}, \tag{3}$$

where $i = 0, \dots, N-1$, u_j are input vector elements, v_i are output vector elements, and

$$c_{ij} = \frac{P}{\sqrt{N}} \cos\left[\frac{\pi}{N}(j+\frac{1}{2})i\right], \tag{4}$$

where $i, j = 0, \dots, N-1$, and

$$P = \begin{cases} 1, & i = 0 \\ \sqrt{2}, & i > 0 \end{cases} \qquad [11]. \tag{5}$$

Useful properties of the DCT in term of both energy compaction and implementation friendliness are discussed below [11].

1. The DCT have orthogonal basis, which make spectral domain good at de-correlating data.
2. Good energy compaction is provided by the DCT which leads to high compression efficiency.

3. Quantization and de-quantization steps are relatively easy because the DCT have equal norm. If equal quantization error along frequency domain is desirable, quantization matrices will not be needed.
4. Basis vectors of smaller matrix are contained in larger matrix. This fact can be employed to reduce hardware implementation cost by sharing the design among different transform sizes.
5. Only $2^M - 1$ distinct numbers are required to compute an M-point DCT, which reduce hardware complexity.
6. The fact that even basis vectors of the DCT have symmetry and odd basis vectors have anti-symmetry can be used to reduce the required number of operations.
7. The DCT has trigonometric relationships which can be employed to further reduce the required number of operations.

HEVC adopts integer approximation of the DCT for two main reasons. The first reason is to ease the implementation, especially hardware implementation. Another reason is to eliminate a mismatch between different encoder-decoder implementations, which might use different strategies to represent floating-point numbers.

The first goal for approximating DCT in HEVC transform is to preserve properties 4-6, discussed earlier. These properties are important for implementation friendliness, considered to be significant for complex standard like the HEVC. Property 7 is too difficult to be preserved by a transform represented by integer numbers. Properties 1-3 are preserved to some degree, compromised by the number of bits needed to represent the transform coefficients. The final transform standardized into HEVC uses 8 bits to represent the transform coefficients. The exact value of each coefficients are hand-tuned to give the best trade-offs between properties 1-3 [11].

2.4 The Partial Butterfly Algorithm

The HEVC core integer transform can be implemented using two strategies [10]. The first strategy is by using direct matrix multiplication, which facilitate readability of the design. In contrast, the second strategy called the partial butterfly algorithm helps improving the performance of the design. The partial butterfly algorithm is feasible because the HEVC core integer transform has inherited symmetries from the DCT. It should be noted that the partial butterfly algorithm is employed in the HEVC reference software [5].

Principally, the algorithm reduces the number of multiplications by first grouping inputs that have same coefficient value and then factoring out the common coefficient. This approach gives need to a pre-processing of adding or subtracting a set of numbers followed by a multiplication with the common coefficient.

As an example, the sequence for computing the 1-D 16-point transform using the partial butterfly algorithm will be explained. In the first step, the odd and even components, $E, O, EE, EO, EEE, EEO,$ are generated from the inputs, $X_i; i = 0,1, \dots ,7$, as stated by the following equations,

$$\begin{gathered}
E_i = X_i + X_{15-i}; i = 0,1, \dots ,7 , \\
O_i = X_i - X_{15-i}; i = 0,1, \dots ,7 , \\
EE_i = E_i + E_{7-i}; i = 0,1,2,3 , \\
EO_i = E_i - E_{7-i}; i = 0,1,2,3 , \\
EEE_i = EE_i + EE_{3-i}; i = 0,1 , \\
EEO_i = EE_i - EE_{3-i}; i = 0,1.
\end{gathered} \tag{6}$$

In the second step, each component is multiplied with coefficients as required by the algorithm. For example, EEO_0 is multiplied with 83 and 36 to generate two product terms necessary for finding Z_4 and Z_{12} , respectively. In the third step, a number of resulting product terms are added together or subtracted out to produce an output, $Z_i; i = 0,1, \dots ,7.$ These two steps are shown through the following set of equations,

$$
\begin{aligned}
&Z_0 = 64 \cdot EEE_0 + 64 \cdot EEE_1,\\
&Z_8 = 64 \cdot EEE_0 - 64 \cdot EEE_1,\\
&Z_4 = 83 \cdot EEO_0 + 36 \cdot EEO_1,\\
&Z_{12} = 36 \cdot EEO_0 - 83 \cdot EEO_1,\\
\\
&Z_2 = 89 \cdot EO_0 + 75 \cdot EO_1 + 50 \cdot EO_2 + 18 \cdot EO_3,\\
&Z_6 = 75 \cdot EO_0 - 18 \cdot EO_1 - 89 \cdot EO_2 - 50 \cdot EO_3,\\
&Z_{10} = 50 \cdot EO_0 - 89 \cdot EO_1 + 18 \cdot EO_2 + 75 \cdot EO_{3,}\\
&Z_{14} = 18 \cdot EO_0 - 50 \cdot EO_1 + 75 \cdot EO_2 - 89 \cdot EO_3,\\
\\
&Z_1 = 90 \cdot O_0 + 87 \cdot O_1 + 80 \cdot O_2 + 70 \cdot O_3 + 57 \cdot O_4 + 43 \cdot O_5 + 25 \cdot O_6 + 9 \cdot O_7,\\
&Z_3 = 87 \cdot O_0 + 57 \cdot O_1 + 9 \cdot O_2 - 43 \cdot O_3 - 80 \cdot O_4 - 90 \cdot O_5 - 70 \cdot O_6 - 25 \cdot O_7,\\
&Z_5 = 80 \cdot O_0 + 9 \cdot O_1 - 70 \cdot O_2 - 87 \cdot O_3 - 25 \cdot O_4 + 57 \cdot O_5 + 90 \cdot O_6 + 43 \cdot O_7,\\
&Z_7 =\\
&70 \cdot O_0 - 43 \cdot O_1 - 87 \cdot O_2 + 9 \cdot O_3 + 90 \cdot O_4 + 25 \cdot O_5 - 80 \cdot O_6 - 57 \cdot O_7,\\
&Z_9 = 57 \cdot O_o - 80 \cdot O_1 - 25 \cdot O_2 + 90 \cdot O_3 - 9 \cdot O_4 - 87 \cdot O_5 + 43 \cdot O_6 + 70 \cdot O_7,\\
&Z_{11} = 43 \cdot O_0 - 90 \cdot O_1 + 57 \cdot O_2 + 25 \cdot O_3 - 87 \cdot O_4 + 70 \cdot O_5 + 9 \cdot O_6 - 80 \cdot O_7,\\
&Z_{13} = 25 \cdot O_0 - 70 \cdot O_1 + 90 \cdot O_2 - 80 \cdot O_3 + 43 \cdot O_4 + 9 \cdot O_5 - 57 \cdot O_6 + 87 \cdot O_7,
\end{aligned}
\tag{7}
$$

As the final step, these results are rounded by added with an offset and shifted right before being sent out to the outputs, $Y_i; i = 0,1, \ldots ,7$, according to the equation,

$$Y_i = (Z_i + 4) \gg 3; i = 0,1, \ldots ,15. \tag{8}$$

It should be noted that in the example above, intermediate outputs, $Z_i; i = 0,1, \ldots ,15$ are intentionally grouped according to their level of symmetry. Notice the symmetry properties of the 16-point matrix, which is shown in Fig.8(a) :

- row number 0 and 8 have symmetry with symmetry points before coefficient no. 2,4, … , 14.
- row number 4 and 12 have anti-symmetry with symmetry point before coefficient no. 2, 4, … , 14.
- row number 2, 6, 10, 14 have anti-symmetry with symmetry point before coefficient no. 4, 8, and 12.

- other rows, row number 1, 3, … , 15, have anti-symmetry with symmetry point before coefficient no. 8.

2.5 Reconstructed Video Quality Measurement

Measuring quality of a video sequence can be accomplished by averaging the quality measurement scores of each video frame over the entire sequences. Evaluation of the quality of a video frame is image quality measurement. Image quality is hard to be measured because quality issues are typically subjective. The quality depends on the Human Visual System (HVS), the eye and the brain, of each individual. Furthermore, people judge quality of an image differently depending on their current situation. They usually feel better quality when the picture is perceived in relax situations, such as watching television [12].

People also give more attention to highlighted parts of the picture. For example, consider a face in front of a blue background picture. People usually concentrate on the face more than the background. Assume that we equally distort two pictures, the first one in the face part while the other one in the background part. We should get equal quality image, but people perceive better quality from the second picture, because the blurry part is outside their interested spot.

Despite the difficulties explained above, an objective image quality score is invented [12]. This objective image quality score is called Peak Signal-to-Noise Ratio (PSNR). PSNR can be computed by

$$PSNR_{dB} = 10log_{10}\frac{(2^n - 1)^2}{MSE} \qquad (9)$$

,where n is the number of bits used for representing each data pixel of the image, and MSE is the mean square error between the reconstructed image and the original image [12].

PSNR is widely adopted for measuring reconstruction quality in many video coding standards. It can be easily computed from reconstructed videos. The PSNR computation is repeatable since it is an objective measurement. Reconstructed videos with high PSNR usually mean good quality, while reconstructed videos with lower PSNR mean worse quality. Nevertheless, there are no direct conversions between PSNR and subjective quality.

2.6 Coding Efficiency Measurement

Compression performance of video coding standards can be compared using a score called Bjontegaard Delta bitrate (BD-rate). The BD-rate is an average bitrate gain over a specific interval of PSNR. This indicator can be computed using Rate Distortion plots (RD plots). The RD plots are plots between PSNR and bitrate of video sequences, which is illustrated in Fig.9.

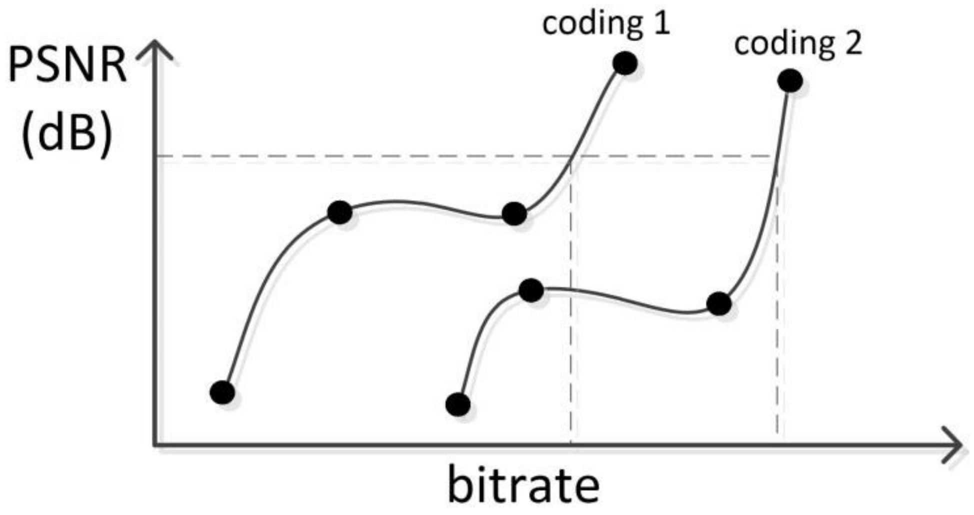


Fig. 9 Rate Distortion Plots (RD plots).

Each curve in Fig.9 is constructed by interpolation of the four data on the curve. The four data are obtained by varying quantization parameter (QP) used in encoding a video sequence. Quantization parameter (QP) affects the step size in the quantization process. After interpolation, the curves are employed for evaluating bitrate saving at each PSNR level. Then, bitrate savings are averaged over the entire range of PSNR to get Bjontegaard Delta bitrate (BD-rate) [13].

In Fig.9, coding scheme 1 has BD-rate gain over the coding scheme 2, because the coding scheme 1 requires less bitrate on average. So, coding 1 has better coding performance than coding 2. Coding 1 and coding 2 can be different video coding standards like the HEVC and the H.264, or the same video coding standard with some different parameters like different profiles of the HEVC.

Chapter 3
The High Throughput Architecture

A 1D HEVC transform architecture is presented. The architecture utilizes the dedicated multipliers inside the DSP slices to gain high throughput design and save general purpose resources [14]. Details about the proposed architecture will be discussed in the next subsections. First, top level block diagram and flow of the design are discussed in section 3.1. Further detail on multiplier sharing scheme is then described in section 3.2. Design results and comparisons with other works are presented in section 3.3. Finally, the pseudo code of the high throughput architecture is given in section 3.4.

3.1 Top Level Block Diagram

Fig.10 shows block diagram of the high throughput architecture. This architecture is a direct implementation of the partial butterfly algorithm discussed earlier with some modifications to make it more efficient for hardware implementation. Modification details will be described further later in this section.

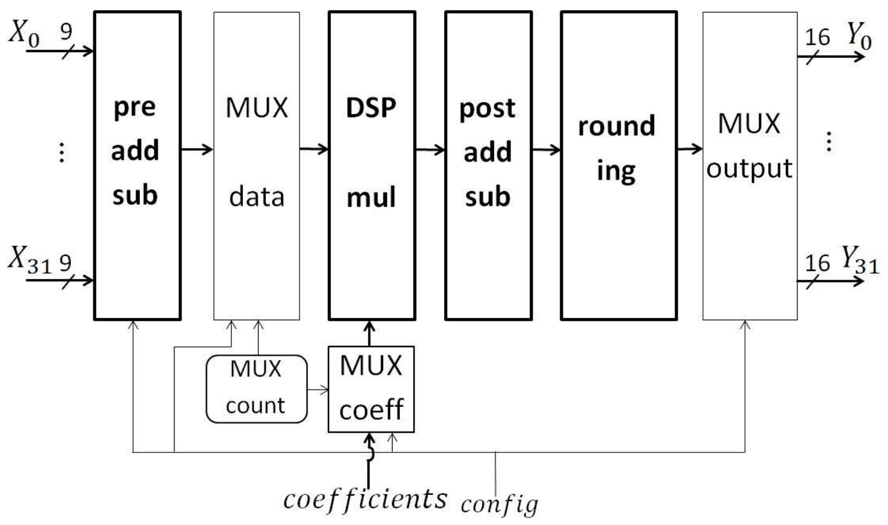


Fig. 10 The top level block diagram [14].

The architecture in Fig.10 always receives thirty-two 9-bit inputs, but can be selected to perform 32-point transform on one set of data, 16-point transform on two sets of data simultaneously, 8-point transform on four sets of data simultaneously, or 4-point transform on eight sets of data simultaneously depending on the 2-bit configuration, *config*. $Y_0, \dots, Y_{31}$ are 16-bit outputs which are matched to the corresponding inputs. For example, in case of performing the 16-point transform, $X_i, Y_i\ ; i = 0,1, \dots,15$ are assigned to the first data set, while $X_i, Y_i\ ; i = 16,17, \dots,31$ are assigned to the second data set.

Four main blocks are highlighted in Fig.10. These four blocks essentially realize the four steps of the partial butterfly algorithm described in the section 2.4. In block *gen odd even*, the odd-even components, such as $E_i, O_i, EE_i, EO_i, EEE_i, EEO_i$ are generated. In block *DSP mul*, each odd and even component is multiplied with coefficients as required. In block *post add sub*, product terms are added or subtracted according to the defined equations such as in (7). The results are then sent to block *rounding*.

For the rest of Fig.9, *MUX data*, *MUX coeff*, and *MUX output*, are multiplexers for implementing the multiplier sharing scheme. Basically, the *MUX data* block selects and latches appropriate data from the *gen odd even* block to internal input registers of the *DSP mul* block. These data are subsequently multiplied with appropriate coefficients in transform matrices selected by the *MUX coeff* block. The *MUX output* is used for multiplexing the output ports among different transform sizes. *MUX count* is a 2-bit counter that controls the sharing of DSP multipliers among four inputs and synchronizes the data flow throughout the design. The necessity of synchronization will be explained later in this subsection. More details about the sharing scheme can be found in Section 3.2.

The *gen odd even* block consists of several pre-adder/subtractor trees, each of which computes a number of odd-even components later used in multiplications. First enhancement for hardware implementation is made here by inserting pipeline registers after each addition and subtraction. These pipeline registers will ensure high throughput and prevent the odd-even generation step from becoming the bottleneck of the system. Fig.11 is an example of such a tree that computes the EEO_0 component of the 16-point transform. Note that the tree also generates the E_0, E_3, E_4, E_7, EE_0, and EE_3 components which will be involved in computing other coefficients.

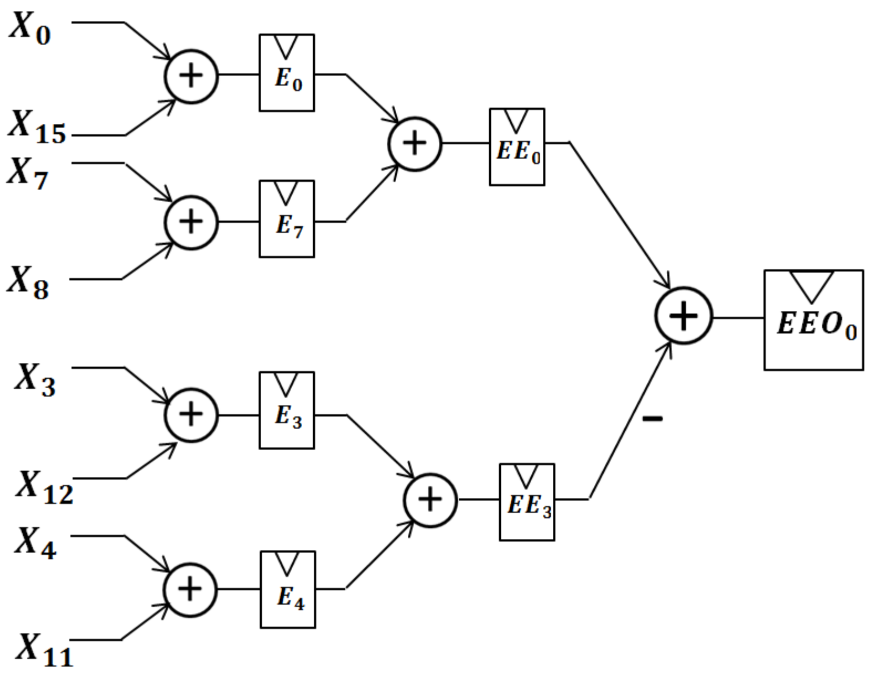


Fig. 11 A part of the hardware used for generating the odd-even components.

The presence of pipeline registers gives rise to the need to synchronize the flow of data in the design, because the $O, EO, EEO,$ and $EEEO$ components generated at different pipeline register levels must be used as the inputs to the *MUX data* block at the same clock cycle. The *MUX count* 2-bit counter is intended to be used for this synchronization purpose. For example, data are latches into input registers of the *DSP mul* block only when the *MUX count* equals three, as shown in timing diagram in Fig.12.

This latching strategy is also used for synchronizing the data path in other part of the design, which includes the step of collecting multiplied results from the DSP slices and the step of gathering outputs of the *post add sub* blocks.

It should be noted that the synchronization could also be accomplished by adding dummy registers. However, the dummy register approach is not suitable for the proposed architecture. Since the design has a large data path, inserting dummy registers will consume a large amount of flip-flops, which are limited usable resources on FPGAs. For each input, X_i, which is nine bits wide, each generated even-odd component will require ten bits to store, which means a ten-bit dummy register must be required for each level of dummies inserted into the pipeline.

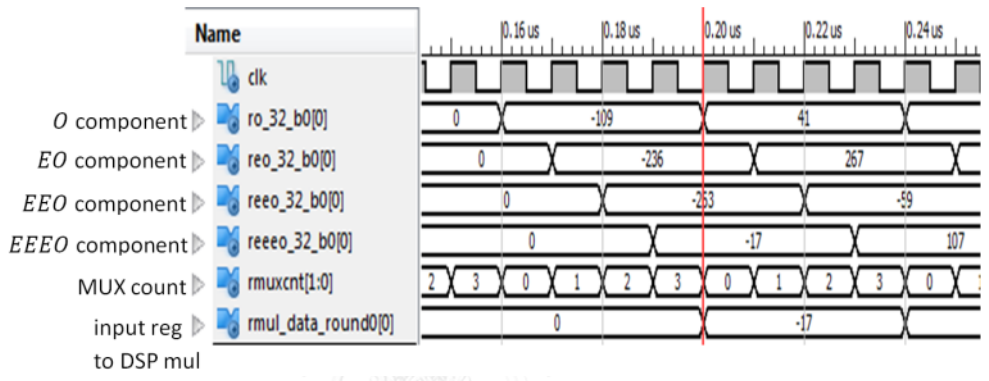


Fig. 12 Synchronization timing diagram. Data are latched into input registers of the DSP mul block only when the MUX count equal three.

