



University of Thessaly
Department of Computer & Communication Engineering

Κανονικό Συνθήκη ορισμού Λίστα ορισμών Διεύθυνση Σχεδίαση και
Υλοποίηση Αρχιτεκτονικής Υψηλής Απόδοσης για τους Αλγορίθμους
Μετασχηματισμού και Κβαντοποίησης του H.264

Student
Muhsen Owaida

Advisor
Prof. George Stamulis

VOLOS, 25 APRIL 2007



**ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ
ΒΙΒΛΙΟΘΗΚΗ & ΚΕΝΤΡΟ ΠΛΗΡΟΦΟΡΗΣΗΣ
ΕΙΔΙΚΗ ΣΥΛΛΟΓΗ «ΓΚΡΙΖΑ ΒΙΒΛΙΟΓΡΑΦΙΑ»**

Αριθ. Εισ.: 5266/1
Ημερ. Εισ.: 27-09-2007
Δωρεά: Συγγραφέα
Ταξιθετικός Κωδικός: Δ
005.1
OWA

Acknowledgment

The Author would like to thank the PhD Graduates Students Lab members in the Communication & Computer Engineering Department of Thessaly University for their Help and Support of this project. Special Thanks for Ms. Maria Koziri and Mr. Dimitrios Karampatzakis for their Interest and support. And Special Thanks for My advisor; Prof. George Stamulis, for his concepts and encouragement.

I would like also to thank my family and friends, all for their encouragement, and good atmosphere that they saved for me.

Abstract

Efficient digital video coding techniques are increasingly gaining importance due to the widespread of low bit rate video streaming applications (like videotelephony and videoconferencing). This raises the need for an industry standard for compressed video representation with substantially increased coding efficiency and enhanced robustness to network environments.

In 2001, the Joint Video Team (JVT) was formed to represent the cooperation between the ITU-T Video Coding Expert Group (VCEG) and the ISO/IEC Moving Picture Expert Group (MPEG) aiming for the development of a new Standard. The JVT aim was to finalize the H.26L proposal and convert it into an international standard (H.264/MPEG-4 Part 10) published by both ISO/IEC and ITU-T.

H.264 provides similar functionality to earlier standards such as H.263+ and MPEG-4 Visual (Simple Profile) but with significantly better compression performance and improved support for reliable transmission. It does not use the traditional 8×8 DCT transform as the basic transform, instead it suggests 4×4 DCT-based transform that can be implemented only using integer addition and shift units and avoids use of multiplication.

In this project, a hardware prototype is designed for the H.264 supported quantization and variant types of supported transforms (core transform, 2×2 and 4×4 hadmard transforms). Also the inverse transform and quantization path is considered. The architecture is prototyped and simulated using ModelSim 6.1®. It is synthesized using Synopsys Design Compiler®.

Contents

Acknowledgment	i
Abstract	ii
Contents.....	iii
Figures' List	v
Tables' List.....	viii
Summary in Greek	1
Μετασχηματισμός και κβαντοποίηση στο H.264	1
Σχεδίαση σε επίπεδο RTL των αλγορίθμων μετασχηματισμού και κβαντοποίησης του H.264.....	4
Σύνθεση και Υπολογισμός Ισχύος	6
Appendix A	8
Introduction.	9
1.1. Video Format and Encoding.....	9
1.1.1. Video Format.....	9
1.1.2. Video CODEC.....	12
1.1.3. Video Encoding Standards.....	17
1.2. H.264 Video Encoding Standard.	18
1.2.1. H.264 Structure.	18
1.2.2. H.264 CODEC.	18
1.3. Transformation and Quantization of H.264 standard.	22
1.3.1. Transformation.	22
1.3.2. Quantization.	24
1.3.3. Reforming Quantization Process.....	27
1.4. Previous Work.	33
2.RTL Level Design of H.264 Transform and Quantization.....	35
2.1. Transform.....	35
2.1.1. Core Transform	36
2.1.2. Hadmard $4 \times 4/2 \times 2$ Transform.	39
2.1.3. Configurator.	43
2.1.4. Faster Implementation of 4×4 Hadmard Transform.	44
2.2. Standard Quantization Implementation.	49
2.2.1. Quantization Parameters.	49
2.2.2. Quantization phase.	50

2.3.	Quantization using f -Modification method.....	59
2.3.1.	f -Modification Stage.....	60
2.3.2.	Addition Stage.....	61
2.4.	Inverse Transform and Quantization.	62
2.4.1.	Inverse 4×4 & 2×2 Hadamard Transforms.	63
2.4.2.	V-Factors Calculation.	64
2.4.3.	Rescale Unit.	64
2.4.4.	Inverse Core Transform.	65
2.4.5.	Rounding Unit.	67
3.	Optimization and Synthesis.	69
3.1.	Design Optimization.	69
3.1.1.	Modeling of Optimization problem.	69
3.1.2.	Optimization Levels.	70
3.2.	Design Synthesis.....	73
3.2.1.	Synthesis Setup.	73
3.2.2.	Synthesis Process.	76
3.3.	Transformation and Quantization Synthesis.....	77
3.3.1.	Transform Synthesis Results.....	81
3.3.2.	Quantization Synthesis Results.	87
3.3.3.	Transform and Quantization Integration.....	95
3.3.4.	Inverse Transform and Quantization Synthesis.	99
3.4	Conclusion and Final Results.	100
Appendix B: <i>Synopsys Design Compiler</i>		103
References.		115

Figures List

Εικόνα 1: Ένας H.264 κωδικοποιητής	1
Εικόνα 2: Διάγραμμα του σταδίου του Μετασχηματισμού	4
Εικόνα 3: Διάγραμμα του Μετασχηματισμού Hadmard.....	4
Εικόνα 4: Διάγραμμα του «πυρήνα» του Μετασχηματισμού.....	5
Εικόνα 5: Διάγραμμα του κυκλώματος υπολογισμού των παραμέτρων	5
Εικόνα 6: Διάγραμμα του κυκλώματος που χρησιμοποιεί την μέθοδο f-Modification για την υλοποίηση της κβαντοποίησης,	6
Figure1.1: Video signal structure	9
Figure1.2: 4:2:0, 4:2:2 and 4:4:4 sampling patterns (progressive).	12
Figure1.3: Video transmission system	13
Figure1.4: Video Processing system	13
Figure1.5: Close-up of 4×4 block; DCT coefficients	15
Figure1.6: Scalar quantizer: linear; nonlinear with dead zone	16
Figure 1.7: Zigzag scan order (frame block).....	17
Figure1.8: H.264 CODEC.....	19
Figure1.9: presented architecture in [1].	34
Figure2.1: Transform Stage Block Diagram.....	35
Figure2.2: First stage of core transform butterfly analysis.	37
Figure2.3: second stage core transform butterfly analysis.....	37
Figure2.4: Core Transform Structure.....	37
Figure2.5: structure of butterfly block used in core transform	38
Figure2.6: first stage butterfly of 4×4 hadmard transform analysis.....	40
Figure2.7: second stage butterfly of 4×4 hadmard transform analysis.....	40
Figure2.8: butterfly block used for 2×2 Hadmard transform.....	41
Figure2.9: Implementation of hadmard transforms without configurator ...	41
Figure2.10: Implementation of hadmard transforms using configurator.....	41
Figure2.11: 2×2 and 4×4 hadmard transforms block with configurator.....	42
Figure2.12: Inputs order to 2×2 and 4×4 Hadmard transforms	43
Figure2.13: Configurator Structure.....	44
Figure2.14: Butterfly1_1N.....	46
Figure2.15: Butterfly1_2N.....	46
Figure2.16: First Term of equation (2.12) implementation.	47
Figure2.17: Butterfly1_1N used for Third term calculation.	47
Figure2.18: Third Term in equation (2.12) implementation.	48
Figure2.19: Blocks used to add the last three terms in equation (2.12).	48

Figure2.20: Quantization parameters calculation structure.	50
Figure2.21: Standard quantization data flow.	51
Figure2.22: absolute value implementation.	51
Figure2.23: Absolute value Unit.	52
Figure2.24: Multiplication Unit structure.	53
Figure2.25: Array multiplier.	53
Figure2.26: Tree Multiplier.	54
Figure2.27: 14×15 Tree multiplier architecture.	54
Figure2.28: Carry-save multiplier architecture.	55
Figure2.29: <i>f</i> -modification structure.	55
Figure2.30: Addition Unit Structure.	57
Figure2.31: 4-bits funnel shifter.	57
Figure2.32: Shift Unit structure.	58
Figure2.33: resign function structure.	58
Figure2.34: Resigning Unit structure.	59
Figure2.35: Quantization using <i>f</i> -Modification Method structure.	60
Figure2.36: F-Modification Stage structure.	60
Figure2.37: Addition Unit structure.	62
Figure2.38: Inverse transform and quantization structure.	63
Figure3.39: Inverse Hadmard transform unit.	63
Figure2.40: V-Factors calculation.	64
Figure2.41: Rescale Unit Structure.	64
Figure2.42: 5×19 tree multiplier used in rescale unit.	65
Figure2.43: Butterfly's block structure used in inverse core transform.	66
Figure2.44: First level butterfly block used in inverse core transform.	66
Figure2.45: second level butterfly block used in inverse core transform. ...	67
Figure2.46: Division by 64 and round implementation.	68
Figure3.1: Sharing resources.	71
Figure3.2: Factorizing expressions.	71
Figure3.3: Parallel processing.	72
Figure3.4: logic level optimizations.	72
Figure3.5: minimizations using Demorgan's laws.	73
Figure3.6: Design files Hierarchy.	74
Figure3.7: Design environment parameters.	75
Figure3.8: Transform_CONF design files hierarchy structure.	78
Figure3.9: Transform_No_CONF design files hierarchy structure.	78
Figure3.10: 4×4 hadmard transform implementation subdesign.	78
Figure3.11: Quantization_FM_TRM design files hierarchy structure.	79

Figure3.12: Quantization_STM_TRM design files hierarchy structure.....	79
Figure3.13: CSMull_Unit design files hierarchy structure.....	79
Figure3.14: Transform implementation with configurator schematic.	81
Figure3.15: full path slack histogram (Transform_CONF).	82
Figure3.16: Transform implementation without configurator schematic.	83
Figure3.17: full path slack histogram (Transform_No_CONF).	84
Figure3.18: Transform_N_HAD schematics.	85
Figure3.19: full path slack histogram (Transform_N_HAD).	85
Figure3.20: Quantization_STM_TRM schematic.....	87
Figure3.21: full path delay slack histogram (Quantization_STM_TRM). ...	88
Figure3.22: full path delay slack histogram (Quantization_STM_CSM)....	89
Figure3.23: Quantization_FM_TRM schematic.	91
Figure3.24: full path delay slack histogram (Quantization_FM_TRM).....	91
Figure3.25: Quantization_FM_CSM schematic.	93
Figure3.26: full path delay slack histogram (Quantization_FM_CSM).	93
Figure3.27: Tr_QuFTRM_Phases schematic.....	95
Figure3.28: Full path delay histogram (Tr_QuFTRM_Phases).....	89
Figure3.29: full path delay slack histogram (Tr_QuSTRM_Phases).....	98
Figure3.30: Inverse_TR_Qu schematic.	99
Figure3.31: full path delay slack histogram (Inverse_TR_Qu).	100
Figure3.32: Forward and reverse paths inputs and outputs.	101