Since the odd and even components of each transform size are derived differently, separate hardware are needed for every transform data sets. Eight, four, two, and one odd-even generator modules for the 4-point, 8-point, 16-point, and 32-point are required respectively.

The hard multipliers in Xilinx's DSP slices are exploited in the *DSP mul* block in Fig.10. Structure of the DSP slice, *Xtreme DSP*, in the Spartan-3A family is depicted in Fig.13 [15]. A single DSP slice contains an 18-bit input pre-adder followed by an 18x18 bit two's complement multiplier and a 48-bit sign-extended adder/subtractor/accumulator. The DSP slice is dedicated to DSP related tasks which extensively require a lot of additions and multiplications. Multiple levels of optional pipeline registers in the DSP slice are available and can be programmed by a user through the IP core generator to improve its performance.

The *post add sub* and *rounding* blocks in Fig.10 are now discussed. After all product terms are generated by the *DSP mul* block, they will be sent to the *post add sub* block. The *post add sub* block selects which product terms will be added/subtracted together to generate pre-rounding outputs, Z_i. The *post add sub* block therefore consists of several trees, each of which adds/subtracts a set of given product terms according to the defined equation. Each output Z_i is then sent for rounding in the *rounding* block. The rounding is done by adding an offset to Z_i, then shifting the result down to the specified resolution.

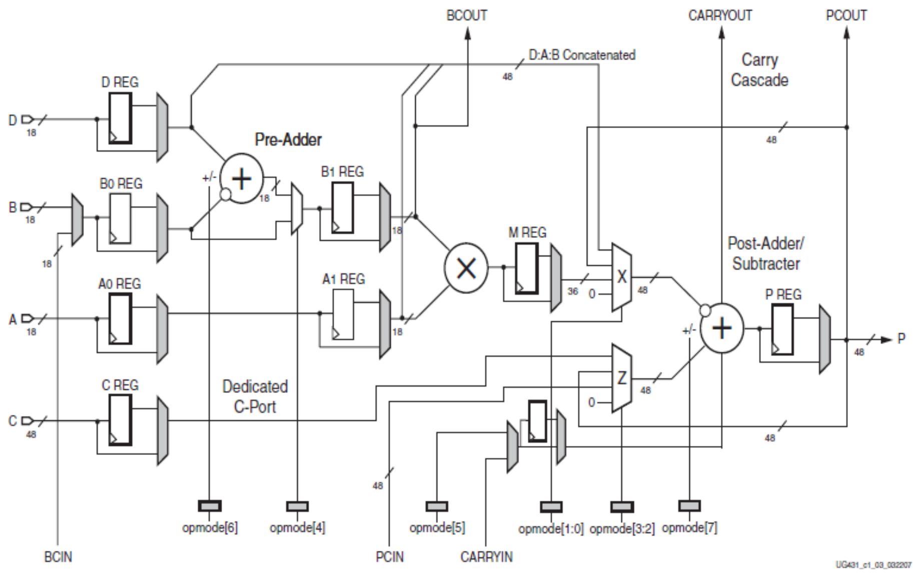


Fig. 13 The XtremeDSP DSP48A [15].

We will follow the process of generating output Y_1 to illustrate the idea of how these two blocks are built. Following the Partial Butterfly algorithm used in the reference software, as described in Section 2.4, the steps for computing the second element or Z_1 of the16-point transform are

$$
\begin{aligned}
O_i &= X_i - X_{15-i} \;\; ; i = 0,1,\dots,7, \\
Z_1 &= 90 \cdot O_0 + 87 \cdot O_1 + 80 \cdot O_2 + 70 \cdot O_3 \\
&\quad + 57 \cdot O_4 + 43 \cdot O_5 + 25 \cdot O_6 + 9 \cdot O_7, \qquad (10) \\
Y_1 &= (Z_1 + 4) \gg 3.
\end{aligned}
$$

Fig.14 depicts parts of the structures in the *post add sub* and *rounding* blocks that related to equation (10) above. For this case, the *post add sub* block is responsible for the aggregation of the $90 \cdot O_0, 87 \cdot O_1, \ldots,$ and $9 \cdot O_7$ product terms and the *rounding* block is responsible for rounding Z_1 to produce Y_1. Note that pipeline registers are again inserted throughout these structures to enhance overall performance as in the *gen odd even* block.

In contrast to separate hardware required in the *gen odd even* block, the *post add sub* modules can be shared among different transform sizes. The sharing is feasible, because the lower point transform matrices are sub-sampled versions of the 32-point transform matrix. The *post add sub* module sharing technique is further explained in Section 3.2.

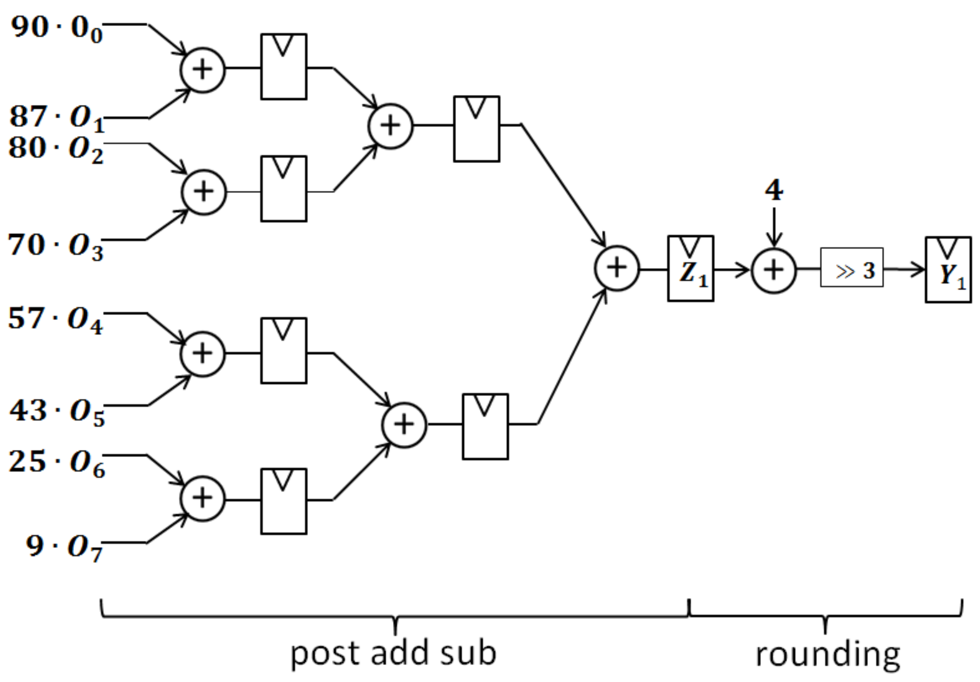


Fig. 14 The structure for computing the post add sub and rounding step of Z_1 and Y_1 of the 16-point transform.

3.2 Multiplier Sharing Scheme

Direct implementation of the partial butterfly algorithm is rather expensive. It requires a total of 308, 84, 20, and 4 multipliers for each 32-point, 16-point, 8-point, and 4-point transform respectively. Note that these numbers do not include any multiplication with coefficients of power of 2 values since they are easily achievable with binary shifts. Although the requirement for multipliers can be fulfilled on some FPGA families, where large number of DSP slices are available, the cost of these ICs are not cheap. To lower the cost, we have targeted our final design on a Spartan-3A, the XC3SD1800A, which contains only 84 DSP slices [16]. It is therefore necessary to invent a multiplier sharing scheme that can accomplish the HEVC transform.

There are two types of sharing in the design. In *intra*-sharing, the multipliers are shared to compute several product terms of the same transform. For example, in 16-point transform (see eq. (7)), the coefficient of value *18* is needed four times to be multiplied with components EO_0 to EO_3, so these four multiplications can effectively share a multiplier. For our design, each multiplier employed in the 32-point transform is reused four times, so the total number of multipliers required is lowered to 77.

In *inter*-sharing, a multiplier is shared among different size transform. For example, the coefficient of value *18* is used in multiplication with O_1 in 8-point transform, EO_1 in 16-point transform, and EEO_1 in 32-point transform, so a multiplier can be shared. All multiplications in the 16-point, 8-point, and 4-point can be shared with the 32-point transform since their coefficients are just sub-sampled.

For a multiplier, the two sharing types can be mixed but we have limited the number of different product terms that a multiplier could generate to four at most for the reason of performance. In total, the amount of multipliers needed for our HEVC transform implementation are just 77, which can be fitted into the targeted Spartan-3A FPGA.

An example of *intra*-sharing multiplier structure is shown in Fig.15. The structure multiplies four elements of the EEO components of the 32-point transform with *18*. The 2-bit *MUX count* selects an appropriate input into the multiplier, and enables the corresponding register to store the result. The multiplier is implemented using the hard multiplier embedded in a DSP slice.

Since transform matrices have symmetry properties with different symmetry points depending on their row number [10], coefficients in the transform matrices can be grouped according to their symmetric properties and their uses as shown in Table 2. For example, coefficients in odd rows of the 32-point transform matrix, coefficient group 4, have anti-symmetry with reflection point between the 15th and the 16th column. The coefficients are also grouped according to their involvement in computing the transform. *Group1* is used in transform of all sizes; *group2* is used in the 8-point, 16-point, and 32-point transforms; *group3* is used in the 16-point and 32-point transforms, while *group4* is used only in the 32-point transform.

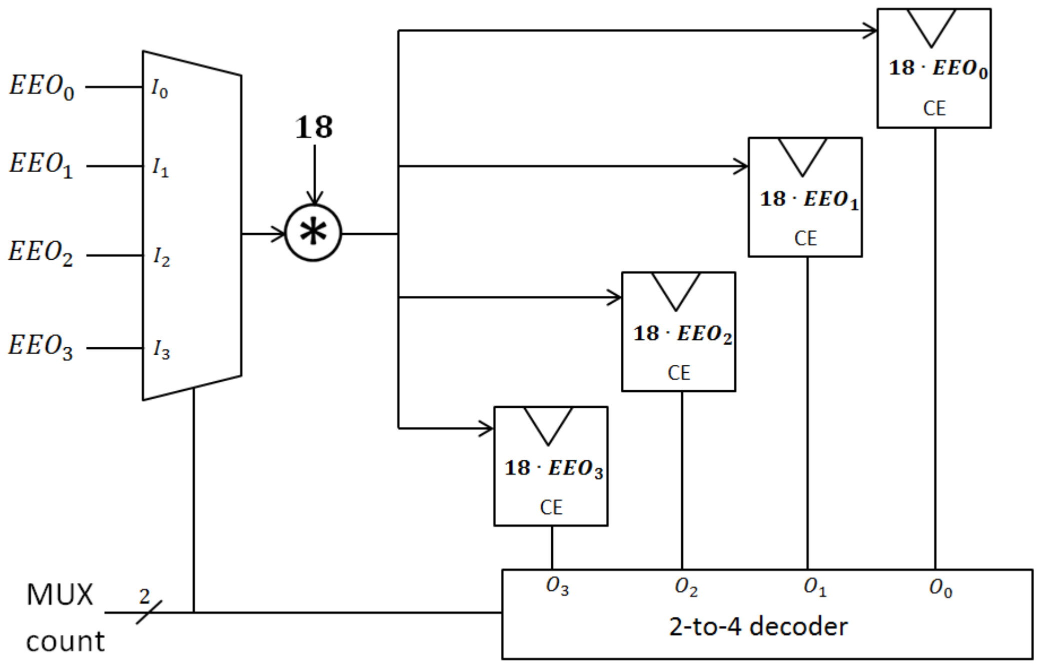
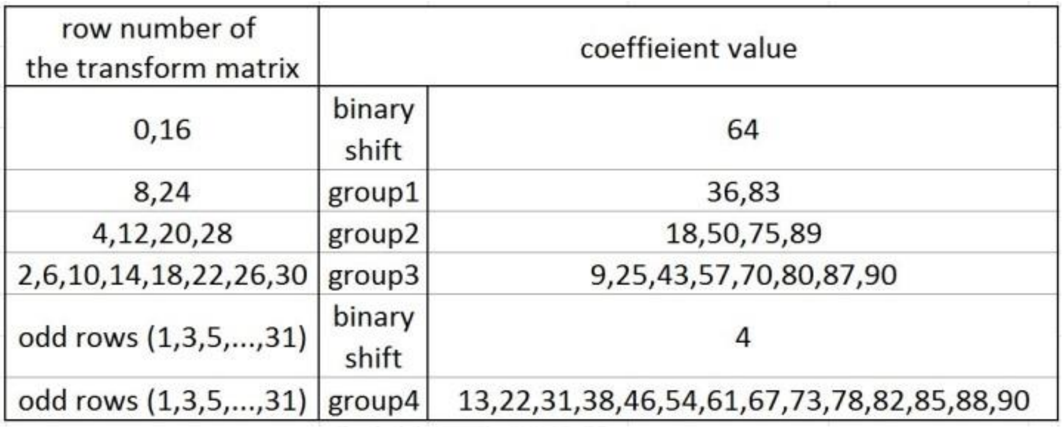


Fig. 15 A four-time sharing multiplier structure.

Multiplier sharing scheme based on these idea is shown in Table 3. The seventy-seven multipliers are reused among different transform sizes. They can be configured to do the multiplication task for 32-point transform on a data set; 16-

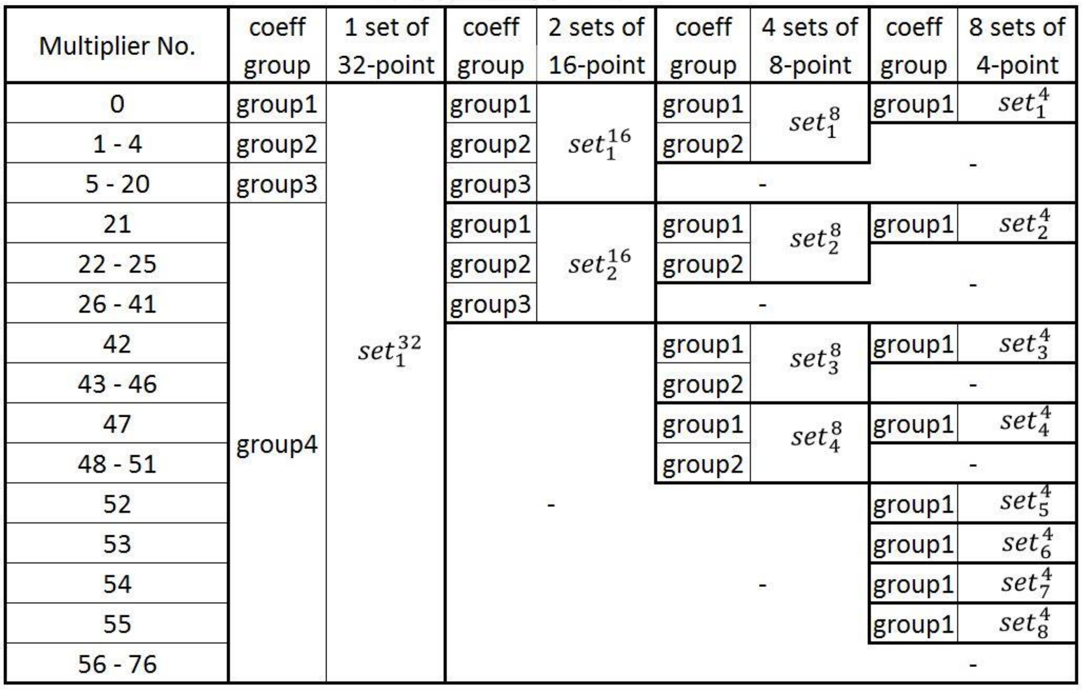
point transform on two data sets; 8-point transform on four data sets; or 4-point transform on eight data sets. The set_i^j symbol in Table 3 represents the i data set of j-point transform, e.g. set_2^{16} is the second data set of 16-point transform.

Table. 2 Coefficients of the 3-point transform matrix [14].



row number of the transform matrix	coeffieient value	
0,16	binary shift	64
8,24	group1	36,83
4,12,20,28	group2	18,50,75,89
2,6,10,14,18,22,26,30	group3	9,25,43,57,70,80,87,90
odd rows (1,3,5,...,31)	binary shift	4
odd rows (1,3,5,...,31)	group4	13,22,31,38,46,54,61,67,73,78,82,85,88,90

Table. 3 The multiplier sharing scheme [14].



Multiplier No.	coeff group	1 set of 32-point	coeff group	2 sets of 16-point	coeff group	4 sets of 8-point	coeff group	8 sets of 4-point
0	group1	set_1^{32}	group1	set_1^{16}	group1	set_1^8	group1	set_1^4
1 - 4	group2		group2		group2		-	
5 - 20	group3		group3		-			
21	group4		group1	set_2^{16}	group1	set_2^8	group1	set_2^4
22 - 25			group2		group2		-	
26 - 41			group3		-			
42			-		group1	set_3^8	group1	set_3^4
43 - 46					group2		-	
47					group1	set_4^8	group1	set_4^4
48 - 51					group2		-	
52					-		group1	set_5^4
53							group1	set_6^4
54							group1	set_7^4
55							group1	set_8^4
56 - 76							-	

Detailed block diagram illustrating sharing details of the proposed architecture is depicted in Fig.16. The multiplier sharing scheme in Table.3 is carefully designed so that *post add sub* block can also be shared among different transform size. For example, consider the following computations:

$$\begin{aligned}
Z_{3,8p} &= 75\cdot O_0 - 18\cdot O_1 - 89\cdot O_2 - 50\cdot O_3,\\
Z_{6,16p} &= 75\cdot EO_0 - 18\cdot EO_1 - 89\cdot EO_2 - 50\cdot EO_3,\\
Z_{12,32p} &= 75\cdot EEO_0 - 18\cdot EEO_1 - 89\cdot EEO_2 - 50\cdot EEO_3,
\end{aligned} \tag{11}$$

where 8*p*, 16*p*, and 32*p* represents the 8-point transform, 16-point transform, and 32-point transform, respectively. Observe that, for these three equations, they are able to share not only multipliers in the *DSP mul* ($75\,\cdot, 18\,\cdot, 89\,\cdot, 50\,\cdot$) blocks but also a *post add sub* structure since they follow a similar pattern of aggregation (-,-,-).

In Fig.16, the *shared mul #1, 2, 3, and 4* and the *post add sub & rounding #33* are reused and shared among $Z_{3,8p}, Z_{6,16p},$ and $Z_{12,32p}$. The *shared mul #1, 2, 3,* and *4* is responsible for multiplying by 18, 50, 75, and 89 respectively, which are coefficients in group2, as shown in Table.2. The precise operation of the structure is controlled by the configuration bits, $config$.