Tables List

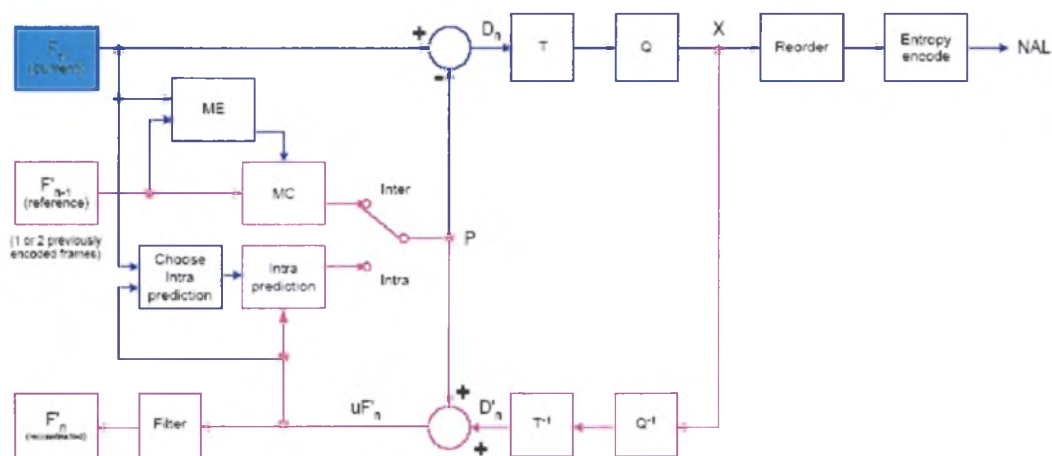
Πίνακας 1: Απόδοση των αρχιτεκτονικών για το στάδιο του Μετασχηματισμού	6
Πίνακας 2: Απόδοση των αρχιτεκτονικών για το στάδιο της Κβαντοποίησης.....	7
Πίνακας 3: Απόδοση τελικών κυκλωμάτων.....	7
Table1.1: MPEG-4 VISUAL and H.264 standards comparison.	21
Table1.2: QP values and corresponding Qsteps.	25
Table1.3: PF values	25
Table1.4: MF multiplication factor.....	26
Table1.5: V scaling factor.....	26
Table2.1: funnel shifter map table.	57
Table3.1: Created top-level design files	77
Table3.2: Wire load model.	80
Table3.3: critical path (Transform_CONF).....	81
Table3.4: design area	82
Table3.5: Total consumed power	82
Table3.6: cells' internal consumed power.	82
Table3.7: critical path (Transform_No_CONF).....	83
Table3.8: design area.	84
Table3.9: Total consumed power	84
Table3.10: cells' internal consumed power.	84
Table3.11: critical path (Transform_N_HAD).....	85
Table3.12: design area.	86
Table3.13: Total consumed power	86
Table3.14: cells' internal consumed power.	86
Table3.15: critical path (Quantization_STM_TRM).....	87
Table3.16: design area.	88
Table3.17: Total consumed power	88
Table3.18: cells' internal consumed power.	88
Table3.19: critical path (Quantization_STM_CSM).....	89
Table3.20: design area.	90
Table3.21: Total consumed power	90
Table3.22: cells' internal consumed power.	90
Table3.23: critical path (Quantization_FM_TRM).	90
Table3.24: design area.	92

Table3.25: Total consumed power	92
Table3.26: cells' internal consumed power.	92
Table3.27: critical path (Quantization_FM_CSM).	92
Table3.28: design area.	93
Table3.29: Total consumed power	94
Table3.30: cells' internal consumed power.	94
Table3.31: critical path (TR_QuFTRM).	96
Table3.32: design area (TR_QuFTRM).	96
Table3.33: Total consumed power (TR_QuFTRM).....	97
Table3.34: cells' internal consumed power (TR_QuFTRM).	97
Table3.35: critical path (TR_QuSTRM).	97
Table3.36: design area (TR_QuSTRM).	97
Table3.37: Total consumed power (TR_QuSTRM).....	98
Table3.38: cells' internal consumed power (TR_QuSTRM).	98
Table3.39: critical path (Inverse Path).	99
Table3.40: design area (Inverse Path).	99
Table3.41: Total consumed power (Inverse Path).....	100
Table3.42: cells' internal consumed power (Inverse Path).	100

1 Μετασχηματισμός και κβαντοποίηση στο H.264

Οι τεχνικές κωδικοποίησης video είναι όλο και περισσότερο σημαντικές, εξαιτίας της ραγδαίας εμφάνισης εφαρμογών που απαιτούν ροές video χαμηλού bit – rate, όπως η video – τηλεφωνία και η video – συνεδρία. Το γεγονός αυτό κάνει απαραίτητη την ύπαρξη ενός βιομηχανικού standard για αναπαράσταση κωδικοποιημένου video με υψηλή απόδοση συμπίεσης και ενισχυμένη αντοχή σε δικτυακά περιβάλλοντα. Το πιο πρόσφατο standard στην περιοχή της κωδικοποίησης video είναι το H.264, το οποίο δημιουργήθηκε από την Joint Video Team (JVT), μια συνεργασία μεταξύ των ITU-T Video Coding Expert Group (VCEG) και ISO/IEC Moving Picture Expert Group (MPEG).

Το H.264 παρέχει παρόμοια λειτουργικότητα με τα προηγούμενα standards, επιτυγχάνοντας όμως σημαντική βελτίωση στην συμπίεση και βελτιωμένη υποστήριξη για αξιόπιστη μετάδοση. Στην Εικόνα 1 παρουσιάζεται το σχεδιάγραμμα ενός H.264 κωδικοποιητή.



Εικόνα 1. Ένας H.264 κωδικοποιητής

Ένας από τους λόγους που το H.264 επιτυγχάνει τόσο υψηλή απόδοση στην κωδικοποίηση είναι οι αλλαγές που έχουν γίνει στους αλγορίθμους μετασχηματισμού και κβαντοποίησης, σε σχέση με αυτούς που χρησιμοποιούνταν στα προηγούμενα standard. Στην εργασία αυτή παρουσιάζεται η σχεδίαση και η υλοποίηση μιας καινούργιας αρχιτεκτονικής για τους αλγορίθμους μετασχηματισμού και κβαντοποίησης του H.264, η οποία αυξάνει σημαντικά την απόδοση ενώ ταυτόχρονα μειώνει την κατανάλωση ισχύος που απαιτεί ένα τέτοιο σύστημα.

Πριν προχωρήσουμε όμως στην παρουσίαση της αρχιτεκτονικής, ας δούμε περιληπτικά τους αλγορίθμους μετασχηματισμού και κβαντοποίησης του H.264 καθώς και τις βελτιστοποιήσεις που έγιναν στα πλαίσια αυτής της εργασίας.

Στα προηγούμενα standards ήταν ευρέως διαδεδομένη η χρήση του DCT μετασχηματισμού. Το H.264 αντικαθιστά τον DCT με έναν ακέραιο μετασχηματισμό, ο οποίος απαιτεί μόνο προσθήσεις και ολισθήσεις, απαλείφοντας με αυτόν τον τρόπο το κόστος του πολλαπλασιασμού που απαιτείται από τον DCT. Ο ακέραιος μετασχηματισμός του H.264 δίνεται από την παρακάτω συνάρτηση:

$$Y = (C_f X C_f^T) \otimes E_f = \left(\begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & -2 \\ 1 & -1 & -1 & 1 \\ 1 & -2 & 2 & -1 \end{bmatrix} * \begin{bmatrix} X \end{bmatrix} * \begin{bmatrix} 1 & 2 & 1 & 1 \\ 1 & 1 & -1 & -2 \\ 1 & -1 & -1 & 2 \\ 1 & -2 & 1 & -1 \end{bmatrix} \right) \otimes \begin{bmatrix} a^2 & \frac{ab}{2} & a^2 & \frac{ab}{2} \\ \frac{ab}{2} & \frac{b^2}{4} & \frac{ab}{2} & \frac{b^2}{4} \\ 2 & 4 & 2 & 4 \\ a^2 & \frac{ab}{2} & a^2 & \frac{ab}{2} \\ \frac{ab}{2} & \frac{b^2}{4} & \frac{ab}{2} & \frac{b^2}{4} \\ 2 & 4 & 2 & 4 \end{bmatrix} \quad (1)$$

Όπου, $a = \frac{1}{2}$, $b = \sqrt{\frac{2}{5}}$ και $d = \frac{1}{2}$. Στην παραπάνω συνάρτηση το σύμβολο \otimes σημαίνει ότι έχουμε πολλαπλασιασμό μεταξύ αντίστοιχων στοιχείων και όχι πολλαπλασιασμό πινάκων. Αυτό απλοποιεί πολύ τον αλγόριθμο του μετασχηματισμού, αφού ο πολλαπλασιασμός αυτός μπορεί να ενσωματωθεί στο στάδιο της κβαντοποίησης. Με αυτόν τον τρόπο ο «πυρήνας» του μετασχηματισμού μπορεί να θεωρηθεί ότι είναι απλά ο CXC^T .

Σε αντιστοιχία με τα παραπάνω, υπάρχει και ο αντίστροφος μετασχηματισμός, ο οποίος δίνεται από τη συνάρτηση:

$$Y = C_i^T (Z \otimes E_i) C_i = \begin{bmatrix} 1 & 1 & 1 & \frac{1}{2} \\ 1 & \frac{1}{2} & -1 & -1 \\ 1 & -\frac{1}{2} & -1 & 1 \\ 1 & -1 & 1 & -\frac{1}{2} \end{bmatrix} * \left(\begin{bmatrix} Z \end{bmatrix} \otimes \begin{bmatrix} a^2 & ab & a^2 & ab \\ ab & b^2 & ab & b^2 \\ a^2 & ab & a^2 & ab \\ ab & b^2 & ab & b^2 \end{bmatrix} \right) * \begin{bmatrix} 1 & \frac{1}{2} & -\frac{1}{2} & 1 \\ 1 & \frac{1}{2} & -\frac{1}{2} & -1 \\ \frac{1}{2} & -1 & -1 & \frac{1}{2} \\ \frac{1}{2} & -1 & 1 & -\frac{1}{2} \end{bmatrix} \quad (2)$$

Και σε αυτήν την περίπτωση ο πολλαπλασιασμός με τον πίνακα E_i μπορεί να ενσωματωθεί στο στάδιο της αποκλιμάκωσης (re-scaling), οπότε και ο «πυρήνας» του αντίστοιχου μετασχηματισμού μπορεί να θεωρηθεί ότι είναι απλά ο $C^T Z C$.

Σε δυο περιπτώσεις, σε αυτήν του 16x16 Intra mode και σε αυτήν που έχουμε Chroma στοιχεία, το H.264, πέραν του ακέрайου μετασχηματισμού που μόλις περιγράψαμε, χρησιμοποιεί και τον μετασχηματισμό Hadmard. Για την πρώτη περίπτωση, δηλαδή όταν έχουμε 16x16 Intra mode ο Hadmard μετασχηματισμός δίνεται από την εξής συνάρτηση:

$$Y_D = \left(\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} * \begin{bmatrix} W_D \end{bmatrix} * \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} \right) / 2 \quad (3)$$

και ο αντίστροφος μετασχηματισμός από τη συνάρτηση:

$$W_D = \left(\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} * \begin{bmatrix} Z_D \end{bmatrix} * \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} \right) \quad (4)$$

Στην περίπτωση που δεν έχουμε στοιχεία φωτεινότητας αλλά έχουμε Chroma Components, ο Hadmard μετασχηματισμός δίνεται από την εξής συνάρτηση:

$$W_{QD} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} * \begin{bmatrix} W_D \end{bmatrix} * \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \quad (5)$$