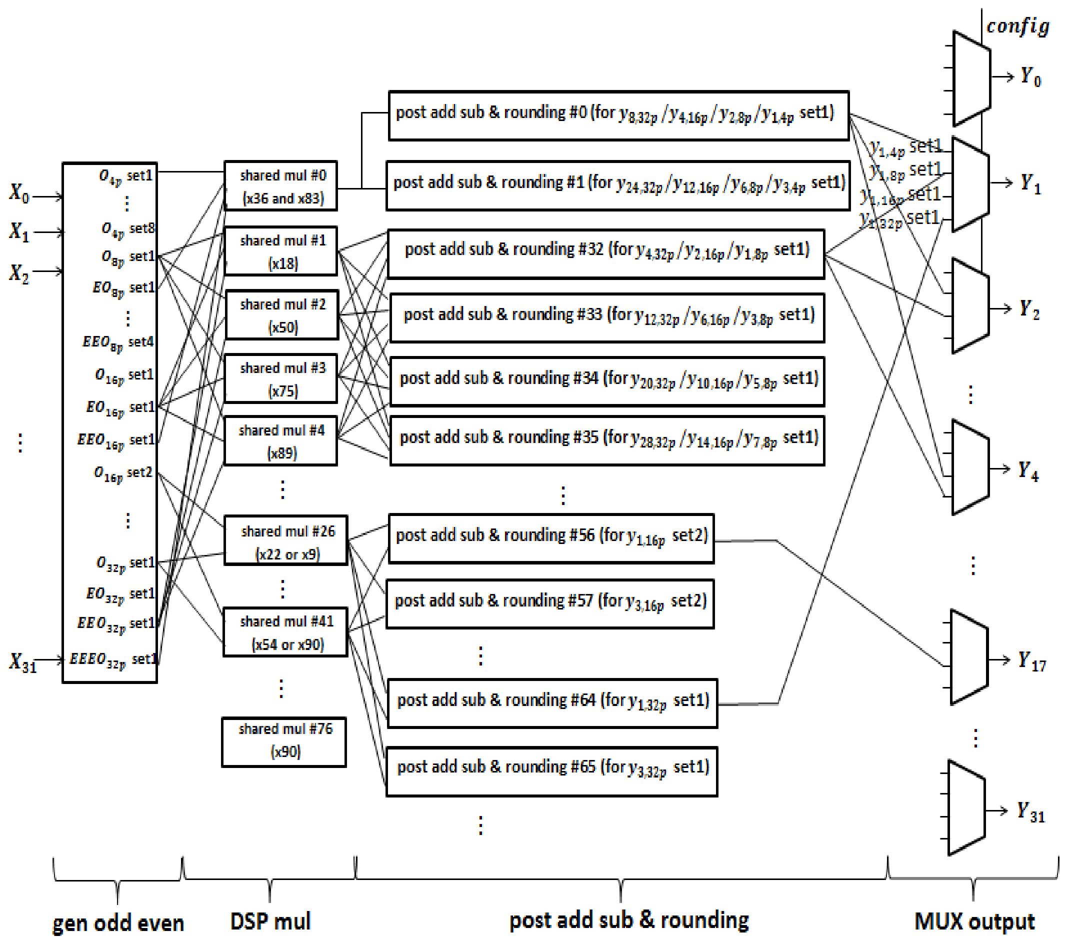
In Table.3, there are some multipliers that have to be shared between different coefficient groups. This is necessary so as to ensure high utilization of a multiplier. For example, the *shared mul #26-41* in Fig.16 are shared between coefficients in *group3* and *group4*. This method of sharing between different coefficient groups, however, is limited to no more than two groups, to maintain high performance of the design.

Consider the *shared mul #26* in Fig.16. It receives both the O components of the 16-point transform of data set 2 and the O components of the 32-point transform of data set 1. An internal multiplexer is provided inside the *shared mul #26* to select whether the input will be multiplied with 22, which is in coefficient group4, or 9, which is in coefficient *group3*.

Outputs of the *shared mul #26* is supplied to two *post add sub & rounding* groups. The first group, *post add sub & rounding #56, #57*, generates results for 16-point transform, while the second group, *post add sub & rounding #64, #65*, generates results for 32-point transform. Finally, the correct results, depending on the transform size as selected by $config$, are assigned to output ports.

Note that multipliers that multiply with coefficient *group1* are special case. Since each of the two coefficients in *group1* (*36* and *83*) is used only twice in multiplying with odd or even components, we can combine their need into using just one multiplier. In Fig.16, multiplication with *36* and *83* is done by a single shared multiplier, *shared mul #0*.

It should also be noted that there are multipliers that are not used in case of 4-point, 8-point, and 16-point transforms, because the 32-point transform demands



X_0
X_1
X_2
X_{31}
O_{4p} set1
O_{4p} set8
O_{8p} set1
EO_{8p} set1
EEO_{8p} set4
O_{16p} set1
EO_{16p} set1
EEO_{16p} set1
O_{16p} set2
O_{32p} set1
EO_{32p} set1
EEO_{32p} set1
$EEEO_{32p}$ set1
shared mul #0 (x36 and x83)
shared mul #1 (x18)
shared mul #2 (x50)
shared mul #3 (x75)
shared mul #4 (x89)
shared mul #26 (x22 or x9)
shared mul #41 (x54 or x90)
shared mul #76 (x90)
post add sub & rounding #0 (for $y_{8,32p}/y_{4,16p}/y_{2,8p}/y_{1,4p}$ set1)
post add sub & rounding #1 (for $y_{24,32p}/y_{12,16p}/y_{6,8p}/y_{3,4p}$ set1)
post add sub & rounding #32 (for $y_{4,32p}/y_{2,16p}/y_{1,8p}$ set1)
post add sub & rounding #33 (for $y_{12,32p}/y_{6,16p}/y_{3,8p}$ set1)
post add sub & rounding #34 (for $y_{20,32p}/y_{10,16p}/y_{5,8p}$ set1)
post add sub & rounding #35 (for $y_{28,32p}/y_{14,16p}/y_{7,8p}$ set1)
post add sub & rounding #56 (for $y_{1,16p}$ set2)
post add sub & rounding #57 (for $y_{3,16p}$ set2)
post add sub & rounding #64 (for $y_{1,32p}$ set1)
post add sub & rounding #65 (for $y_{3,32p}$ set1)
$config$
$y_{1,4p}$ set1
$y_{1,8p}$ set1
$y_{1,16p}$ set1
$y_{1,32p}$ set1
Y_0
Y_1
Y_2
Y_4
Y_{17}
Y_{31}
gen odd even
DSP mul
post add sub & rounding
MUX output

Fig. 16 Sharing details of the proposed architecture.

much bigger number of multiplications than the rest. As we can see, the number of multiplier needed for the 32-point transform is 77, while two sets of 16-point transform requires only 21x2 = 42 multipliers. Therefore, we have plenty of freedom to choose the sharing combination that will provide good performance.

3.3 Design Results

The proposed architectures are described in VHDL and synthesized by the Xilinx ISE 14.4, targeting Spartan-3A FPGA. Simulation results of the architecture on FPGA are then compared with the results retrieved from the HEVC reference software [5] to verify its correctness.

Design summary on FPGA is shown in Table.4. The primary usage of the slice flip flops is for pipelining purpose. Four-input LUTs are used mostly for constructing the adders/subtractors, and multiplexing the data path. The saving on LUTs comes from the sharing of the *post add sub* trees as discussed earlier. Seventy-seven DSP slices are consumed for the multiplication step. Note that, with multiplier sharing scheme, the HEVC transform can be completely implemented on a single Spartan-3A FPGA.

Table. 4 The preliminary design summary.

Technology	Spartan-3A
4-input LUTs	15521/33280 (46%)
Slice flip flops	20818/33280 (62%)
DSP slices	77/84 (92%)
Max Frequency (MHz)	211.5
Throughput (Msamples/s)	1692
Supporting Video Format	7680x4320 @30 fps

Fig.17 is the timing simulation result obtained from Xilinx ISE 14.4. The design reads 32 data samples simultaneously in each round or every four cycles, and is able to perform 8 sets of 4-point transform; 4 sets of 8-point transform; 2 sets of 16-point transform; or 1 set of 32-point transform as selected by *config*. The multiplication step takes four clock cycles to complete, due to the sharing of the multipliers. After the initial latency of 18 cycles, new outputs are produced every four cycles.

At maximum operating frequency of 211.5 MHz, with new results produced at the rate of 32 samples per four clock cycles, our proposed design can reach a maximum throughput of 1692 Msamples/s, which is enough to support 8K videos at 30 frames/s. As a note, for 4:2:0 color sampling format, the 8K video at 30 frames/s needs a processing rate of 1.5x7680x4320x30, which is approximately 1500 Msamples/s.

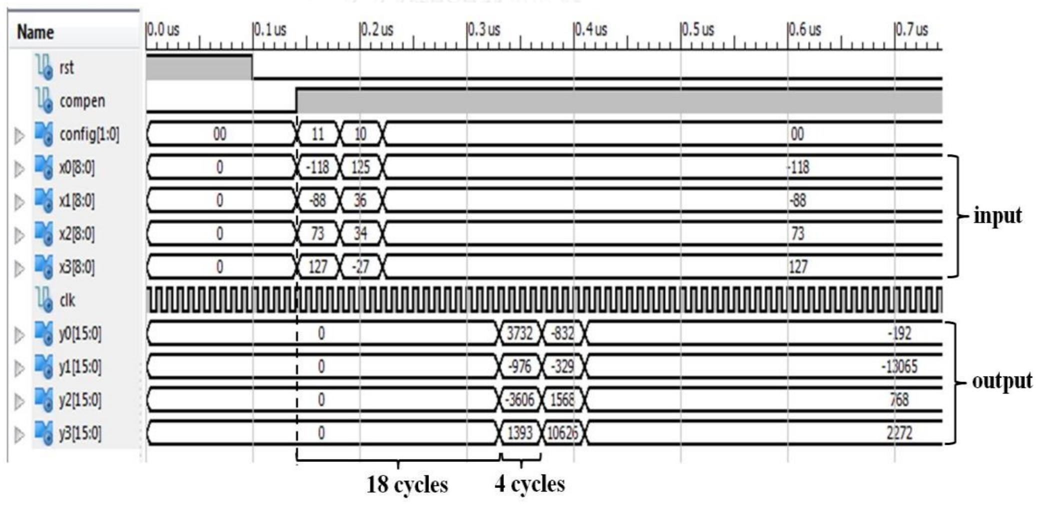


Fig. 17 The simulation result.

Comparisons with the other 1-D transform designs for HEVC are shown in Table.5. Since the designs have been targeting many different technologies, it is difficult to make a concrete conclusion. In terms of resources, the area of an ASIC design is usually reported as the number of equivalent gates. On the other hand, FPGA tools report resource usage based on the primary elements in a particular FPGA fabric, which are varied for different vendors or families. However, for performance aspect, the throughput can still be roughly compared. The results in Table.5 show that the proposed architecture has more throughput than any previous designs on FPGAs [6], [7].

Table. 5 Comparison with other works.

Design	Technology	Function	Throughput (Msamples/s)	Supporting Video Format
Jeske et al. [6]	Cyclone II (90 nm)	1-D 16x16	376.2	3840x2160 @30 fps
Zhao et al. [7]	Cyclone IV (60 nm)	2-D all sizes	238.13	-
Park et al. [8]	ASIC 150 nm	1-D all sizes	1,504	7680x4320 @30 fps
Meher et al. [9]	ASIC 90 nm	1-D all sizes	2,990	7680x4320 @60 fps
Proposed architecture	Spartan-3A (90 nm)	1-D all sizes	1,692	7680x4320 @30 fps

Consider the implementations on 90 nm FPGAs, Cyclone II [17] and Spartan-3A [16]. The proposed architecture has four times and seven times more throughput than the work in [6] and [7] respectively. This significant increase in performance is due to the use of the dedicated resources, DSP slices, in FPGAs.

It must be pointed out that the resource utilization of the proposed architecture and the designs in [6] and [7] should be compared with care, since they are targeted at different FPGA vendors. The previous designs also did not exploit any dedicated hardware blocks. Nevertheless, it can be inferred that implementing the multiplication tasks on the dedicated hardware blocks will help saving the general resources such as LUTs and flip flops.

3.4 Pseudo Code of the High Throughput Architecture

Fig.18-19 in this section present the pseudo code of the high throughput architecture. Further details about the multiplier sharing scheme of this architecture can be found in Appendix B.

```
module partialButterfly_high_tp

  input  (reset, clock, input_combination, input)
  output (output)

  ---- CONTROL PATH
  process : counter for controlling sharing of dedicated multipliers and synchronizing the data path
    if reset is enable then
      mux_counter ← 3
    else
      mux_counter ← mux_counter + 1
    end if
  end process

  ---- DATA PATH
  -- 1st step : generate odd even components
  process #1 : generate O components
    if mux_counter equals 3 then
      for i ← 0 to 15 do
        $O_i$ ← $input_i$ − $input_{31-i}$
      end for
    end if
  end process

  process #2 : generate E components
    if mux_counter equals 3 then
      for i ← 0 to 15 do
        $E_i$ ← $input_i$ + $input_{31-i}$
      end for
    end if
  end process

  process #3 : generate EO components
    for i ← 0 to 7 do
      $EO_i$ ← $E_i$ − $E_{15-i}$
    end for
  end process

  process #4 : generate EE components
    for i ← 0 to 7 do
      $EE_i$ ← $E_i$ + $E_{15-i}$
    end for
  end process

  process #5 : generate EEO components
    for i ← 0 to 3 do
      $EEO_i$ ← $EE_i$ − $EE_{7-i}$
    end for
  end process

  process #6 : generate EEE components
    for i ← 0 to 3 do
      $EEE_i$ ← $EE_i$ +$EE_{7-i}$
    end for
  end process

  process #7 : generate EEEO components
    for i ← 0 to 1 do
      $EEEO_i$ ← $EEE_i$ − $EEE_{3-i}$
    end for
  end process

  process #8 : generate EEEE components
    for i ← 0 to 1 do
      $EEEE_i$ ← $EEE_i$ + $EEE_{3-i}$
    end for
  end process

  -- 2nd step : multiplication
```

Fig. 18 Pseudo code of the High Throughput architecture.

```
process #9 : multiplexers for selecting multiplicands for dedicated multipliers
  for k ← 0 to 76 do
    case input_combination, mux_counter of
      $mul\_data_k$ ← an appropriate component according to the multiplier sharing scheme
    end case
  end for
end process

process #10 : multiplexers for selecting multipliers for dedicated multipliers
  for k ← 0 to 76 do
    case input_combination, mux_counter of
      $mul\_coeff_k$ ← an appropriate component according to the multiplier sharing scheme
    end case
  end for
end process

process #11 : initiate dedicate multipliers
  component k ← 0 to 76 port map (
    multiplier    ← $mul\_data_k$,
    multiplicand  ← $mul\_coeff_k$,
    product   ← $mul\_output_k$
  )
end process

process #12 : collecting multiplied terms
  for k ← 0 to 76 do
    case mux_counter of
      collect outputs from $mul\_output_k$ to an appropriate multiplied term register
    end case
  end for
end process

process #13 : computing pseudo multiplication
  if mux_counter equals 3 then
    for all element of pseudo-multiplied term registers
      compute pseudo-multiplied terms by binary shifting
    end for
  end if
end process

-- 3rd step : post addition subtraction
process #14 : post additions and subtractions
  if mux_counter equals 0 then
    for all s in possible set number do
      for i ← 0 to 31 do
        $y_{si}$ ← an aggregation tree of multiplied terms according to the transform matrix
      end for
    end for
  end if
end process

-- 4th step : rounding and output multiplexer
process #15 : output multiplexing and rounding
  for i ← 0 to 31 do
    case config of
      0  :  $output_i \leftarrow (y_{\lfloor i/4 \rfloor,\ 8*mod(i,4)} + 1) >> 1$
      1  :  $output_i \leftarrow (y_{\lfloor i/8 \rfloor,\ 4*mod(i,8)} + 2) >> 2$
      2  :  $output_i \leftarrow (y_{\lfloor i/16 \rfloor,\ 2*mod(i,16)} + 4) >> 3$
      3  :  $output_i \leftarrow (y_{\lfloor i/32 \rfloor,\ i} + 8) >> 4$
    end case
  end for
end process

end module
```

Fig. 19 Pseudo code of the High Throughput architecture (continue).

Chapter 4
The Flexible Input Architecture

The second forward transform architecture designed in this thesis is the flexible input architecture [18]. Transform step of the HEVC processes on transform units of a video sequence. Transform units are derived from coding units using residual quad-tree partitioning as described earlier in section 2.2.

Fig.20 shows a coding unit of size 32x32, which is partitioned into 13 different size transform units. Consider each column of the coding unit, there are several input combinations for the transform step. For example, the first column consists of a set of 8 inputs, followed by two sets of 4 inputs and a set of 16 inputs. This input combination is represented as (8, 4, 4, 16). Input combination of (16, 8, 8) can be observed from the middle column of this coding unit.

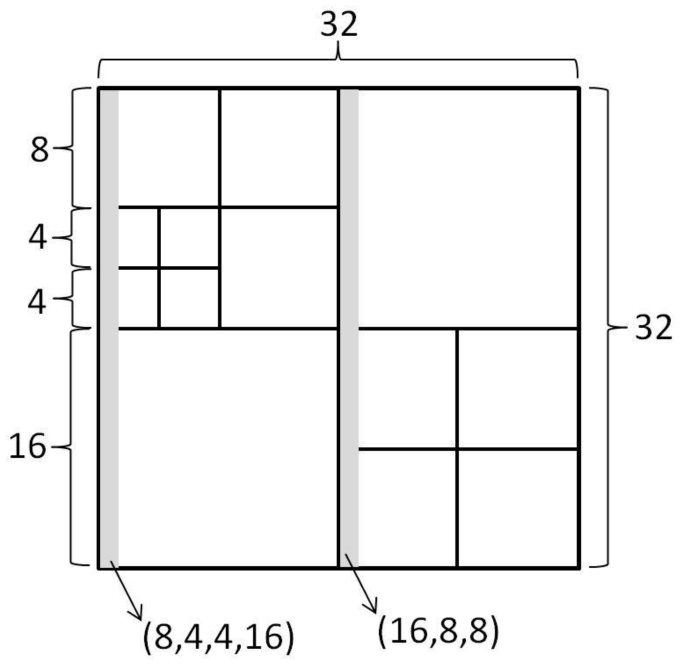


Fig. 20 A coding unit of size 32x32 pixels, partitioned into 13 transform units of different sizes.

The flexible input architecture is an improved version of the high throughput architecture, presented in chapter 3. Basically, the flexible input architecture can compute forward HEVC integer transform of any input combinations resulting from a residual quad-tree partitioning [18], while the high throughput architecture can compute only four input combinations. Besides that, multiplier sharing scheme used in the new architecture is simpler than the original architecture.

In this chapter, the flexible input architecture is discussed. The design inside the architecture itself is explained in section 4.1. Section 4.2 shows the design results compared with other works. Finally, the pseudo code of the flexible input architecture is given in section 4.3.

4.1 The Improved Architecture

Top level block diagram of the flexible input architecture is shown on Fig.21. The diagram is similar to the diagram of the high throughput architecture, depicted in Fig.10, but there are two main differences. The first difference is that the configuration bits, config, are composed of 7 bits in the new architecture, instead of 2 bits in the first architecture, which make it feasible to represent all possible input combinations. Other difference is that the *config* in the new architecture also controls the *gen odd even* block.

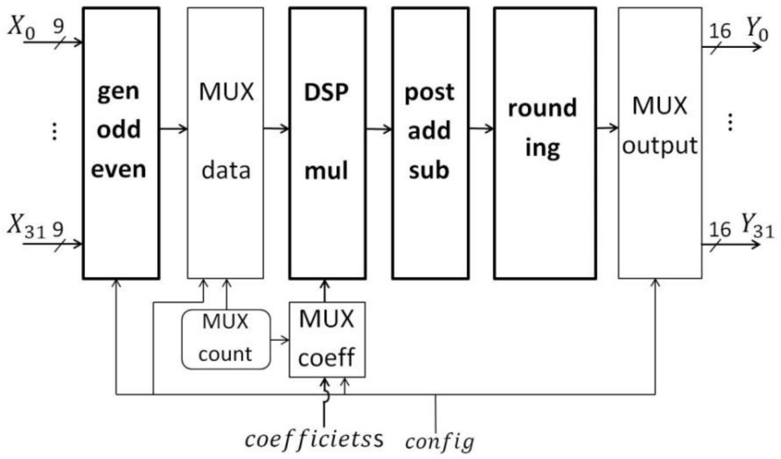


Fig. 21 Top level block diagram of the Flexible Input architecture.

The flexible input architecture uses the partial butterfly algorithm as basis of the design. However, in small-size transform, hardware employed in the first three steps of the algorithm is completely reused by larger-size transform, as proposed in [9].