Το επόμενο στάδιο μετά τον μετασχηματισμό είναι η κβαντοποίηση. Η διαδικασία της κβαντοποίησης μπορεί να αναπαρασταθεί από την ακόλουθη συνάρτηση:

$$Z_{ij} = \text{round}\left(\frac{Y_{ij}}{Qstep}\right) \quad (6)$$

Όπου το Y_{ij} είναι ο συντελεστής που προκύπτει από τον μετασχηματισμό, και $Qstep$ είναι ο συντελεστής κβαντισμού. Το H.264 ορίζει την κβαντοποίηση με την εξής διαδικασία:

$$Z_{ij} = \text{round}\left(w_{ij} \frac{MF}{2^{qbits}}\right) \quad (7)$$

Όπου

$$\frac{MF}{2^{qbits}} = \frac{PF}{Qstep}$$

και

$$qbits = 15 + \text{floor}\left(\frac{QP}{6}\right)$$

Επιπλέον ο παράγοντας PF είναι είτε a^2 ή $ab/2$ ή $b^2/4$, αναλόγως τη θέση που βρίσκεται ο συντελεστής.

Σε ακέραια αριθμητική η εξίσωση (7) μπορεί να αποδοθεί ως εξής:

$$\begin{aligned} \left(|Z_{ij}| = \left(|W_{ij}| \cdot MF + f\right) \gg qbits\right. \\ \left. \text{sign}(Z_{ij}) = \text{sign}(W_{ij})\right) \end{aligned} \quad (8)$$

Η διαδικασία της αποκλιμάκωσης, η οποία είναι η αντίστροφη από αυτή του κβαντισμού μπορεί να περιγραφεί από τη συνάρτηση:

$$\widehat{W}_{ij} = Z_{ij} * Qstep * PF * 64 \quad (9)$$

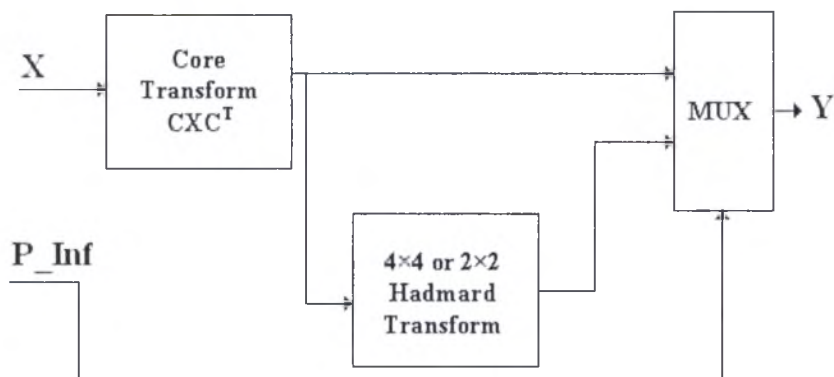
ή αντίστοιχα από τη συνάρτηση

$$\widehat{W}_{ij} = Z_{ij} * V_{ij} * 2^{\text{floor}(QP/6)} \quad (10)$$

Ένα σημαντικό μέρος της δουλειάς αυτής είναι η βελτιστοποίηση που έχει γίνει στη διαδικασία της κβαντοποίησης, και συγκεκριμένα η τροποποίηση του παράγοντα f . Η τροποποίηση αυτή βασίζεται στον διαφορετικό τρόπο που χρησιμοποιείται για την στρογγυλοποίηση των αριθμών και αναλύεται λεπτομερώς στο παράρτημα που ακολουθεί.

2 Σχεδίαση σε επίπεδο RTL των αλγορίθμων μετασχηματισμού και κβαντοποίησης του H.264.

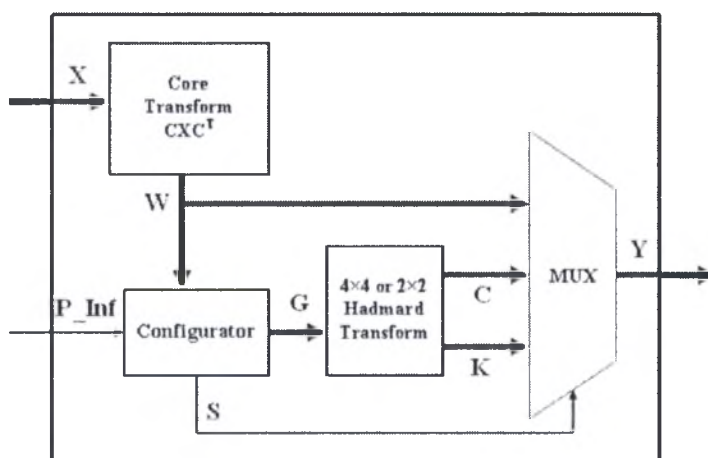
Στο σημείο αυτό ήρθε η ώρα να παρουσιάσουμε την αρχιτεκτονική που αναπτύχθηκε για την υλοποίηση των αλγορίθμων που περιγράφηκαν στην προηγούμενη ενότητα. Στην Εικόνα 2 παρουσιάζεται το διάγραμμα του σταδίου του μετασχηματισμού.



Εικόνα 2. Διάγραμμα του σταδίου του Μετασχηματισμού

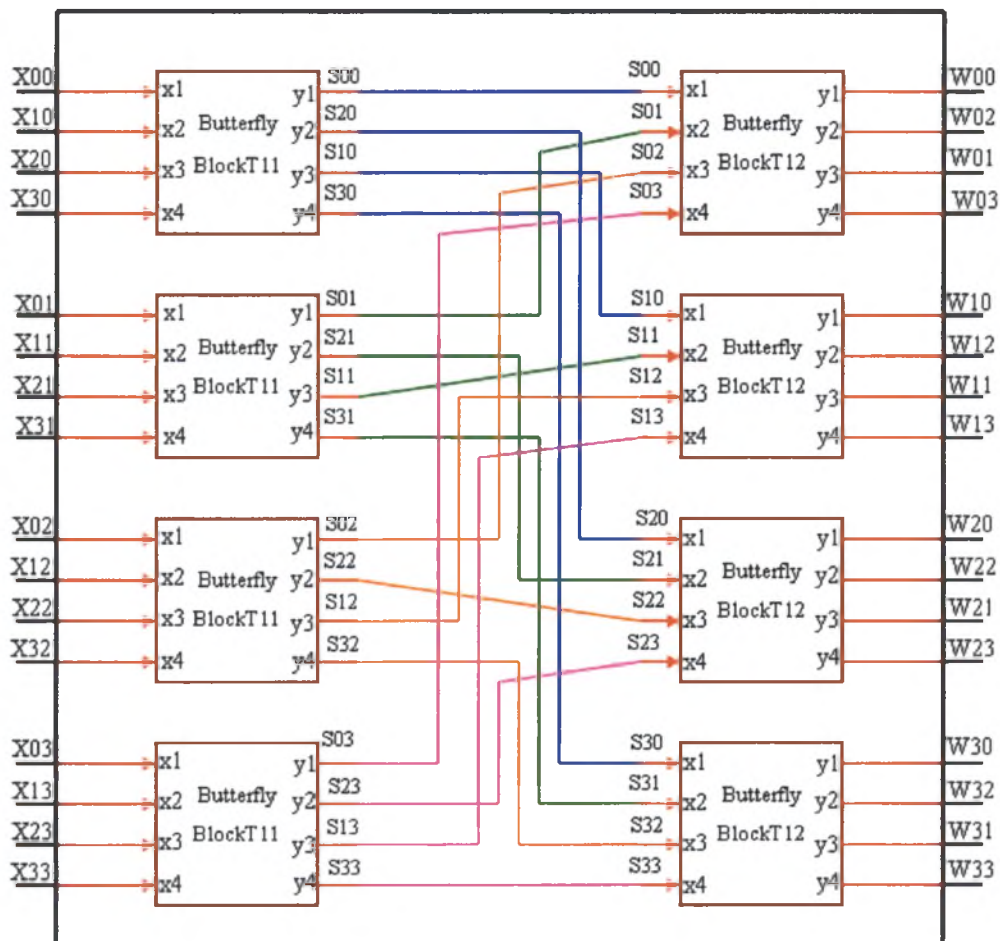
Το κύκλωμα που υλοποιεί τον μετασχηματισμό δέχεται ως εισόδους τον δίαυλο X, μέσω του οποίου έρχονται τα δεδομένα του προς επεξεργασία 4x4 block, και έναν 2-bit αριθμό, τον P_Inf, ο οποίος μεταφέρει κωδικοποιημένη πληροφορία για τα στοιχεία προς επεξεργασία καθώς και τον τύπο πρόβλεψης. Η έξοδός του είναι ο δίαυλος Y.

Η υλοποίηση του «πυρήνα» του μετασχηματισμού γίνεται από το κύκλωμα του οποίου το διάγραμμα παρουσιάζεται στην εικόνα 4, ενώ το διάγραμμα του κυκλώματος που υλοποιεί τον Hadmard μετασχηματισμό φαίνεται στην εικόνα 3.



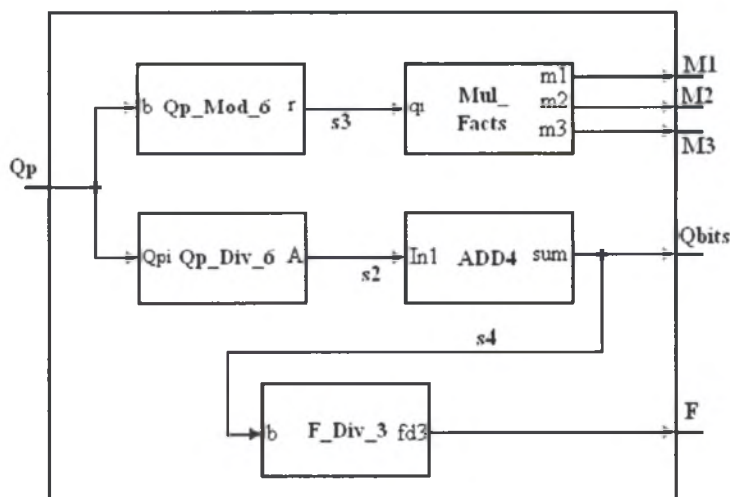
Εικόνα 3. Διάγραμμα του Μετασχηματισμού Hadmard

Αξίζει να σημειωθεί στο σημείο αυτό ότι η αρχιτεκτονική του κυκλώματος που υλοποιεί τον μετασχηματισμό Hadmard είναι πρωτότυπη και παρουσιάζει σημαντικά βελτιωμένα αποτελέσματα σε σχέση με προηγούμενες αρχιτεκτονικές που έχουν παρουσιαστεί στη βιβλιογραφία.