To describe sharing of hardware between different transform sizes, consider the equations for computing 4-point and 8-point transform as follow,

$$\begin{aligned} E_{i,4p} &= X_{i,4p} + X_{3-i,4p} \;; i = 0,1, \\ O_{i,4p} &= X_{i,4p} - X_{3-i,4p} \;; i = 0,1, \end{aligned} \tag{12}$$

$$\begin{aligned} Z_{0,4p} &= 64 \cdot E_{0,4p} + 64 \cdot E_{1,4p}, \\ Z_{2,4p} &= 64 \cdot E_{0,4p} - 64 \cdot E_{1,4p}, \\ Z_{1,4p} &= 83 \cdot O_{0,4p} + 36 \cdot O_{1,4p}, \\ Z_{3,4p} &= 36 \cdot O_{0,4p} - 83 \cdot O_{1,4p}, \end{aligned} \tag{13}$$

$$\begin{aligned} E_{i,8p} &= X_{i,8p} + X_{7-i,8p} \;; i = 0,1,2,3, \\ O_{i,8p} &= X_{i,8p} - X_{7-i,8p} \;; i = 0,1,2,3, \\ EE_{i,8p} &= E_{i,8p} + E_{3-i,8p} \;; i = 0,1, \\ EO_{i,8p} &= E_{i,8p} - E_{3-i,8p} \;; i = 0,1, \end{aligned} \tag{14}$$

$$\begin{aligned} Z_{0,8p} &= 64 \cdot EE_{0,8p} + 64 \cdot EE_{1,8p}, \\ Z_{4,8p} &= 64 \cdot EE_{0,8p} - 64 \cdot EE_{1,8p}, \\ Z_{2,8p} &= 83 \cdot EO_{0,8p} + 36 \cdot EO_{1,8p}, \\ Z_{6,8p} &= 36 \cdot EO_{0,8p} - 83 \cdot EO_{1,8p}, \end{aligned} \tag{15}$$

$$\begin{aligned} Z_{1,8p} &= 89 \cdot O_{0,8p} + 75 \cdot O_{1,8p} + 50 \cdot O_{2,8p} + 18 \cdot O_{3,8p}, \\ Z_{3,8p} &= 75 \cdot O_{0,8p} - 18 \cdot O_{1,8p} - 89 \cdot O_{2,8p} - 50 \cdot O_{3,8p}, \\ Z_{5,8p} &= 50 \cdot O_{0,8p} - 89 \cdot O_{1,8p} + 18 \cdot O_{2,8p} + 75 \cdot O_{3,8p}, \\ Z_{7,8p} &= 18 \cdot O_{0,8p} - 50 \cdot O_{1,8p} + 75 \cdot O_{2,8p} - 89 \cdot O_{3,8p}. \end{aligned} \tag{16}$$

The equations show only the first three steps of the partial butterfly algorithm, among which hardware will be shared.

Reuse of hardware for the first three steps of the partial butterfly algorithm between 4-point and 8-point transform is illustrated in Fig.22. Basically, hardware for computing the 4-point transform can be reused for computing some intermediate results of the 8-point transform. To be more specific, if $E_{i,8p}; i = 0,1,2,3$ are inserted into a 4-point transform instead of $X_{i,4p}; i = 0,1,2,3,$ the intermediate results $Z_{i,4p}; i = 0,1,2,3$ of the 4-point transform will actually be the even-row intermediate results $Z_{2i,8p}; i = 0,1,2,3$ of the 8-point transform.

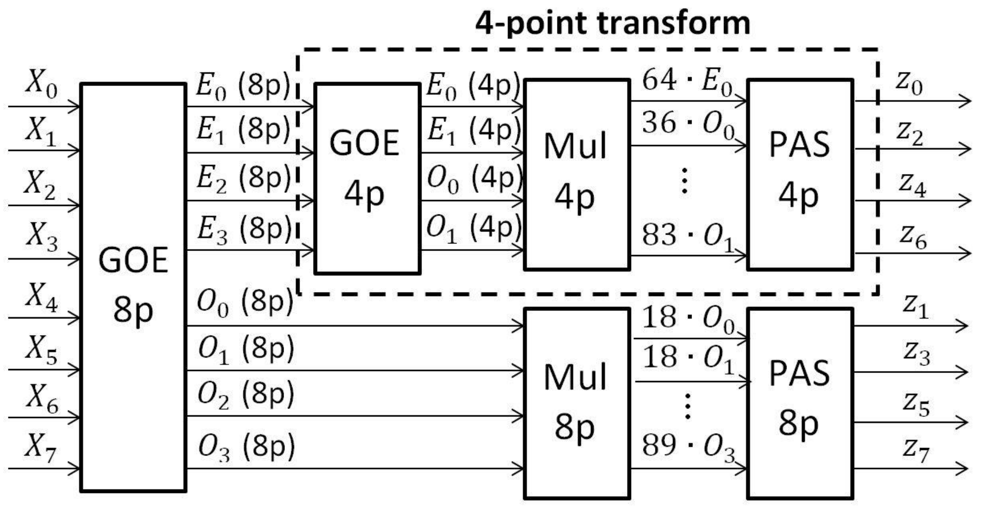


Fig. 22 Hardware reusing between 4-point and 8-point transform.

We shall now discuss the controlling scheme using *config* bit. Residual quad-tree partitioning divides residual data blocks into four smaller blocks recursively in two-dimension. There are only 26 possible input combinations, which can also be viewed as one-dimension partitioning. All possible input combinations are listed in Table.6 with their corresponding *config* control bit. The method for encoding the *config* will be described next.

The method used for encoding possible input combinations into *config* bit is called the configuration encoding scheme. The scheme, depicted in Fig.23, is basically a direct representation of the residual quad-tree partitioning in one-dimension. Each possible partitioning point is represented by a node in a binary tree, and each node is represented by a bit in *config*. A total number of 7 bits is required in a *config* bit.

Table. 6 Possible input combinations resulted from residual quad-tree partitioning.

	Input combination representation	$config$
0	(32)	"000 0000"
1	(16, 16)	"000 0001"
2	(16, 8, 8)	"000 0101"
3	(16, 8, 4, 4)	"100 0101"
4	(16, 4, 4, 8)	"010 0101"
5	(16, 4, 4, 4, 4)	"110 0101"
6	(8, 8, 16)	"000 0011"
7	(8, 4, 4, 16)	"001 0011"
8	(4, 4, 8, 16)	"000 1011"
9	(4, 4, 4, 4, 16)	"001 1011"
10	(8, 8, 8, 8)	"000 0111"
11	(8, 8, 8, 4, 4)	"100 0111"
12	(8, 8, 4, 4, 8)	"010 0111"
13	(8, 8, 4, 4, 4, 4)	"110 0111"
14	(8, 4, 4, 8, 8)	"001 0111"
15	(8, 4, 4, 8, 4, 4)	"101 0111"
16	(8, 4, 4, 4, 4, 8)	"011 0111"
17	(8, 4, 4, 4, 4, 4, 4)	"111 0111"
18	(4, 4, 8, 8, 8)	"000 1111"
19	(4, 4, 8, 8, 4, 4)	"100 1111"
20	(4, 4, 8, 4, 4, 8)	"010 1111"
21	(4, 4, 8, 4, 4, 4, 4)	"110 1111"
22	(4, 4, 4, 4, 8, 8)	"001 1111"
23	(4, 4, 4, 4, 8, 4, 4)	"101 1111"
24	(4, 4, 4, 4, 4, 4, 8)	"011 1111"
25	(4, 4, 4, 4, 4, 4, 4, 4)	"111 1111"

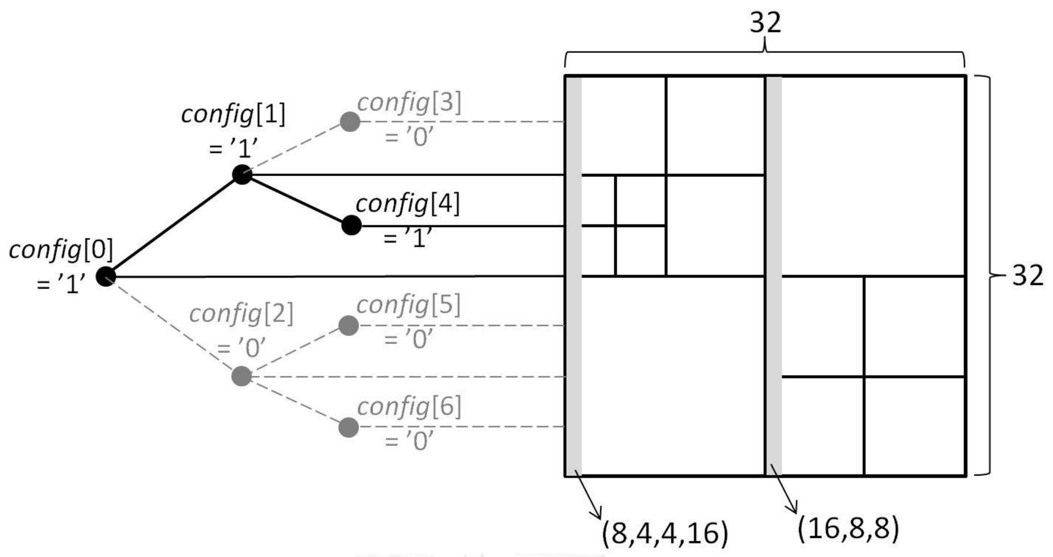


Fig. 23 7-bit configuration encoding scheme.

Consider the following example for a further explanation on the configuration encoding scheme. In Fig.23, the block picture on the right-hand side is the same as the coding unit in Fig.20, described earlier. The first column of this block contains input combination of (8, 4, 4, 16). This input combination can be represented by the *config* on the left-hand side. A config bit assigned at a partitioning point indicates whether a partitioning has occurred ("1") or not ("0"). In Fig.23, the *config* bits that are corresponding to the active partitioning points are *config*[0], *config*[1], and *config*[4], so the *config* equals "0010011". This *config* pattern is also highlighted in Table.6. Note that *config* bits are arranged by the most significant bit first.

Despite the number of bits in *config* are seven, it should be noticed that the number of possible input combinations are not 128, but only 26. This less number of input combinations is because there are dependencies between bits of *config*. From a partitioning resulted from a residual quad-tree, *config*[1] or *config*[2] cannot be '1' while *config*[0] equals '0'. This fact is also true for other *config* bits, bits that are corresponding to a lower level in the binary tree cannot be '1', while their upper level bits are '0'. This restriction reduces the number of patterns represented by *config* to 26.

Fig.24 describes further detail about the flexible input architecture. Each big dash box represent each step of the architecture, which are *gen odd even* (GOE), *DSP mul* (Mul), *post add sub* (PAS), *rounding*, and *MUX output* (MUXout). The partial butterfly algorithm with sharing between different transform sizes, as shown in Fig.22, is used as basis of the design.

A total number of eight hardware sets is present in the flexible input transform architecture. The hardware sets are composed of a set for 32-point transform (set0), a set for 16-point transform (set1), two sets for 8-point transform (set2, set3), and four sets for 4-point transform (set4, set5, set6, set7). It is worth noting that part of the hardware set for computing large-size transform can also be used for computing smaller-size transform. For example, part of set0, which are for computing a 32-point transform, can also be used for computing 16-point, 8-point, or 4-point transform. Set1, which are for computing a 16-point transform, can also be used for computing 8-point or 4-point transform.

The eight hardware sets in Fig.24 are carefully arranged. The arrangement not only allows the architecture to compute transform of any possible input combination, but also directly mapped the input combination from its configuration encoding scheme. Notice how *config* bits are used to control multiplexers inside the *gen odd even* (GOE) block. The arrangement of *config* bits on the multiplexers is in the same position of their arrangement on the configuration encoding scheme.

To further explain about the hardware set arrangement, consider the following example. The active hardware sets for computing transform of the input combination of (8, 4, 4, 16) are highlighted in Fig.24. The hardware set0 is used for computing the 8-pint transform. The hardware set2 is used for computing the first 4-point transform. The *config*[4] is equal ‘1’, so inputs to the hardware set2 are $X_8 - X_{11}$, not $X_8 - X_{15}$. The *config*[1] is equal ‘1’, so inputs to the hardware set0 are $X_0 - X_7$, not $X_0 - X_{31}$ or $X_0 - X_{15}$. Only the first eight outputs from the *post add sub* (PAS) of the hardware set0 are actually used, not all the outputs.

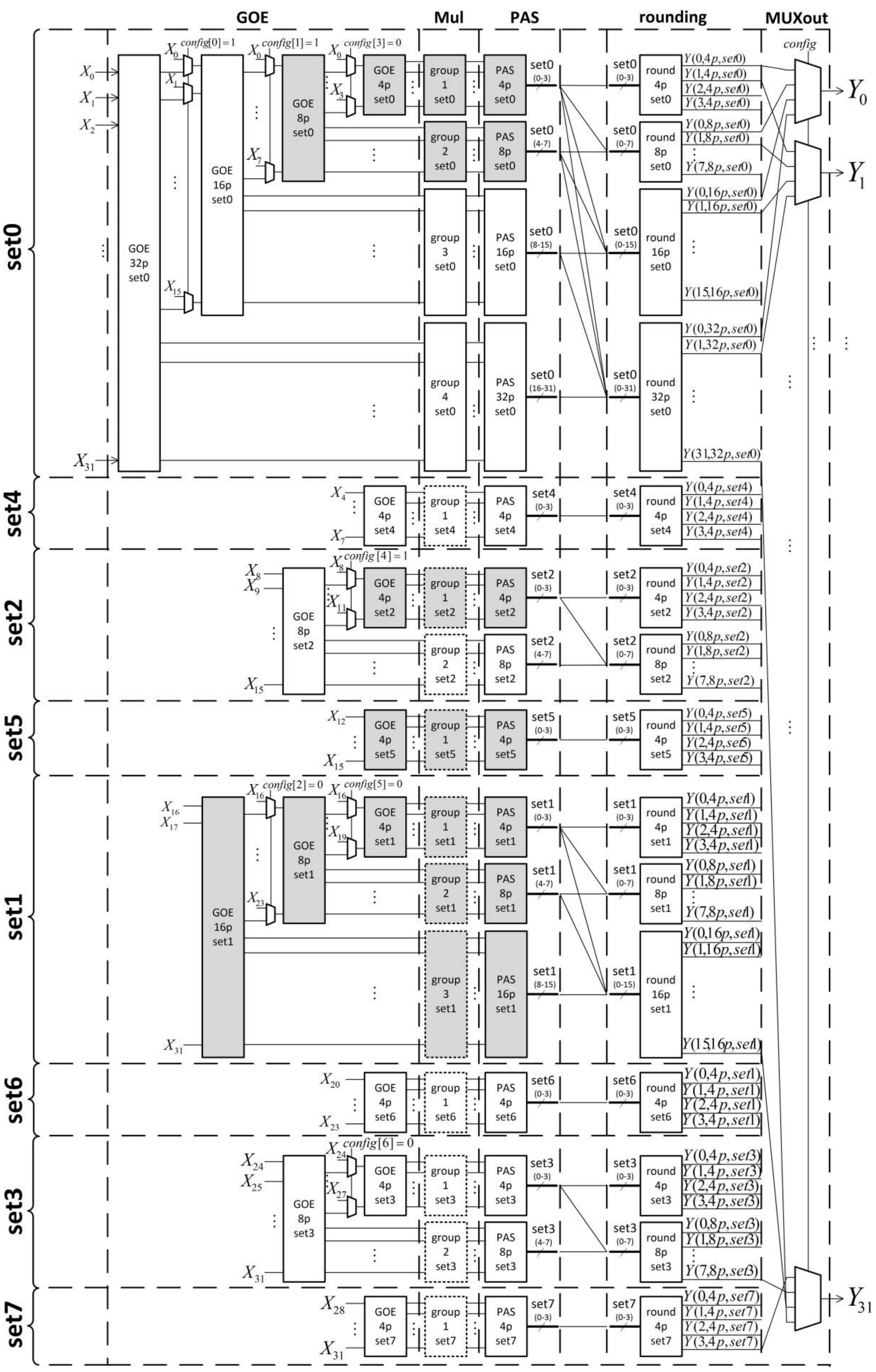


Fig. 24 Details of hardware inside the Flexible Input architecture.

It should be noted that input $X_0 - X_{31}$ are also routed to some multiplexers inside the *gen odd even* (GOE) block. For example, the GOE 4p set0 block can receive inputs of $X_0 - X_3$ if *config*[3] equals ‘1’. Many connections are omitted for clarity of the diagram.

The multiplication step, *DSP mul* (Mul), is carried out using dedicated multipliers inside the DSP slices to get a high performance design, as done by the high throughput architecture in chapter 3. The dedicated multipliers are shared in similar manner as in the high throughput architecture.

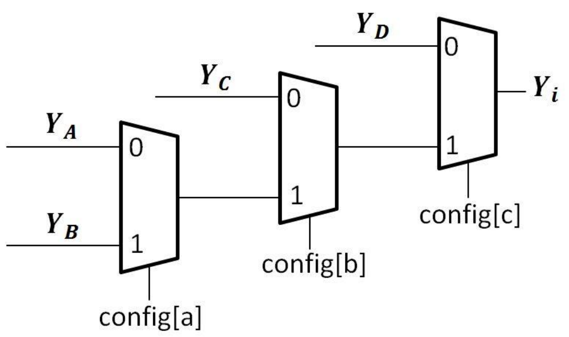
Multiplier coefficients are grouped according to their level of symmetry as shown in Table.2 in chapter 3. The coefficients that are power of two, 4 and 64, are separated out from other coefficients because multiplying with them can be carried out using binary shifting. From the 32-point transform matrix, presented in section 2.3, a lot of coefficient symmetries can be observed. Odd rows contain coefficient asymmetric points at the middle column. The coefficients in odd rows are in group4. The 4th, 12th, 20th, and 28th rows contain coefficients from group2. There are several symmetric and asymmetric points in these rows, each point separated by four coefficients.

The multiplier sharing scheme for the flexible input architecture is presented in Table.7. There are two kinds of sharing as in the high throughput architecture, which are intra-sharing (sharing in the time domain) and inter-sharing (sharing in the space domain). However, since small-size transform hardware is completely reused by larger-size transform in the flexible input architecture [9], the multiplier sharing scheme in the new design is actually simpler than in the high throughput architecture. For example, the multiplier no.0 need not be responsible for multiplication of group1 set0 of 8-point, 16-point, and 32-point transform, because these tasks are already taken care of by multiplication of group1 set0 of 4-point transform.

Table. 7 The multiplier sharing scheme for the Flexible Input architecture.

Multiplier No.	coeff group	1 set of 32-point	coeff group	2 sets of 16-point	coeff group	4 sets of 8-point	coeff group	8 sets of 4-point	HW sets
0	-		-		-		group1	set0	set0
1 - 4					group2	set0	-		
5 - 20			group3	set0	-				
21	group4	set0	-				group1	set1	set0/set1
22 - 25					group2	set1	-		
26 - 41			group3	set1	-				
42			-				group1	set2	set0/set2
43 - 46					group2	set2	-		
47					-		group1	set3	set0/set3
48 - 51					group2	set3	-		
52					-		group1	set4	set0/set4
53							group1	set5	set0/set5
54							group1	set6	set0/set6
55							group1	set7	set0/set7
56 - 76							-		set0

The sharing of dedicated multipliers is limited to almost two multiplication tasks to retain high performance design. Basically, the multipliers for multiplication tasks of group4 set0 of the 32-point transform are reused by multiplication tasks of set1-set7. So, the multiplication tasks of set1-set7 are represented as dash boxes in Fig.24, to represent the fact that they are not real multipliers.



$i = 0,1,\dots,31$

$A = mod(i,4),4p,set\lfloor i/4 \rfloor$

$a = 3 + \lfloor i/8 \rfloor$ $\quad B = mod(i,8),8p,set\lfloor i/8 \rfloor$

$b = 1 + \lfloor i/16 \rfloor$ $\quad C = mod(i,16),16p,set\lfloor i/16 \rfloor$

$c = 0$ $\quad D = i,32p,set0$

Fig. 25 MUX output logic.

Fig.25 shows MUX output logic of the flexible input architecture. The $Y_i; i = 0,1,\ldots,31$ are outputs of the architecture. The value of indices A, B, C, D of Y_A, Y_B, Y_C, Y_D can be derived by the equations on lower part of Fig.25. For example, if i is 0, Y_A, Y_B, Y_C, Y_D are $Y_{0,4p,set0}, Y_{0,8p,set0}, Y_{0,16p,set0}$, and $Y_{0,32p,set0}$ respectively. Three of seven *config* bits are selected to be inputs of the MUX output logic. The indices selected are indicated by $a, b, c,$ which the exact value calculated by the equations in the lower part of Fig.25. For example, if i is equal 0, *config*[a], *config*[b], *config*[c] are *config*[3], *config*[1], and *config*[0] respectively.

The MUX output logic shows another benefit of using the configuration encoding scheme, instead of direct encoding using minimum number of five bits. The logic is composed of three inputs instead of five inputs in case of the direct encoding. When consider the targeted Spartan-3A FPGAs which have 4-input LUTs embedded inside, three-input logic and five-input logic will be synthesized into one and two level of logic respectively, which give very different performance in term of speed.

It should be noticed that *config*[c], *config*[b], and *config*[a] together is a branch in the configuration encoding scheme tree. For example, in case of Y_7, *config*[c], *config*[b], *config*[a] are set as *config*[0], *config*[1], and *config*[3] respectively. This sequence of *config* bits corresponds to the uppermost branch of the configuration encoding scheme tree.

4.2 Design Results

The flexible input architecture is designed for the Spartan-3A FPGAs. The design can also easily be migrated to other Xilinx FPGAs with 77 or more number of DSP slices, because their DSP slice structures are similar. The architecture is described in VHDL language and synthesized using Xilinx ISE 14.7. We simulate the architecture using ISim simulator, which comes with the Xilinx ISE, and compare the transform results with the results gotten from the reference software. The compared results ensure correctness of the flexible input architecture.

The flexible input architecture is an improved version of the high throughput architecture presented in chapter3. Design results of the flexible architecture compared with the proposed architecture in chapter3 are summarized in Table.8. The improved architecture can compute transform of much more flexible input combinations.

Table. 8 Design summary, compared with the High Throughput architecture.

	High Throughput	Flexible Input
Technology	Spartan-3A	Spartan-3A
4-input LUTs	15,521/33,280 (46%)	15,677/33,280 (47%)
Slice flip flops	20,818/33,280 (62%)	22,767/33,280 (68%)
DSP slices	77/84 (92%)	77/84 (92%)
Max Frequency (MHz)	211.5	205
Throughput (Msamples/s)	1,692	1,640
Supporting Video Format	7680x4320 @30 fps	7680x4320 @30fps

Results in Table.8 indicate that there is a little penalty in performance aspect of the design, but the architecture still has high enough throughput to support encoding of 8K (7680x4320) at 30 frame per second, which is the same format as the supported format of the high throughput architecture. The new design use 1% more LUTs and 6% more flip flops, which are not significantly increased. The number of required DSP slices, which is usually expensive resources, remains the same as the previous architecture.

The flexible input architecture receives 32 inputs in each turn or every four clock cycles. The inputs can be composed of any possible input combinations. For example, an input combination can be composed of 4 set of 8 inputs, which is represented as (8, 8, 8, 8). Other possible input combinations resulting from a residual quad-tree partitioning are such as (8, 4, 4, 16), (16, 16) etc.

The architecture produces transform of inputs with a high throughput of 1,640 Msamples/s independent of input combinations. The throughput can be derived by the maximum frequency of the design. The maximum frequency which the flexible input architecture can achieve is 205 MHz. Combine the maximum frequency with the fact that the architecture produces new 32 outputs at every turn or every four clock cycles, we come to the conclusion that the throughput of the flexible input architecture is 1,640 Msamples/s. It should be noted that the maximum frequency of 205 MHz is attainable through the use of Digital Clock Manager (DCMs) on the Spartan-3A family, which can generate frequency up to 320 MHz [16].

Comparisons with other designs in literature are summarized in Table.9. Most of the designs in literature cannot support transform computation of flexible input combinations, except the work in [19]. Moreover, the design in [19] produces varying throughput depending on the current input combination and it is designed for inverse transform.

Table. 9 Comparison with other works.

Design	Technology	Function	Flexible	Throughput (Msamples/s)	Supporting Format
Jeske et al. [6]	Cyclone II (90 nm)	1-D 16x16	No	376.2	3840x2160 @30 fps
Zhao et al. [7]	Cyclone IV (60 nm)	1-D all sizes	some combination	238.13	N/A
Park et al. [8]	ASIC 150 nm	1-D all sizes	some combination	1,504	7680x4320 @30 fps
Meher et al. [9]	ASIC 90 nm	1-D all sizes	some combination	2,990	7680x4320 @60 fps
Chiang et al. [19]	ASIC 90 nm	2-D all sizes	some combination	375	3840x2160 @30 fps
proposed	Spartan-3A (90 nm)	1-D all sizes	all combination	1,640	7680x4320 @30 fps

The design in [19] does not report their throughput, but the throughput can be inferred from their clock speed and the number of clock cycles required to do the computation. The resulting throughput is approximately 375 Msamples/s, which is enough to support encoding of 3840x2160 videos at 30 frames per second. The flexible input architecture has higher throughput than [19] and is able to support encoding of 7680x4320 videos at 30 frame per second. Besides that, the design in [19] requires complex scheduling to fill the use of transform unit in order to maintain its high throughput.

Concrete comparisons with other works in literature are difficult, because other architectures have less flexibility, in term of input combinations, than the flexible input architecture and also targeting many different technologies. For example, resource usages in Application Specific Integrated Circuits (ASICs) design are usually reported in term of equivalent number of gates, while FPGAs tools report resource usages in term of basic elements inside the FPGAs. The basic elements of the FPGAs also differ from vendor to vendor, and from family to family.

However, some aspect of the designs such as their performance can be roughly compared. Besides the work in [9], the flexible input architecture provides higher throughput than all other designs in literature. The work in [9] is targeted to be implemented on ASICs, which typically have better performance in term of speed than FPGAs.

The flexible input architecture has high throughput because of two reasons. The first reason is because dedicated multipliers are employed in the design [14]. Other reason is that configuration encoding scheme for representing input combinations is carefully invented.

4.3 Pseudo Code of the Flexible Input Architecture

Fig.26-28 in this section present the pseudo code of the flexible input architecture. Further details about the multiplier sharing scheme of this architecture can be found in Appendix C.

```
module partialButterfly_flex

  input (reset, clock, enable, input_combination, input)
  output (done, output)

  ---- CONTROL PATH
  process #1 : state machine
    case state of
      stIdle  : -- Idle state, wait for enable signal
        if enable is inserted then
          state ← stProcess
        end if
      stProcess : -- transform computing state
        if latency_count_up equals 22 then
          state ← stDone
        end if
      stDone : -- done output is high in this state
        if latency_count_down equals 22 then
          state ← stIdle
        end if
    end case
  end process

  process #2 : generate done signal
    if state equals stDone then
      done ← 1
    end if
  end process

  process #3 : counting up latency
    if state equals stIdle and enable is not inserted, or state equals stDone then
      latency_count_up ← 0
    else
      latency_count_up ← latency_count_up + 1
    end if
  end process

  process #4 : counting down latency
    if state equals stIdle then
      latency_count_down ← 0
    else if enable is not inserted then
      latency_count_down ← latency_count_down + 1
    end if
  end process

  process #5 : counter for controlling sharing of dedicated multipliers and synchronizing the data path
    if state equals stIdle then
      mux_counter ← 0
    else
      mux_counter ← mux_counter + 1
    end if
  end process

  ---- DATA PATH
  process #6 : latching input
    if enable is inserted then
      for i ← 0 to 31 do
        $x_i$ ← $input_i$
      end for
    end if
  end process

  -- 1st step : generate odd even components
  process #7 : generate E components of a 32-point transform
    for i ← 0 to 15 do
      $E\_32p_i$ ← $x_i + x_{31-i}$
    end for
  end process
```

Fig. 26 Pseudo code of the Flexible Input architecture.

```

process #8 : generate O components of a 32-point transform
  for i ← 0 to 15 do
    $O\_32p_i \leftarrow x_i - x_{31-i}$
  end for
end process

process #9 : input multiplexers before 16-point transforms
  for i ← 0 to 15 do
    if $input\_combination_0$ equals 0 then
      $in\_16p_{0i} \leftarrow E\_32p_i$
    else
      $in\_16p_{0i} \leftarrow x_i$
    end if
    $in\_16p_{1i} \leftarrow x_{16+i}$
  end for
end process

process #10 : generate E components of two 16-point transforms
  for s ← 0 to 1 do
    for i ← 0 to 7 do
      $E\_16p_{si} \leftarrow in\_16p_i + in\_16p_{15-i}$
    end for
  end for
end process

process #11 : generate O components of two 16-point transform
  for s ← 0 to 1 do
    for i ← 0 to 7 do
      $O\_16p_{si} \leftarrow in\_16p_i - in\_16p_{15-i}$
    end for
  end for
end process

process #12 : input multiplexers before 8-point transforms
  for i ← 0 to 7 do
    if  $input\_combination_1$ equals 0 then
      $in\_8p_{0i} \leftarrow E\_16p_{0i}$
    else
      $in\_8p_{0i} \leftarrow x_i$
    end if

    if $input\_combination_2$ equals 0 then
      $in\_8p_{1i} \leftarrow E\_16p_{1i}$
    else
      $in\_8p_{1i} \leftarrow x_{8+i}$
    end if

    $in\_8p_{2i} \leftarrow x_{16+i}$
    $in\_8p_{3i} \leftarrow x_{24+i}$
  end for
end process

similar processes for the 1st step : generate odd even components step

-- 2nd step : multiplication
process #13 : multiplexers for selecting multiplicands for dedicated multipliers
  for k ← 0 to 76 do
    case input_combination, mux_counter of
      $mul\_data_k$ ← an appropriate component according to the multiplier sharing scheme
    end case
  end for
end process

process #14 : multiplexers for selecting multipliers for dedicated multipliers
  for k ← 0 to 76 do
    case input_combination, mux_counter of
      $mul\_coeff_k$ ← an appropriate component according to the multiplier sharing scheme
    end case
```

Fig. 27 Pseudo code of the Flexible Input architecture (continue).

```
    end for
  end process

  process #15 : initiate dedicate multipliers
  component k ← 0 to 76 port map (
    multiplier     ← $mul\_data_k$,
    multiplicand   ← $mul\_coeff_k$,
    product   ← $mul\_output_k$
  )
  end process

  process #16 : collecting multiplied terms
    for k ← 0 to 76 do
      case mux_counter of
        collect outputs from $mul\_output_k$ to an appropriate multiplied term register
      end case
    end for
  end process

  process #17 : computing pseudo multiplication
    if mux_counter equals 3 then
      for all element of pseudo-multiplied term registers
        compute pseudo-multiplied terms by binary shifting
      end for
    end if
  end process

  -- 3rd step : post addition subtraction
  process #18 : post additions and subtractions
    if mux_counter equals 0 then
      for all s in possible set number do
        for i ← 0 to 31 do
          $y_{si}$ ← an aggregation tree of multiplied terms according to the transform matrix
        end for
      end for
    end if
  end process

  -- 4th step : rounding and output multiplexer
  process #19 : output multiplexing and rounding
    for i ← 0 to 31 do
      if $input\_partition_0$ equals 0 then
        $output_i \leftarrow (y_{0i}+8) >> 4$
      else if $input\_partition_{1+\lfloor i/16 \rfloor}$ equals 0 then
        $output_i \leftarrow (y_{\lfloor i/16 \rfloor,\ 2i}+4) >> 3$
      else if $input\_partition_{3+\lfloor i/8 \rfloor}$ equals 0 then
        $output_i \leftarrow (y_{\lfloor i/8 \rfloor,\ 4i}+2) >> 2$
      else
        $output_i \leftarrow (y_{\lfloor i/4 \rfloor,\ 8i}+1) >> 1$
      end if
    end for
  end process

end module
```

Fig. 28 Pseudo code of the Flexible Input architecture (continue).

Chapter 5
Experimental Results

After described about all the transform architecture for the HEVC designed on FPGAs, we turn our attention to the experimental results. Since the transform architectures invented in this thesis have no parameter, the only thing needed to be verified is the robustness of the architecture. The robustness of the architectures are checked by comparing their results with the results retrieved from the reference software [5] through a large number of input data from real video sequences.

The outline for this chapter is as follow. First, we discuss about the initial simulation in section 5.1. In this section, functional simulations of the design are shown and timing specifications are explained. In section 5.2, the standard test sequences are introduced. The test sequences used in this thesis are selected from the common test conditions [20]. Then, reference software configuration in our test is presented in section 5.3. First automated testbench is discussed in section 5.4. This section shows that the proposed architectures can correctly compute transform of all sizes without problems such as overflow over a large number of input data sets. Finally, another section, section 5.5, is dedicated to another automated testbench. This second testbench checks that the configuration encoding scheme works correctly.

5.1 Initial Simulation

After completely describing the flexible transform architecture in VHDL, the design is test visually using ISim simulator accompanied with the Xilinx ISE. It should be noted that only the flexible transform architecture is tested, because the high throughput architecture is considered to be a subset of the flexible input transform architecture. The flexible input transform architecture is an improved version of the high throughput architecture.

The data set used for the initial test is retrieved from the reference software and shown in the first column of Table.10. This data set will be called the initial data set. The following columns labeled as 4p results, 8p results, 16p results, and 32p results are the expected results after applying 4-point, 8-point, 16-point, and 32-point transform to the initial data set respectively.

The data in Table.10 can be used for checking the correctness of computing the transform of the initial data set, independent of input combinations selected. For example, if the input combinations selected is (8, 4, 4, 16), then the highlighted data in the 4p results, 8p results, and 16p results columns are used for checking the correctness of the architecture.

An example of the initial simulation is shown in Fig.29. The data set used for this simulation is the initial data set, which first four elements -118, -88, 73, and 127 can be seen from the picture. The *partition* in the timing diagram is the *config* in the design. The *config* is first set to “0000000”, then later changed to “0000001”. This configuration represents input combinations of (32) and (16, 16) respectively. It is worth noting that the *config*, the *partition* in the timing diagram, can be changed as soon as necessary.

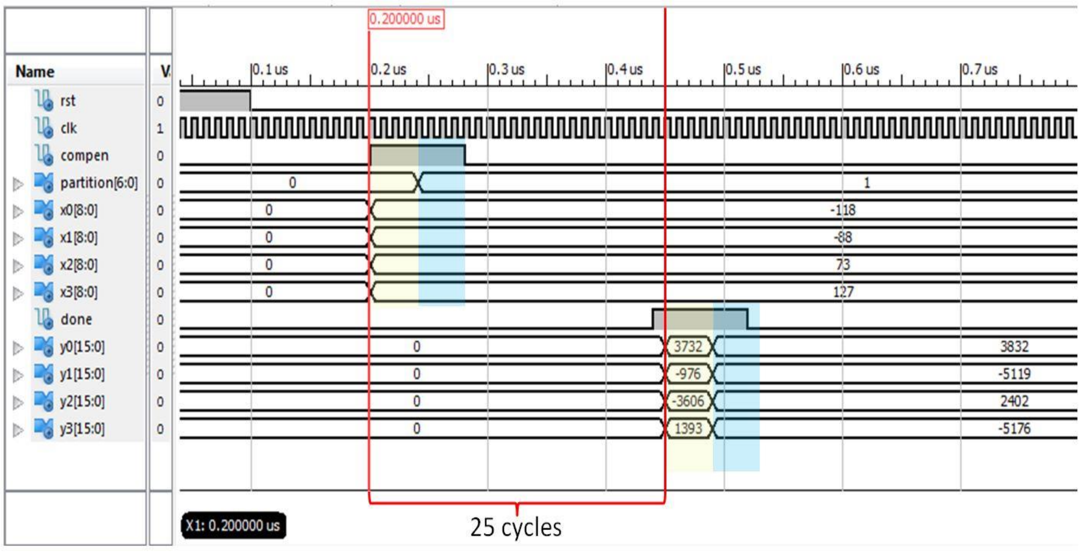


Fig. 29 An example of initial simulation.

Table. 10 The initial data set, retrieved from the HM reference software.

residual input	4p results	8p results	16p results	32p results
-118	-192	672	3832	3732
-88	-13065	-3487	-5119	-976
73	768	-6533	2402	-3606
127	2272	-4782	-5176	1393
12	1536	384	-3288	-1434
12	0	3031	-1654	739
12	0	1136	-3811	118
12	0	-750	-481	-7045
-20	-2144	6992	136	-88
-14	-88	-8292	661	414
-16	-224	-44	2531	-1728
-17	-137	2840	360	-1572
126	16128	-112	534	-2744
126	0	-2030	854	715
126	0	-68	-1181	-1824
126	0	1612	-1	781
-17	-2048	6928	3632	36
-16	-59	-8214	2093	-473
-15	-64	116	-5270	1486
-16	24	2783	-5411	-147
119	15904	-144	3112	1449
126	-290	-1794	1802	1160
126	-224	75	-1676	504
126	-126	1492	3167	-635
127	-192	336	-64	-38
73	13066	2327	-2311	1776
-88	768	6109	368	-224
-118	-2271	6135	-647	-450
11	864	16	-610	-424
13	848	-2530	1301	-20
12	-736	-1295	333	-321
-9	319	826	642	171

The design can be virtually checked by comparing the data in Table.10 with the simulation results. For example, in case of the (32) input combinations, the first four outputs can be get from the 32p column which are 3732, -976, -3606, and 1393 respectively, which is consistent with the yellow highlighted outputs in the timing diagram. Another example can be verified from the blue highlighted inputs and outputs in the diagram.

Timing specifications of the flexible input transform architecture is as followed. The *compen* is the component enable signal. The *partition* is the *config* in the design. Each input needs to hold its value for four clock cycles to wait for multiplier sharing which reusing each multiplier four times. The *compen* and *partition* are synchronous with the inputs and also hold the value for four clock cycles.

The *done* signal indicates that the design is almost ready to give the outputs. This signal is intentionally set a clock cycle prior to the outputs to allow other receiver modules to have time for decisions before the outputs come. Each output stays at the same value for four clock cycles. Finally, the latency between each input and output sets is 25 clock cycles. This latency is mainly resulted from pipelining inside the design.

5.2 Standard Test Sequences

A set of standard test sequences for the HEVC is defined in the common test conditions and software reference configurations [20]. These test sequences are mainly aimed to be used during the development of the standard to test for the trade-off between coding efficiency and reconstruction picture quality of coding tools.

The standard test sequences are classified into class A to class F according to their picture size and applications [11]. This classification is summarized in Table.11.

Class A deals with very high resolution videos, with resolution higher than 1920x1080. The aims of Class A sequences are for evaluating the 4K/8K videos, but to save computation times the video size is reduced to 2560x1600. Class B sequences are used for testing 1080p or 1920x1080 resolution videos. Class C and Class D are for testing mobile applications, which Class C videos have resolution of 832x480 and Class D videos have resolution of 416x240 respectively.

Class E sequences are for a specific low-latency application such as visual communications. The resolution of Class E sequences is 1280x720. The last test sequence class is Class F. The Class F aims to test non-camera video such as videos created by computer graphic.

Table. 11 Test sequence classes summary

Class	Resolution	Applications
A	2560x1600	Ultra high resolutions, 4K, 8K videos
B	1920x1080	Full HD, 1080p, videos
C	832x480	Mobile applications
D	416x240	Mobile applications
E	1280x720	Low-latency applications, e.g. visual communication
F	1280x720	Non-camera videos, computer graphic

All test sequences of the HEVC are listed in Table.12. Three sequences are selected from the standard test sequences to test the transform architecture designed in this thesis. The selected sequences are highlighted in Table.12 and their example frames are shown in Fig.30. The reason for selecting low resolution videos is because the time required for simulating higher resolution videos is too long, as will be further discussed in section 5.4.

Frequency characteristic of videos has effects on the transform step, so we select two video sequences which have different frequency characteristic to test our transform architecture. Objects inside the “BasketballPass” sequence move rather fast, so the sequence contains significantly high frequencies. While, objects inside the

“BQSquare” move relatively slow, so the sequence contains significantly low frequencies.

All of the test sequences are available on ftp://hevc@ftp.tnt.uni-hannover.de/testsequences/, however username and password are required for authentication. Other place to find the test sequences is ftp://ftp.kw.bbc.co.uk/hevc/hm-10.0-anchors/bitstreams/i_main. However, only encoded bitstreams are provided, since raw video data required large space to be store.

5.3 The HEVC Reference Software Configuration

The HEVC reference software, the HM test model, version 13 is used as the reference to check the correctness of our transform architecture [5]. The software encoder profile is set to Main profile. The encoding bit depth is 8 bits.

Since it is out of the thesis scope to thoroughly study the structure of the reference software, it is difficult to let the reference software directly control the *config* of the flexible input architecture. However, the inputs and outputs of each one-dimension HEVC transform step can be retrieved out from the partialButterfly4, partialButterfly8, partialButterfly16, and partialButterfly32 function inside the program. This four functions code is in appendix A.

Only two frames are used in each video sequence, because HEVC encoding and testing are time consuming processes. The time required to encode 2 frames of each testing sequence is summarized in Table.13.

Table. 12 The HEVC standard test sequences [20].