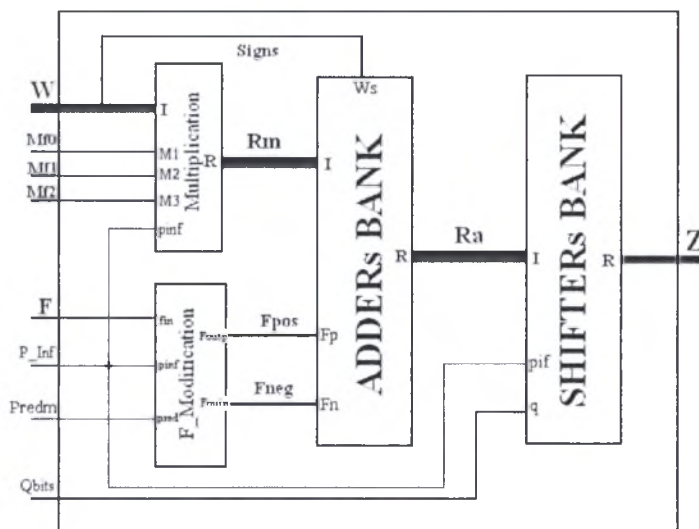


Εικόνα 4. Διάγραμμα του «πυρήνα» του Μετασχηματισμού

Το στάδιο της κβαντοποίησης μπορεί να θεωρηθεί ότι αποτελείται από δυο μέρη. Στο ένα υπολογίζονται όλοι οι παράμετροι οι οποίοι είναι απαραίτητοι για να υπολογιστεί η συνάρτηση 8 και στο άλλο στάδιο υλοποιείται η συνάρτηση αυτή. Στις Εικόνες 5 και 6 φαίνονται τα διαγράμματα των κυκλωμάτων που υπολογίζουν τις παραμέτρους και τη συνάρτηση 8, αντίστοιχα.



Εικόνα 5. Διάγραμμα του κυκλώματος υπολογισμού των παραμέτρων



Εικόνα 6. Διάγραμμα του κυκλώματος που χρησιμοποιεί την μέθοδο f-Modification για την υλοποίηση της κβαντοποίησης,

Πριν προχωρήσουμε στην παρουσίαση μερικών αποτελεσμάτων από τις υλοποιήσεις των διαφόρων αρχιτεκτονικών που περιγράφονται αναλυτικά στο παράρτημα, μερικές από τις οποίες είναι αυτές στις οποίες μόλις αναφερθήκαμε, πρέπει να τονίσουμε ότι πέραν των αρχιτεκτονικών αυτών έχουν σχεδιαστεί και υλοποιηθεί όλες οι αντίστοιχες αρχιτεκτονικές για τον αντίστροφο μετασχηματισμό και την αποκλιμάκωση.

3 Σύνθεση και Υπολογισμός Ισχύος

Οι διάφορες αρχιτεκτονικές που σχεδιάστηκαν για τους αλγορίθμους του μετασχηματισμού και της κβαντοποίησης, περιγράφηκαν και προσομοιώθηκαν με την χρήση του ModelSim®. Τα κυκλώματα που δημιουργήθηκαν από τη διαδικασία αυτή συντέθηκαν με τη χρήση του Synopsys Design Compiler® ενώ η κατανάλωση ισχύος υπολογίστηκε με τη βοήθεια του Synopsys PrimePower®. Στη συνέχεια παρουσιάζονται μερικά από τα αποτελέσματα που προέκυψαν από την παραπάνω διαδικασία.

Για τον μετασχηματισμό υλοποιήθηκαν τρεις διαφορετικές αρχιτεκτονικές και η απόδοσή τους συνοψίζεται στον Πίνακα 1.

Πίνακας 1. Απόδοση των αρχιτεκτονικών για το στάδιο του Μετασχηματισμού

	Transform_CONF	Transform_No_CONF	Transform_N_HAD
Delay (ns)	6.82	6.57	5.3
Area	178937.859375	179554.750000	186347.515625
Cells#	11818	14137	13369
Dynamic power	31.7733 mW	37.9148 mW	46.7057 mW
Leakage power	323.9445 uW	278.0365 uW	303.4926 uW

Για την κβαντοποίηση υλοποιήθηκαν δυο διαφορετικές αρχιτεκτονικές και η απόδοσή τους συνοψίζεται στον Πίνακα 2.

Πίνακας 2. Απόδοση των αρχιτεκτονικών για το στάδιο της Κβαντοποίησης

	Standard Method		f-Modification Method	
	Tree Multiplier	Carry-save Multiplier	Tree Multiplier	Carry-save Multiplier
Delay (ns)	8.63065	9.21604	6.3815	6.98
Area	412912.500	336792.375	418205.375	326448.562
Cells#	33881	23058	32247	22178
Dynamic Power.	40.9392 mW	51.5387 mW	37.4036 mW	47.8006 mW
Leakage Power.	409.9400 uW	584.6456 uW	349.1045 uW	555.1185 uW

Η εργασία αυτή ολοκληρώνεται με την ενοποίηση των δυο αλγορίθμων, μετασχηματισμού και κβαντοποίησης, σε ένα κύκλωμα, καθώς και με την ενοποίηση των δυο αντίστοιχων, αντίστροφων αλγορίθμων, του αντίστροφου μετασχηματισμού και της αποκλιμάκωσης, σε ένα δεύτερο κύκλωμα. Η απόδοση των δυο αυτών, τελικών, κυκλωμάτων παρουσιάζεται στον Πίνακα 3.

Πίνακας 3. Απόδοση τελικών κυκλωμάτων

	Forward Path	Inverse Path
Delay (ns)	8.78	8.8
Dynamic power (mW)	56.6784	37.4244
Leakage power (uW)	1030.2	799.2933
Area	788451	560350
Cells#	56524	39985

Εάν κάποιος επιχειρήσει να κάνει μια σύγκριση μεταξύ των αποτελεσμάτων που παρουσιάζονται στον Πίνακα 3, και αυτών που υπάρχουν διαθέσιμα στη βιβλιογραφία, τότε θα διαπιστώσει ότι οι αρχιτεκτονικές που αναπτύχθηκαν στα πλαίσια αυτής της εργασίας είναι πολύ πιο αποδοτικές. Για παράδειγμα ο απαιτούμενος χρόνος που παρουσιάζεται στην [1] είναι 28.3 ns, ενώ αυτός που παρουσιάζεται στην [2] είναι 23.38 ns. Και στις δυο περιπτώσεις είναι φανερό ότι η παρούσα υλοποίηση μειώνει το χρόνο κατά ένα παράγοντα μεταξύ 2 και 3.