Class	Sequence name	Resolution	Bit depth
A	Traffic	2560x1600	8
A	PeopleOnStreet	2560x1600	8
A	Nebuta	2560x1600	10
A	StreamLocomotive	2560x1600	10
B	Kimono	1920x1080	8
B	ParkScene	1920x1080	8
B	Cactus	1920x1080	8
B	BQTerrace	1920x1080	8
B	BasketballDrive	1920x1080	8
C	RaceHorses	832x480	8
C	BQMall	832x480	8
C	PartyScene	832x480	8
C	BasketballDrill	832x480	8
D	RaceHorses	416x240	8
D	BQSquare	416x240	8
D	BlowingBubbles	416x240	8
D	BasketballPass	416x240	8
E	FourPeople	1280x720	8
E	Johnny	1280x720	8
E	KristenAndSara	1280x720	8
F	BasketballDrillText	1280x720	8
F	ChinaSpeed	1280x720	8
F	SlideEditing	1280x720	8
F	SlideShow	1280x720	8

(a)

(b)

(c)

Fig. 30 Test sequences (a) BasketballDrill (b) BQSquare (c) BasketballPass

Table. 13 Two-frame encoding time of the selected test sequences.

Class	Sequence Name	Resolution	Two-frame Encoding Time (min)
C	BasketballDrill	832x480	88.23
D	BasketballPass	416x240	25.68
D	BQSquare	416x240	34.58

5.4 The First Automated Testbench

The first testbench for the flexible input architecture is a functional automated testbench. This testbench is an exhaustive testing to verify the correctness of the architecture when computing 4-point, 8-point, 16-point, and 32-point HEVC transform respectively.

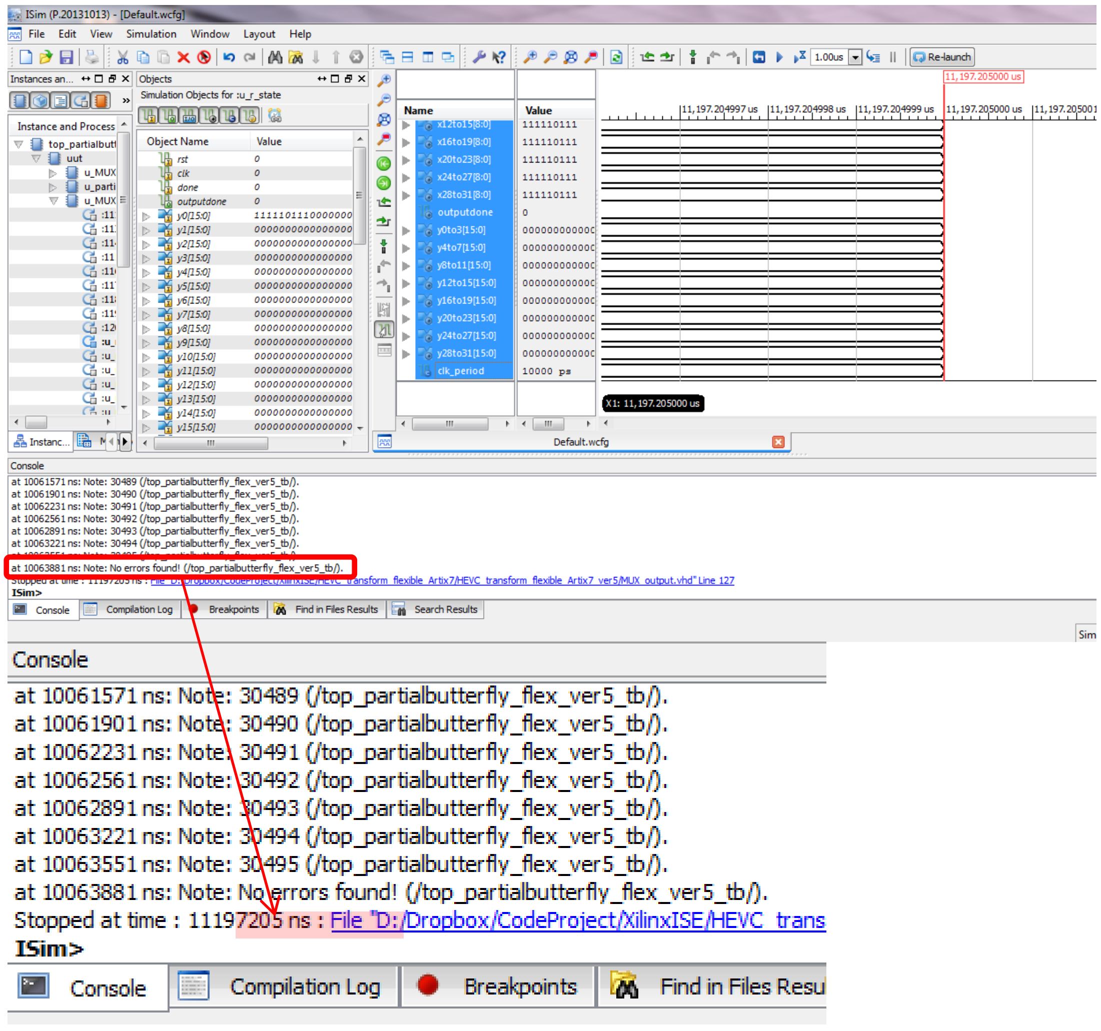
The inputs and outputs data from one-dimension transform steps computing during encoding process of each sequence are written out in text file format. Each text file corresponds to a transform size of a video sequence. For example, partialButterfly_16_BasketballPass.txt stores all 16-point transform inputs and outputs get from the BasketballPass sequence.

An automated testbench is written in VHDL as shown in Fig.31.

```
module partialButterfly_flex_tb_1

   process #1 : initiate the unit under test
      component partialButterfly_flex port map
   end process

   process #2 : generate clock signal
      if simulation time elapses for half clock-period then
         clock ← not clock
      end if
   end process

   process #3 : generate stimuli
      initial reset
      while( not endfile )
         read input data form a file
         insert input data to the unit under test
         read output data from a file
         assert output data form a file equals output data from the architecture
      end while
      print “No Error Found!”
   end process

end module
```

Fig. 31 Pseudo code of the first automated testbench.

Basically the code reads the text file database, computes HEVC transform of the inputs received, and compare the architecture outputs with the results database. The *std.textio* library is used for assisting text files reading. The *readline*, *read*, *assert*, and *report* functions are employed in the testbench.



```
Console
at 10061571 ns: Note: 30489 (/top_partialbutterfly_flex_ver5_tb/).
at 10061901 ns: Note: 30490 (/top_partialbutterfly_flex_ver5_tb/).
at 10062231 ns: Note: 30491 (/top_partialbutterfly_flex_ver5_tb/).
at 10062561 ns: Note: 30492 (/top_partialbutterfly_flex_ver5_tb/).
at 10062891 ns: Note: 30493 (/top_partialbutterfly_flex_ver5_tb/).
at 10063221 ns: Note: 30494 (/top_partialbutterfly_flex_ver5_tb/).
at 10063551 ns: Note: 30495 (/top_partialbutterfly_flex_ver5_tb/).
at 10063881 ns: Note: No errors found! (/top_partialbutterfly_flex_ver5_tb/).
Stopped at time : 11197205 ns : File "D:/Dropbox/CodeProject/XilinxISE/HEVC_transform_flexible_Artix7/HEVC_transform_flexible_Artix7_ver5/MUX_output.vhd" Line 127
ISim>
```

Fig. 32 “No Error” report in ISim simulator.

After a time consuming simulation process in ISim simulator, the simulator console report no error as shown in Fig.32. The total number of transform tested in each case is summarized in Table.14. The total number of 475,176 4-point transforms, 626,008 8-point transforms, 482,544 16-point transforms, and 328,832 32-point transforms are tested, which results in the total number of 1,912,560 transforms error-freed. So we can claim with a great degree of confidence that the designed transform architecture is robust.

Table. 14 The number of transform tested.

	4p	8p	16p	32p	
BasketballDrill	305,200	407,768	316,216	215,232	
BasketballPass	85,844	112,864	90,456	60,352	
BQSquare	84,132	105,376	75,872	53,248	
Total	475,176	626,008	482,544	328,832	1,912,560

The time required for testing each transform size of each test sequence is summarized in Table.15 to show how long it takes to simulate the sequences. Note that the unit of Table.15 is hour.

Table. 15 Time required for automated simulation (hours).

	4p	8p	16p	32p
BasketballDrive	136.91	194.50	174.55	117.49
BasketballDrill	22.17	32.77	26.70	22.13
BasketballPass	6.23	8.78	7.33	5.07
BQSquare	13.43	12.90	7.34	6.32
FourPeople	56.49	77.81	61.80	43.52
SlideEditing	59.40	74.91	56.23	40.52

The test sequences are simulated using two laptops. The first laptop, with Intel Core i5-3230M (2.6 GHz, 3 MB L3 Cache, up to 3.2 GHz) CPU, runs the BasketballDrill and BasketballPass sequences on the Xilinx ISE ISim simulator. The second laptop, with Intel Core i7-4700HQ (2.4 GHz, 6 MB L3 Cache, up to 3.4 GHz) CPU, runs the BQSquare sequence on the Xilinx Vivado simulator.

The first platform can simulate faster, so we use time data from the BasketballDrill and BasketballPass sequences to estimate the time required to simulate other higher resolution videos, shaded in Table.15. The time data are plotted as shown in Fig.33 and linear regression is used for estimation. The estimation results are reported in shaded area of Table.15, which emphasize the reason why higher resolution videos are not simulated.

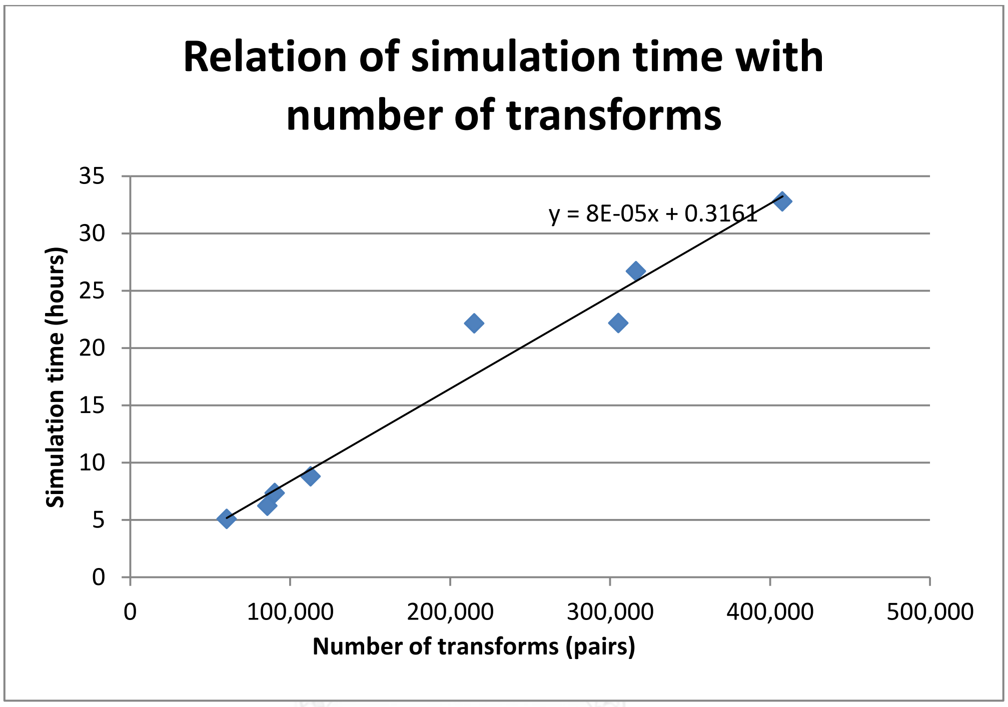


Fig. 33 Relation of simulation time with number of transforms.

5.5 The Second Automated Testbench

The second testbench is also a functional automated testbench written in VHDL. This testbench aims to test correctness of the configuration encoding scheme inside the flexible input transform architecture. Since structure of the HEVC reference software is difficult to know and out of scope of the thesis, we synthesize a test data set to be used in our testing.

First, pairs of inputs and outputs of all transform sizes are collected from the HEVC reference software to create our database, 10,000 pairs are stored for each transform size. The database looks like the Table.16. Data are not fully shown in the table to avoid too much information. For example, I4,0 and O4,0 is a pair of 4-point transform, which might be (-118, -88, 73, 127) and (-192, -13065, 768, 2272) respectively.

Table. 16 Transform input-output pairs database.

pair number	Input 4p	Output 4p	Input 8p	Output 8p	Input 16p	Output 16p	Input 32p	Output 32p
0	I4,0	O4,0						
1			I8,1	O8,1				
2	I4,2	O4,2						
…								
99					I16,99	O16,99		
…								
9999								

A test program is then written in C to generate a test data set as shown in Fig.34.

```
database ← read data from files that are retrieved from the reference software
NUMBER_OF_BLOCK ← 10000
for set ← 0 to NUMBER_OF_BLOCK-1 do
   pattern ← random integer between 0 to 25
   generate 32 input-output data with the selected pattern, using database
   write the pattern and the generated input-output data to an output file
end for
```

Fig. 34 Pseudo code of the test program.

Firstly, the test program randomly selects the test data input combination, which can be any of the 26 possible input combinations resulted from a quad-tree partitioning. Then, data pairs from the database are randomly selected and concatenated to form a pair of 32 inputs and outputs of the specified input combination.

For example, if the test program select input combination of (8, 4, 4, 16), the inputs and outputs pair can be (I8,1, I4,0, I4,2, I16,99) and (O8,1, O4,0, O4,2, O16,99) respectively.

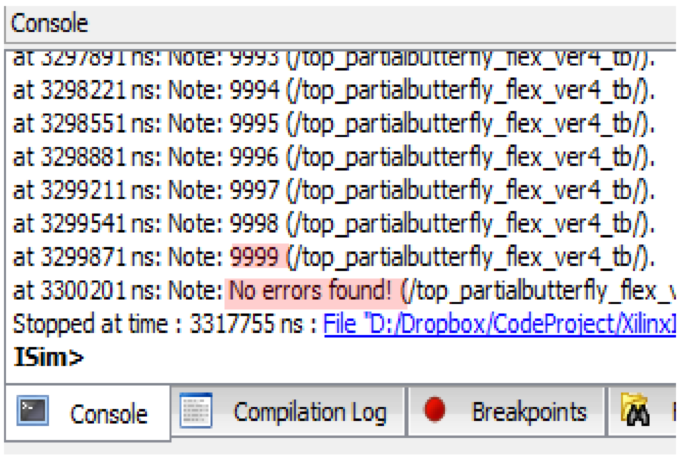
Both input combination and inputs and outputs pair selected are written into a text file. The text file is then read by an automated testbench described by Fig.35.

```
module partialButterfly_flex_tb_2

  process #1 : initiate the unit under test
    component partialButterfly_flex port map
  end process

  process #2 : generate clock signal
    if simulation time elapses for half clock-period then
      clock ← not clock
    end if
  end process

  process #3 : generate stimuli
    initial reset
    while( not endfile )
      read input data from a file
      read input combination from a file
      insert input data and input combination to the unit under test
      read output data from a file
      assert output data form a file equals output data from the architecture
    end while
    print "No Error Found!"
  end process

end module
```

Fig. 35 Pseudo code of the second automated testbench.

This is how the testbench works. The *config* of the design is set by input combination and the corresponding inputs are inserted into the architecture. Outputs of the architecture are compared with the results in the text file. Assertions are used in the simulation to ensure that no data mismatch occurs during the simulation, thus ensure correctness of the design.

Total number of 10,000 blocks of random data is tested. The results are error-freed as indicated by the simulator console shown in Fig.36.



```
at 3297891 ns: Note: 9993 (/top_partialbutterfly_flex_ver4_tb/).
at 3298221 ns: Note: 9994 (/top_partialbutterfly_flex_ver4_tb/).
at 3298551 ns: Note: 9995 (/top_partialbutterfly_flex_ver4_tb/).
at 3298881 ns: Note: 9996 (/top_partialbutterfly_flex_ver4_tb/).
at 3299211 ns: Note: 9997 (/top_partialbutterfly_flex_ver4_tb/).
at 3299541 ns: Note: 9998 (/top_partialbutterfly_flex_ver4_tb/).
at 3299871 ns: Note: 9999 (/top_partialbutterfly_flex_ver4_tb/).
at 3300201 ns: Note: No errors found! (/top_partialbutterfly_flex_
Stopped at time : 3317755 ns : File "D:/Dropbox/CodeProject/Xilinx
ISim>
```

Fig. 36 "No Error" report in ISim simulator.

Chapter 6
Conclusion and Future Works

This thesis contributes two architectures for the transform step of the latest video coding standard, the High Efficiency Video Coding (HEVC). These architectures are aimed to be employed as part of an HEVC real-time encoder on FPGA platforms. The design possesses high throughput feature and can receive flexible input combination.

The first architecture, the high throughput architecture, has a high enough throughput to support encoding of 4K resolution videos at 30 fps in real-time on Spartan-3A FPGAs. It has possibility to support up to 8K resolution videos with better technology. The dedicated multipliers inside the DSP slices, which are dedicated resources inside FPGAs, are extensively used in this design to gain high throughput [14]. Using dedicated resources in critical tasks is an important strategy in FPGA designs.

The second architecture, the flexible input architecture, can compute transform of any input combinations resulting from a residual quad-tree partitioning, which is the partitioning used for getting basic processing units of the transform step. Configuration encoding scheme is invented to represent possible input combinations. By using this configuration encoding scheme, the flexible input transform architecture can be constructed without any increase in the number of dedicated multipliers required [18].

The transform architecture are verified by checking with the HEVC reference software [5]. Automated testbench is written in VHDL to read transform database from text files and compare transform results to ensure robustness of the design.

Possible future works include designing other modules of the HEVC encoder, such as the intra prediction step, the inter prediction step, or the entropy encoder. There are several parameters to compromise the trade-off between the reconstructed picture quality and the coding efficiency.

Another possible future work is to implement an architecture for the HEVC inverse transform. Since inverse transform matrices of the HEVC are transpose of the forward transform matrices, these two categories of matrices have similar structure and symmetries. It is possible to build the inverse transform architecture from the forward transform architecture with minor modification. Section 6.1 gives an idea about this modification.

6.1 Flexible and High Throughput Inverse Transform Architecture idea

This section presents an idea to build inverse transform architecture for the HEVC from the forward transform architecture. Firstly, we will explore the similarities between computation of the 4-point forward and inverse transform. Then, we will show sharing of hardware between different transform sizes in the inverse transform. This idea of sharing is similar to the sharing in the forward transform, described earlier in chapter.4. Finally, we will describe a possible benefit we can get from using flexible inverse transform architecture.

To show similarities between the computation of 4-point forward and inverse transform, only the first three steps of the partial butterfly algorithm, excluding the rounding step, will be shown.

First, consider the first three steps of the 4-point forward transform, which are used to generate the intermediate results Z_i before rounding step,

$$\begin{bmatrix} Z_0 \\ Z_1 \\ Z_2 \\ Z_3 \end{bmatrix} = \begin{bmatrix} 64 & 64 & 64 & 64 \\ 83 & 36 & -36 & -83 \\ 64 & -64 & -64 & 64 \\ 36 & -83 & 83 & -36 \end{bmatrix} \bullet \begin{bmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \end{bmatrix}, \qquad (17)$$

where $X_i; i = 0,1,2,3$ are inputs of the transform and $Z_i; i = 0,1,2,3$ are intermediate results before rounding step. This matrix multiplication can be decomposed into steps according to the partial butterfly algorithm as in the following sets of equations [5],

$$\begin{aligned} E_0 &= X_0 + X_3, \\ E_1 &= X_1 + X_2, \\ O_0 &= X_0 - X_3, \\ O_1 &= X_1 - X_2, \end{aligned} \tag{18}$$

$$\begin{aligned} Z_0 &= 64 \cdot E_0 + 64 \cdot E_1, \\ Z_2 &= 64 \cdot E_0 - 64 \cdot E_1, \\ Z_1 &= 83 \cdot O_0 + 36 \cdot O_1, \\ Z_3 &= 36 \cdot O_0 - 83 \cdot O_1. \end{aligned} \tag{19}$$

Equations (19) can be written in matrix form as,

$$\begin{aligned} \begin{bmatrix} Z_0 \\ Z_2 \end{bmatrix} &= \begin{bmatrix} 64 & 64 \\ 64 & -64 \end{bmatrix} \bullet \begin{bmatrix} E_0 \\ E_1 \end{bmatrix}, \\ \begin{bmatrix} Z_1 \\ Z_3 \end{bmatrix} &= \begin{bmatrix} 83 & 36 \\ 36 & -83 \end{bmatrix} \bullet \begin{bmatrix} O_0 \\ O_1 \end{bmatrix}. \end{aligned} \tag{20}$$

Next, consider the first three steps of the 4-point inverse transform, which will be used to generate the intermediate results F_i before rounding step,

$$\begin{bmatrix} F_0 \\ F_1 \\ F_2 \\ F_3 \end{bmatrix} = \begin{bmatrix} 64 & 83 & 64 & 36 \\ 64 & 36 & -64 & -83 \\ 64 & -36 & -64 & 83 \\ 64 & -83 & 64 & -36 \end{bmatrix} \bullet \begin{bmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \end{bmatrix}, \tag{21}$$

where $X_i; i = 0,1,2,3$ are inputs to the inverse transform step and $F_i; i = 0,1,2,3$ are intermediate results before rounding step. It should be noted that the matrix in (21) is a transpose of the matrix in (17). The matrix multiplication in (21) can be decomposed into steps according to the partial butterfly algorithm for inverse transform as in the following sets of equations [5],

$$\begin{aligned} Z_0 &= 64 \cdot X_0 + 64 \cdot X_2, \\ Z_2 &= 64 \cdot X_0 - 64 \cdot X_2, \\ Z_1 &= 83 \cdot X_1 + 36 \cdot X_3, \\ Z_3 &= 36 \cdot X_1 - 83 \cdot X_3, \end{aligned} \tag{22}$$

$$\begin{aligned} F_0 &= Z_0 + Z_1, \\ F_1 &= Z_2 + Z_3, \\ F_2 &= Z_2 - Z_3, \\ F_3 &= Z_0 - Z_1. \end{aligned} \tag{23}$$

Equations (22) can be written in matrix form as,

$$\begin{aligned} \begin{bmatrix} Z_0 \\ Z_2 \end{bmatrix} &= \begin{bmatrix} 64 & 64 \\ 64 & -64 \end{bmatrix} \bullet \begin{bmatrix} X_0 \\ X_2 \end{bmatrix}, \\ \begin{bmatrix} Z_1 \\ Z_3 \end{bmatrix} &= \begin{bmatrix} 83 & 36 \\ 36 & -83 \end{bmatrix} \bullet \begin{bmatrix} X_1 \\ X_3 \end{bmatrix}. \end{aligned} \tag{24}$$

By comparing (20) and (24), it is obvious that the matrices in these two equations are the same matrices. So, the same hardware used for the second and the third step of the partial butterfly for forward transform can also be used in the inverse transform. Both (18) and (23) contain the same number of additions and subtractions, two additions and two subtractions.

In conclusion, the modification needed to be carried out to construct the inverse transform architecture is to move the hardware for the first step of the partial butterfly algorithm for forward transform to be placed after the third step. This modification is illustrated by Fig.37. The upper-part of Fig.37 is the original hardware for a 4-point forward transform, while the lower-part is the hardware for a 4-point inverse transform.

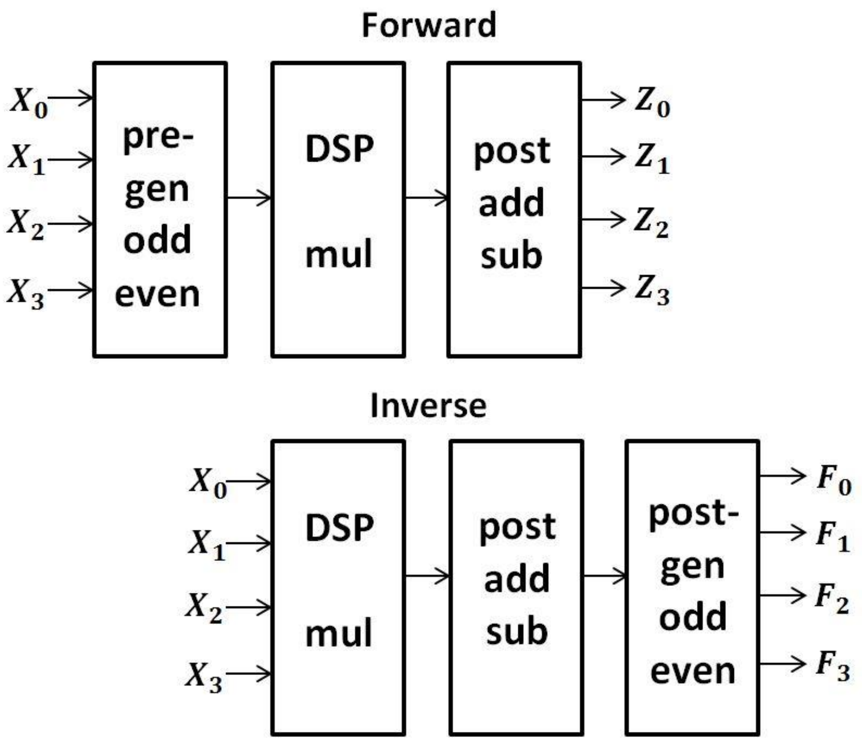
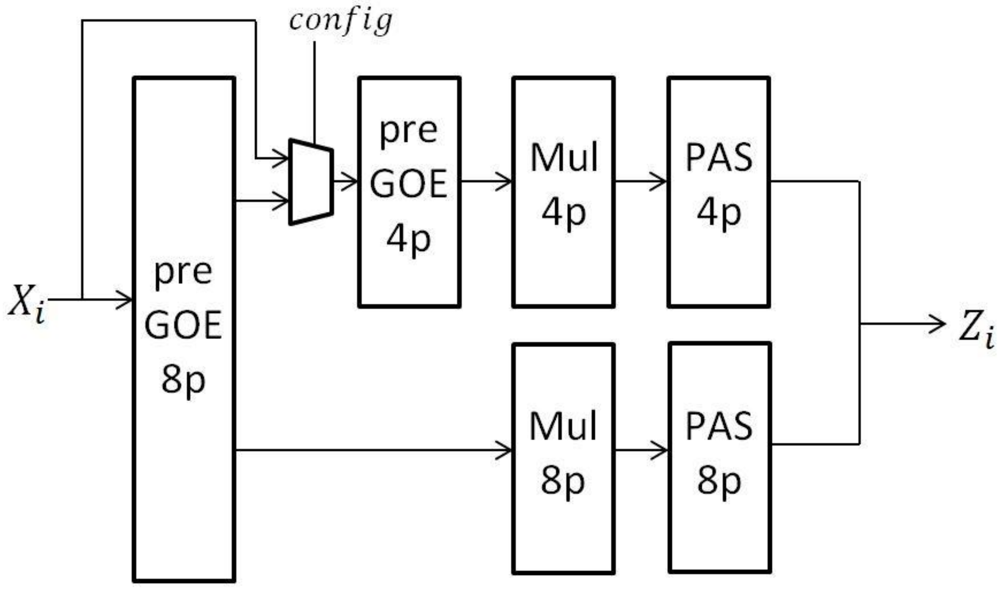


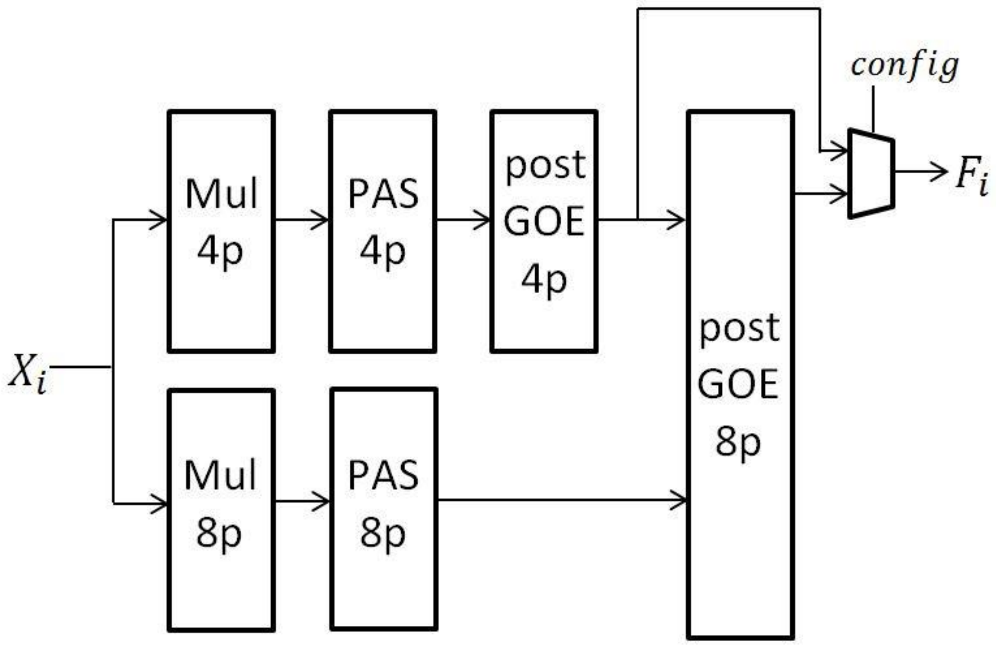
Fig. 37 Hardware for the first three steps of partial butterfly algorithm (for forward and inverse transform).

Next, sharing of hardware between different transform sizes in the inverse transforms will be explained. The sharing structures are basic building blocks for the flexible input architecture, for both forward and inverse transform, so a picture of sharing structure between 4-point and 8-point transform are illustrated in Fig.38. Fig.38 (a) is for the forward transform and Fig.38 (b) is for the inverse transform.

It should be noted that the sharing structure of the forward transform is already seen in chapter.4. Fig.22 and Fig.38 (a) are same structure. Fig.22 shows more details in the data path, but it does not show the input multiplexer. The structure in Fig.22 and Fig.38 (a) is employed in building the flexible input architecture shown in Fig.24.



(a) Forward transform



(b) Inverse transform

Fig. 38 Hardware sharing between 4-point and 8-point transform
(a) forward transform (b) inverse transform.

After showing the hardware sharing structure in the inverse transform, we now analyzes inverse transform equations to make it clear why different transform sizes hardware can be shared.

The first three steps of an 8-point inverse transform can be represented by the following matrix multiplication,

$$\begin{bmatrix} F_0 \\ F_1 \\ F_2 \\ F_3 \\ F_4 \\ F_5 \\ F_6 \\ F_7 \end{bmatrix} = \begin{bmatrix} 64 & 89 & 83 & 75 & 64 & 50 & 36 & 18 \\ 64 & 75 & 36 & -18 & -64 & -89 & -83 & -50 \\ 64 & 50 & -36 & -89 & -64 & 18 & 83 & 75 \\ 64 & 18 & -83 & -50 & 64 & 75 & -36 & -89 \\ 64 & -18 & -83 & 50 & 64 & -75 & -36 & 89 \\ 64 & -50 & -36 & 89 & -64 & -18 & 83 & -75 \\ 64 & -75 & 36 & 18 & -64 & 89 & -83 & 50 \\ 64 & -89 & 83 & -75 & 64 & -50 & 36 & -18 \end{bmatrix} \bullet \begin{bmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \\ X_4 \\ X_5 \\ X_6 \\ X_7 \end{bmatrix}, \quad (25)$$

where $X_i; i = 0,1, \dots ,7$ are inputs to the inverse transform and $F_i; i = 0,1, \dots ,7$ are intermediate results before rounding step.

Matrix multiplication in (25) can be decomposed into steps as shown in the following equations [5],

$$\begin{bmatrix} Z_0 \\ Z_2 \\ Z_4 \\ Z_6 \end{bmatrix} = \begin{bmatrix} 64 & 83 & 64 & 36 \\ 64 & 36 & -64 & -83 \\ 64 & -36 & -64 & 83 \\ 64 & -83 & 64 & -36 \end{bmatrix} \bullet \begin{bmatrix} X_0 \\ X_2 \\ X_4 \\ X_6 \end{bmatrix}, \quad (26)$$

$$\begin{aligned} Z_1 &= 89 \cdot X_1 + 75 \cdot X_3 + 50 \cdot X_5 + 18 \cdot X_7, \\ Z_3 &= 75 \cdot X_1 - 18 \cdot X_3 - 89 \cdot X_5 - 50 \cdot X_7, \\ Z_5 &= 50 \cdot X_1 - 89 \cdot X_3 + 18 \cdot X_5 + 75 \cdot X_7, \\ Z_7 &= 18 \cdot X_1 - 50 \cdot X_3 + 75 \cdot X_5 - 89 \cdot X_7, \end{aligned} \quad (27)$$

$$\begin{aligned}
F_0 &= Z_0 + Z_1, \\
F_1 &= Z_2 + Z_3, \\
F_2 &= Z_4 + Z_5, \\
F_3 &= Z_6 + Z_7, \\
F_4 &= Z_6 - Z_7, \\
F_5 &= Z_4 - Z_5, \\
F_6 &= Z_2 - Z_3, \\
F_7 &= Z_0 - Z_1.
\end{aligned} \tag{28}$$

The matrix in (26) is the same as the matrix in (21), so computation in (26) can be taken care of by a 4-point transform. The multiplexer in Fig.38 (b) selects whether output of the 4-point transform hardware will be directly sent out as 4-point inverse transform outputs or further processed by the 8-point transform hardware.

As final note, the flexible input inverse transform architecture can have another benefit because data in HEVC bitstream can be extracted and used to directly control the circuit.

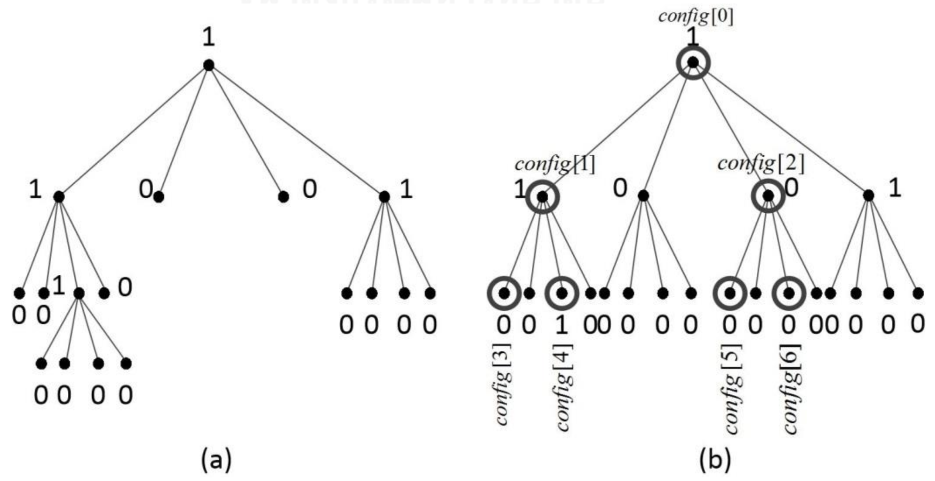


Fig. 39 (a) the quad-tree structure of partitioning in Fig.23 (b) the modified quad-tree structure of partitioning in Fig.23.

According to the standard specification of HEVC [21], partitioning of a coding unit into transform units is represented by a quad-tree structure. For example, quad-tree structure that represents the partitioning in the right-hand side of Fig.23 is shown in Fig.39 (a). This quad-tree structure can be directly decoded from the HEVC binary stream. Further details about syntax for representing the quad-tree can be found in [21] and [22].

The configuration encoding scheme in the flexible input transform architecture can be easily extracted directly from the quad-tree structure with modest modification. The modified version of the quad-tree structure in Fig.39 (a) is shown in Fig.39 (b). Basically, we normalized every branches of the tree to have equal level of three levels from the root. This is possible because a zero node imply no more partition and there cannot be partitioning in the fourth layer since smallest size of transform unit, 4x4, is already reached.

In Fig.39 (b), the nodes from the modified quad-tree structure that are extracted to form the configuration encoding scheme in the left-hand side of Fig.23 are circled. The simple mapping from a quad-tree structure to configuration encoding scheme is beneficial, because header information in HEVC stream can be directly used to control our circuit.

REFERENCES

[1] G. J. Sullivan, J. Ohm, H. Woo-Jin, and T. Wiegand, "Overview of the High Efficiency Video Coding (HEVC) Standard," *Circuits and Systems for Video Technology, IEEE Transactions on,* vol. 22, pp. 1649-1668, Dec. 2012.

[2] F. Bossen, B. Bross, K. Suhring, and D. Flynn, "HEVC Complexity and Implementation Analysis," *Circuits and Systems for Video Technology, IEEE Transactions on,* vol. 22, pp. 1685-1696, Dec. 2012.

[3] K. Ian and J. Rose, "Measuring the Gap Between FPGAs and ASICs," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on,* vol. 26, pp. 203-215, Feb 2007.

[4] K. Kihyun and R. Kwangki, "High performance hardware architecture for multi-mode 1-D forward transform of HEVC," in *Consumer Electronics (GCCE), 2013 IEEE 2nd Global Conference on*, Oct. 2013, pp. 343-344.

[5] JCT-VC, "HM Software," 13 ed.

[6] R. Jeske, J. C. Souza, G. Wrege, R. Cenceicao, M. Grellert, J. Mattos*, et al.*, "Low Cost and High Throughput Multiplierless Design of a 16p 1-D DCT of the New HEVC Video Coding Standard," presented at the VIII Southern Conference on Programmable Logic (SPL), Mar. 2012.

[7] W. Zhao, T. Onoye, and S. Tian, "High-performance multiplierless transform architecture for HEVC," in *Circuits and Systems (ISCAS), 2013 IEEE International Symposium on*, May. 2013, pp. 1668-1671.

[8] S. Y. Park and P. K. Meher, "Flexible integer DCT architectures for HEVC," in *Circuits and Systems (ISCAS), 2013 IEEE International Symposium on*, May. 2013, pp. 1376-1379.

[9] P. K. Meher, P. Sang Yoon, B. K. Mohanty, L. Khoon Seong, and Y. Chuohao, "Efficient Integer DCT Architectures for HEVC," *Circuits and Systems for Video Technology, IEEE Transactions on,* vol. 24, pp. 168-178, Jan. 2014.

[10] A. Fuldseth, G. Bjontegaard, M. Budagavi, and V. Sze, "CE10: Core transform design for HEVC (JCTVC-G495)," Nov. 2011.

[11] V. Sze, M. Budagavi, and G. J. Sullivan, *High Efficiency Video Coding (HEVC) Algorithms and Architectures*: Springer, 2014.

[12] I. E. Richardson, *The H.264 Advanced Video Compression Standard*, 2 ed. Chichester: John Wiley & Sons, 2010.

[13] G. Bjontegaard, "Calculation of average PSNR differences between RD-curves (VCEG-M33)," Mar. 2001.

[14] P. Arayacheeppreecha, S. Pumrin, and B. Supmonchai, "1-D integer transform for HEVC encoder using DSP slices on FPGA," presented at the IEEE International Electircal Engineering Congress (iEECON), 2015.

[15] X. Inc., "XtremeDSP DSP48A for Spartan-3A DSP FPGAs User Guide (UG431)," Jul. 2008.

[16] X. Inc., "Extended Spartan-3A Family Overview (DS706)," Feb. 2011.

[17] A. Corporation, "Cyclone II Device Handbook, Volume 1 (CII5V1-3.3)."

[18] P. Arayacheeppreecha, S. Pumrin, and B. Supmonchai, "Flexible Input Transform Architecture for HEVC Encoder on FPGA," presented at the International Conferences on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON), 2015.

[19] C. Pai-Tse and C. Tian Sheuan, "A reconfigurable inverse transform architecture design for HEVC decoder," in *Circuits and Systems (ISCAS), 2013 IEEE International Symposium on*, May. 2013, pp. 1006-1009.

[20] F. Bossen, "Common test conditions and software reference configurations (JCTVC-I1100)," May. 2012.

[21] ITU., "Recommendation ITU-T H.265," ed, Apr. 2013.

[22] I. K. Kim, J. Min, W. J. Han, and J. H. Park, "Block Partitioning Structure in the HEVC Standard," *IEEE Trans Circuits Syst. Video Technol.,* vol. 22, pp. 1697-1706, Dec. 2012.

APPENDIX

APPENDIX A

Transform Functions inside the Reference Software

```
const Short g_aiT4[4][4] =
{
  {64, 64, 64, 64},
  {83, 36,-36,-83},
  {64,-64,-64, 64},
  {36,-83, 83,-36}
};

const Short g_aiT8[8][8] =
{
  {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}
};


const Short g_aiT16[16][16] =
{
  {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}
};


const Short g_aiT32[32][32] =
{
  {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}
};


void partialButterfly4(Short *src,Short *dst,Int shift, Int line)
{
  Int j;
  Int E[2],O[2];
  Int add = 1<<(shift-1);

  for (j=0; j<line; j++)
  {
    /* E and O */
    E[0] = src[0] + src[3];
    O[0] = src[0] - src[3];
    E[1] = src[1] + src[2];
    O[1] = src[1] - src[2];

    dst[0] = (g_aiT4[0][0]*E[0] + g_aiT4[0][1]*E[1] + add)>>shift;
    dst[2*line] = (g_aiT4[2][0]*E[0] + g_aiT4[2][1]*E[1] + add)>>shift;
    dst[line] = (g_aiT4[1][0]*O[0] + g_aiT4[1][1]*O[1] + add)>>shift;
    dst[3*line] = (g_aiT4[3][0]*O[0] + g_aiT4[3][1]*O[1] + add)>>shift;

    src += 4;
    dst ++;
  }
}

void partialButterfly8(Short *src,Short *dst,Int shift, Int line)
{
  Int j,k;
  Int E[4],O[4];
  Int EE[2],EO[2];
  Int add = 1<<(shift-1);

  for (j=0; j<line; j++)
  {
    /* E and O*/
    for (k=0;k<4;k++)
    {
      E[k] = src[k] + src[7-k];
      O[k] = src[k] - src[7-k];
    }
    /* EE and EO */
    EE[0] = E[0] + E[3];
    EO[0] = E[0] - E[3];
    EE[1] = E[1] + E[2];
    EO[1] = E[1] - E[2];

    dst[0] = (g_aiT8[0][0]*EE[0] + g_aiT8[0][1]*EE[1] + add)>>shift;
    dst[4*line] = (g_aiT8[4][0]*EE[0] + g_aiT8[4][1]*EE[1] + add)>>shift;
    dst[2*line] = (g_aiT8[2][0]*EO[0] + g_aiT8[2][1]*EO[1] + add)>>shift;
    dst[6*line] = (g_aiT8[6][0]*EO[0] + g_aiT8[6][1]*EO[1] + add)>>shift;
    dst[line] = (g_aiT8[1][0]*O[0] + g_aiT8[1][1]*O[1] + g_aiT8[1][2]*O[2]
+ g_aiT8[1][3]*O[3] + add)>>shift;
    dst[3*line] = (g_aiT8[3][0]*O[0] + g_aiT8[3][1]*O[1] +
g_aiT8[3][2]*O[2] + g_aiT8[3][3]*O[3] + add)>>shift;
    dst[5*line] = (g_aiT8[5][0]*O[0] + g_aiT8[5][1]*O[1] +
g_aiT8[5][2]*O[2] + g_aiT8[5][3]*O[3] + add)>>shift;
    dst[7*line] = (g_aiT8[7][0]*O[0] + g_aiT8[7][1]*O[1] +
g_aiT8[7][2]*O[2] + g_aiT8[7][3]*O[3] + add)>>shift;

    src += 8;
    dst ++;
  }
}

void partialButterfly16(Short *src,Short *dst,Int shift, Int line)
{
  Int j,k;
  Int E[8],O[8];
  Int EE[4],EO[4];
  Int EEE[2],EEO[2];
  Int add = 1<<(shift-1);

  for (j=0; j<line; j++)
  {
    /* E and O*/
    for (k=0;k<8;k++)
    {
      E[k] = src[k] + src[15-k];
      O[k] = src[k] - src[15-k];
    }
    /* EE and EO */
    for (k=0;k<4;k++)
    {
      EE[k] = E[k] + E[7-k];
      EO[k] = E[k] - E[7-k];
    }
    /* EEE and EEO */
    EEE[0] = EE[0] + EE[3];
    EEO[0] = EE[0] - EE[3];
    EEE[1] = EE[1] + EE[2];
    EEO[1] = EE[1] - EE[2];

   dst[ 0      ] = (g_aiT16[ 0][0]*EEE[0] + g_aiT16[ 0][1]*EEE[1] +
add)>>shift;
    dst[ 8*line ] = (g_aiT16[ 8][0]*EEE[0] + g_aiT16[ 8][1]*EEE[1] +
add)>>shift;
    dst[ 4*line ] = (g_aiT16[ 4][0]*EEO[0] + g_aiT16[ 4][1]*EEO[1] +
add)>>shift;
    dst[ 12*line] = (g_aiT16[12][0]*EEO[0] + g_aiT16[12][1]*EEO[1] +
add)>>shift;

    for (k=2;k<16;k+=4)
    {
      dst[ k*line ] = (g_aiT16[k][0]*EO[0] + g_aiT16[k][1]*EO[1] +
g_aiT16[k][2]*EO[2] + g_aiT16[k][3]*EO[3] + add)>>shift;
    }
    for (k=1;k<16;k+=2)
    {
      dst[ k*line ] = (g_aiT16[k][0]*O[0] + g_aiT16[k][1]*O[1] +
g_aiT16[k][2]*O[2] + g_aiT16[k][3]*O[3] +
        g_aiT16[k][4]*O[4] + g_aiT16[k][5]*O[5] + g_aiT16[k][6]*O[6] +
g_aiT16[k][7]*O[7] + add)>>shift;
    }

    src += 16;
    dst ++;

  }
}

void partialButterfly32(Short *src,Short *dst,Int shift, Int line)
{
  Int j,k;
  Int E[16],O[16];
  Int EE[8],EO[8];
  Int EEE[4],EEO[4];
  Int EEEE[2],EEEO[2];
  Int add = 1<<(shift-1);

  for (j=0; j<line; j++)
  {
    /* E and O*/
    for (k=0;k<16;k++)
    {
      E[k] = src[k] + src[31-k];
      O[k] = src[k] - src[31-k];
    }
    /* EE and EO */
    for (k=0;k<8;k++)
    {
      EE[k] = E[k] + E[15-k];
      EO[k] = E[k] - E[15-k];
    }
    /* EEE and EEO */
    for (k=0;k<4;k++)
    {
      EEE[k] = EE[k] + EE[7-k];
      EEO[k] = EE[k] - EE[7-k];
    }
    /* EEEE and EEEO */
    EEEE[0] = EEE[0] + EEE[3];
    EEEO[0] = EEE[0] - EEE[3];
    EEEE[1] = EEE[1] + EEE[2];
    EEEO[1] = EEE[1] - EEE[2];

    dst[ 0       ] = (g_aiT32[ 0][0]*EEEE[0] + g_aiT32[ 0][1]*EEEE[1] +
add)>>shift;
    dst[ 16*line ] = (g_aiT32[16][0]*EEEE[0] + g_aiT32[16][1]*EEEE[1] +
add)>>shift;
    dst[ 8*line  ] = (g_aiT32[ 8][0]*EEEO[0] + g_aiT32[ 8][1]*EEEO[1] +
add)>>shift;
```

```
    dst[ 24*line ] = (g_aiT32[24][0]*EEEO[0] + g_aiT32[24][1]*EEEO[1] + 
add)>>shift;
    for (k=4;k<32;k+=8)
    {
      dst[ k*line ] = (g_aiT32[k][0]*EEO[0] + g_aiT32[k][1]*EEO[1] + 
g_aiT32[k][2]*EEO[2] + g_aiT32[k][3]*EEO[3] + add)>>shift;
    }
    for (k=2;k<32;k+=4)
    {
      dst[ k*line ] = (g_aiT32[k][0]*EO[0] + g_aiT32[k][1]*EO[1] + 
g_aiT32[k][2]*EO[2] + g_aiT32[k][3]*EO[3] + 
        g_aiT32[k][4]*EO[4] + g_aiT32[k][5]*EO[5] + g_aiT32[k][6]*EO[6] + 
g_aiT32[k][7]*EO[7] + add)>>shift;
    }
    for (k=1;k<32;k+=2)
    {
      dst[ k*line ] = (g_aiT32[k][ 0]*O[ 0] + g_aiT32[k][ 1]*O[ 1] + 
g_aiT32[k][ 2]*O[ 2] + g_aiT32[k][ 3]*O[ 3] + 
        g_aiT32[k][ 4]*O[ 4] + g_aiT32[k][ 5]*O[ 5] + g_aiT32[k][ 6]*O[ 6] 
+ g_aiT32[k][ 7]*O[ 7] +
        g_aiT32[k][ 8]*O[ 8] + g_aiT32[k][ 9]*O[ 9] + g_aiT32[k][10]*O[10] 
+ g_aiT32[k][11]*O[11] +
        g_aiT32[k][12]*O[12] + g_aiT32[k][13]*O[13] + g_aiT32[k][14]*O[14] 
+ g_aiT32[k][15]*O[15] + add)>>shift;
    }

    src += 32;
    dst ++;
  }
}
```

APPENDIX B

The Multiplier Sharing Scheme of the High Throughput Architecture

Table. 17 The Multiplier Sharing Scheme of the High Throughput Architecture.

Muliplers No.	transform size (config)							
	32x32 ("11")		16x16 ("10")		8x8 ("01")		4x4 ("00")	
0	36x,83x	EEEO(0)-EEE(1)	36x,83x	EEO(0)-EEO(1)	36x,83x	EO(0)-EO(1)	36x,83x	O(0)-O(1)
1	18x	EEO(0)-EEO(3)	18x	EO(0)-EO(3)	18x	O(0)-O(3)		
2	50x	EEO(0)-EEO(3)	50x	EO(0)-EO(3)	50x	O(0)-O(3)		
3	75x	EEO(0)-EEO(3)	75x	EO(0)-EO(3)	75x	O(0)-O(3)		
4	89x	EEO(0)-EEO(3)	89x	EO(0)-EO(3)	89x	O(0)-O(3)		
5	9x	EO(0)-EO(3)	9x	O(0)-O(3)				
6	9x	EO(4)-EO(7)	9x	O(4)-O(7)				
7	25x	EO(0)-EO(3)	25x	O(0)-O(3)				
8	25x	EO(4)-EO(7)	25x	O(4)-O(7)				
9	43x	EO(0)-EO(3)	43x	O(0)-O(3)				
10	43x	EO(4)-EO(7)	43x	O(4)-O(7)				
11	57x	EO(0)-EO(3)	57x	O(0)-O(3)				
12	57x	EO(4)-EO(7)	57x	O(4)-O(7)				
13	70x	EO(0)-EO(3)	70x	O(0)-O(3)				
14	70x	EO(4)-EO(7)	70x	O(4)-O(7)				
15	80x	EO(0)-EO(3)	80x	O(0)-O(3)				
16	80x	EO(4)-EO(7)	80x	O(4)-O(7)				
17	87x	EO(0)-EO(3)	87x	O(0)-O(3)				
18	87x	EO(4)-EO(7)	87x	O(4)-O(7)				
19	90x	EO(0)-EO(3)	90x	O(0)-O(3)				
20	90x	EO(4)-EO(7)	90x	O(4)-O(7)				
21	13x	O(0)-O(3)	36x,83x	EEO(0)-EEO(1)	36x,83x	EO(0)-EO(1)	36x,83x	O(0)-O(1)
22	13x	O(4)-O(7)	18x	EO(0)-EO(3)	18x	O(0)-O(3)		
23	13x	O(8)-O(11)	50x	EO(0)-EO(3)	50x	O(0)-O(3)		
24	13x	O(12)-O(15)	75x	EO(0)-EO(3)	75x	O(0)-O(3)		
25	22x	O(0)-O(3)	89x	EO(0)-EO(3)	89x	O(0)-O(3)		
26	22x	O(4)-O(7)	9x	O(0)-O(3)				
27	22x	O(8)-O(11)	9x	O(4)-O(7)				
28	22x	O(12)-O(15)	25x	O(0)-O(3)				
29	31x	O(0)-O(3)	25x	O(4)-O(7)				
30	31x	O(4)-O(7)	43x	O(0)-O(3)				
31	31x	O(8)-O(11)	43x	O(4)-O(7)				
32	31x	O(12)-O(15)	57x	O(0)-O(3)				
33	38x	O(0)-O(3)	57x	O(4)-O(7)				
34	38x	O(4)-O(7)	70x	O(0)-O(3)				
35	38x	O(8)-O(11)	70x	O(4)-O(7)				
36	38x	O(12)-O(15)	80x	O(0)-O(3)				
37	46x	O(0)-O(3)	80x	O(4)-O(7)				
38	46x	O(4)-O(7)	87x	O(0)-O(3)				
39	46x	O(8)-O(11)	87x	O(4)-O(7)				
40	46x	O(12)-O(15)	90x	O(0)-O(3)				
41	54x	O(0)-O(3)	90x	O(4)-O(7)				
42	54x	O(4)-O(7)			36x,83x	EO(0)-EO(1)	36x,83x	O(0)-O(1)
43	54x	O(8)-O(11)			18x	O(0)-O(3)		
44	54x	O(12)-O(15)			50x	O(0)-O(3)		
45	61x	O(0)-O(3)			75x	O(0)-O(3)		
46	61x	O(4)-O(7)			89x	O(0)-O(3)		
47	61x	O(8)-O(11)			36x,83x	EO(0)-EO(1)	36x,83x	O(0)-O(1)
48	61x	O(12)-O(15)			18x	O(0)-O(3)		
49	67x	O(0)-O(3)			50x	O(0)-O(3)		
50	67x	O(4)-O(7)			75x	O(0)-O(3)		
51	67x	O(8)-O(11)			89x	O(0)-O(3)		
52	67x	O(12)-O(15)					36x,83x	O(0)-O(1)
53	73x	O(0)-O(3)					36x,83x	O(0)-O(1)
54	73x	O(4)-O(7)					36x,83x	O(0)-O(1)
55	73x	O(8)-O(11)					36x,83x	O(0)-O(1)

Table. 18 The Multiplier Sharing Scheme of the High Throughput Architecture (continue).

56	73x	O(12)-O(15)						
57	78x	O(0)-O(3)						
58	78x	O(4)-O(7)						
59	78x	O(8)-O(11)						
60	78x	O(12)-O(15)						
61	82x	O(0)-O(3)						
62	82x	O(4)-O(7)						
63	82x	O(8)-O(11)						
64	82x	O(12)-O(15)						
65	85x	O(0)-O(3)						
66	85x	O(4)-O(7)						
67	85x	O(8)-O(11)						
68	85x	O(12)-O(15)						
69	88x	O(0)-O(3)						
70	88x	O(4)-O(7)						
71	88x	O(8)-O(11)						
72	88x	O(12)-O(15)						
73	90x	O(0)-O(3)						
74	90x	O(4)-O(7)						
75	90x	O(8)-O(11)						
76	90x	O(12)-O(15)						

APPENDIX C

The Multiplier Sharing Scheme of the Flexible Input Architecture

Table. 19 The Multiplier Sharing Scheme of the Flexible Input Architecture.

Mulipliers No.	transform size (config)							
	32x32		16x16		8x8		4x4	
0							36x,83x	O(0)-O(1) [set 0]
1					18x	O(0)-O(3) [set 0]		
2					50x	O(0)-O(3) [set 0]		
3					75x	O(0)-O(3) [set 0]		
4					89x	O(0)-O(3) [set 0]		
5			9x	O(0)-O(3) [set 0]				
6			9x	O(4)-O(7) [set 0]				
7			25x	O(0)-O(3) [set 0]				
8			25x	O(4)-O(7) [set 0]				
9			43x	O(0)-O(3) [set 0]				
10			43x	O(4)-O(7) [set 0]				
11			57x	O(0)-O(3) [set 0]				
12			57x	O(4)-O(7) [set 0]				
13			70x	O(0)-O(3) [set 0]				
14			70x	O(4)-O(7) [set 0]				
15			80x	O(0)-O(3) [set 0]				
16			80x	O(4)-O(7) [set 0]				
17			87x	O(0)-O(3) [set 0]				
18			87x	O(4)-O(7) [set 0]				
19			90x	O(0)-O(3) [set 0]				
20			90x	O(4)-O(7) [set 0]				
21	13x	O(0)-O(3)					36x,83x	O(0)-O(1) [set 1]
22	13x	O(4)-O(7)			18x	O(0)-O(3) [set 1]		
23	13x	O(8)-O(11)			50x	O(0)-O(3) [set 1]		
24	13x	O(12)-O(15)			75x	O(0)-O(3) [set 1]		
25	22x	O(0)-O(3)			89x	O(0)-O(3) [set 1]		
26	22x	O(4)-O(7)	9x	O(0)-O(3) [set 1]				
27	22x	O(8)-O(11)	9x	O(4)-O(7) [set 1]				
28	22x	O(12)-O(15)	25x	O(0)-O(3) [set 1]				
29	31x	O(0)-O(3)	25x	O(4)-O(7) [set 1]				
30	31x	O(4)-O(7)	43x	O(0)-O(3) [set 1]				
31	31x	O(8)-O(11)	43x	O(4)-O(7) [set 1]				
32	31x	O(12)-O(15)	57x	O(0)-O(3) [set 1]				
33	38x	O(0)-O(3)	57x	O(4)-O(7) [set 1]				
34	38x	O(4)-O(7)	70x	O(0)-O(3) [set 1]				
35	38x	O(8)-O(11)	70x	O(4)-O(7) [set 1]				
36	38x	O(12)-O(15)	80x	O(0)-O(3) [set 1]				
37	46x	O(0)-O(3)	80x	O(4)-O(7) [set 1]				
38	46x	O(4)-O(7)	87x	O(0)-O(3) [set 1]				
39	46x	O(8)-O(11)	87x	O(4)-O(7) [set 1]				
40	46x	O(12)-O(15)	90x	O(0)-O(3) [set 1]				
41	54x	O(0)-O(3)	90x	O(4)-O(7) [set 1]				
42	54x	O(4)-O(7)					36x,83x	O(0)-O(1) [set2]
43	54x	O(8)-O(11)			18x	O(0)-O(3) [set 2]		
44	54x	O(12)-O(15)			50x	O(0)-O(3) [set 2]		
45	61x	O(0)-O(3)			75x	O(0)-O(3) [set 2]		
46	61x	O(4)-O(7)			89x	O(0)-O(3) [set 2]		
47	61x	O(8)-O(11)					36x,83x	O(0)-O(1) [set 3]
48	61x	O(12)-O(15)			18x	O(0)-O(3) [set 3]		
49	67x	O(0)-O(3)			50x	O(0)-O(3) [set 3]		
50	67x	O(4)-O(7)			75x	O(0)-O(3) [set 3]		
51	67x	O(8)-O(11)			89x	O(0)-O(3) [set 3]		
52	67x	O(12)-O(15)					36x,83x	O(0)-O(1) [set 4]
53	73x	O(0)-O(3)					36x,83x	O(0)-O(1) [set5]
54	73x	O(4)-O(7)					36x,83x	O(0)-O(1) [set 6]
55	73x	O(8)-O(11)					36x,83x	O(0)-O(1) [set 7]

Table. 20 The Multiplier Sharing Scheme of the Flexible Input Architecture (continue).

56	73x	O(12)-O(15)						
57	78x	O(0)-O(3)						
58	78x	O(4)-O(7)						
59	78x	O(8)-O(11)						
60	78x	O(12)-O(15)						
61	82x	O(0)-O(3)						
62	82x	O(4)-O(7)						
63	82x	O(8)-O(11)						
64	82x	O(12)-O(15)						
65	85x	O(0)-O(3)						
66	85x	O(4)-O(7)						
67	85x	O(8)-O(11)						
68	85x	O(12)-O(15)						
69	88x	O(0)-O(3)						
70	88x	O(4)-O(7)						
71	88x	O(8)-O(11)						
72	88x	O(12)-O(15)						
73	90x	O(0)-O(3)						
74	90x	O(4)-O(7)						
75	90x	O(8)-O(11)						
76	90x	O(12)-O(15)						

VITA

Pancheewa Arayacheeppreecha was born in 1992 in Bangkok, Thailand. She received the Bachelor Degree of Electrical Engineering from Chulalongkorn University in 2013. She has been pursuing for the Master Degree of Electrical Engineering at Chulalongkorn University, Bangkok, Thailand, since 2014. Her research interests include video compression, and Field Programmable Gate Arrays (FPGAs).