Στο Παράρτημα που ακολουθεί βρίσκεται λεπτομερής ανάλυση όλων όσων αναφέρθηκαν παραπάνω.

Appendix A

1 Introduction

1.1 Video Format and Encoding

Digital video is a representation of a natural (real-world) visual scene, sampled spatially and temporally. A scene is sampled at a point in time to produce a frame (a representation of the complete visual scene at that point in time) or a field (consisting of odd- or even-numbered lines of spatial samples). Sampling is repeated at intervals (e.g. 1/25 or 1/30 second intervals) to produce a moving video signal. Three sets of samples (components) are typically required to represent a scene in color.

1.1.1 Video Format

A typical ‘real world’ or ‘natural’ video scene is composed of multiple objects each with their own characteristic shape, depth, texture and illumination. The color and brightness of a natural video scene changes with varying degrees of smoothness throughout the scene (‘continuous tone’). Characteristics of a typical natural video scene that are relevant for video processing and compression include spatial and temporal characteristics.

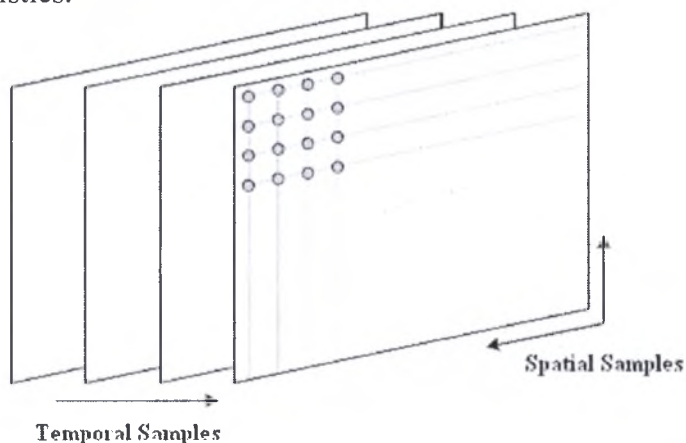


Figure 1.1: Video signal structure.

A natural visual scene is spatially and temporally continuous. Representing a visual scene in digital form involves sampling the real scene spatially (usually on a rectangular grid in the video image plane) and temporally (as a series of still frames or components of frames sampled at regular intervals in time). Digital video is the representation of a sampled video scene in digital form. Each spatial-temporal sample (picture element or pixel) is represented as a number or set of numbers that describes the brightness (luminance) and color of the sample.

Generation of video signal

To obtain a 2D sampled image, a camera focuses a 2D projection of the video scene onto a sensor, such as an array of Charge Coupled Devices (CCD array). In the

case of color image capture, each color component is separately filtered and projected onto a CCD array.

The output of a CCD array is an analogue video signal, a varying electrical signal that represents a video image. Sampling the signal at a point in time produces a sampled image or frame that has defined values at a set of sampling points. The most common format for a sampled image is a rectangle with the sampling points positioned on a square or rectangular grid. The visual quality of the image is influenced by the number of sampling points. Choosing a 'coarse' sampling grid produces a low-resolution sampled image, whilst increasing the number of sampling points slightly increases the resolution of the sampled image.

A moving video image is captured by taking a rectangular 'snapshot' of the signal at periodic time intervals. Playing back the series of frames produces the appearance of motion. A higher temporal sampling rate (frame rate) gives apparently smoother motion in the video scene but requires more samples to be captured and stored. Frame rates below 10 frames per second are sometimes used for very low bit-rate video communications (because the amount of data is relatively small) but motion is clearly jerky and unnatural at this rate. Between 10 and 20 frames per second is more typical for low bit-rate video communications; the image is smoother but jerky motion may be visible in fast-moving parts of the sequence. Sampling at 25 or 30 complete frames per second is standard for television pictures (with interlacing to improve the appearance of motion); 50 or 60 frames per second produces smooth apparent motion (at the expense of a very high data rate).

A video signal may be sampled as a series of complete frames (*progressive* sampling) or as a sequence of interlaced fields (*interlaced* sampling). In an interlaced video sequence, half of the data in a frame (one field) is sampled at each temporal sampling interval. A field consists of either the odd-numbered or even-numbered lines within a complete video frame and an interlaced video sequence contains a series of fields, each representing half of the information in a complete video frame. The advantage of this sampling method is that it is possible to send twice as many fields per second as the number of frames in an equivalent progressive sequence with the same data rate, giving the appearance of smoother motion.

Colored Video

Most digital video applications rely on the display of color video and so need a mechanism to capture and represent color information. A monochrome image requires just one number to indicate the brightness or luminance of each spatial sample. Colored images, on the other hand, require at least three numbers per pixel position to represent color accurately. The method chosen to represent brightness (luminance or luma) and color is described as a color space.

Early known color space is the RGB one. In the RGB color space, a color image sample is represented with three numbers that indicate the relative proportions of Red,

Green and Blue (the three additive primary colors of light). Any color can be created by combining red, green and blue in varying proportions. The red component consists of all the red samples, the green component contains all the green samples and the blue component contains the blue samples.

The RGB color space is well-suited to capture and display of color images. Capturing an RGB image involves filtering out the red, green and blue components of the scene and capturing each with a separate sensor array. Color Cathode Ray Tubes (CRTs) and Liquid Crystal Displays (LCDs) display an RGB image by separately illuminating the red, green and blue components of each pixel according to the intensity of each component. From a normal viewing distance, the separate components merge to give the appearance of ‘true’ color.

The human visual system (HVS) is less sensitive to color than to luminance (brightness). In the RGB color space the three colors are equally important and so are usually all stored at the same resolution but it is possible to represent a color image more efficiently by separating the luminance from the color information and representing luma with a higher resolution than color.

The YCbCr color space and its variations (sometimes referred to as YUV) is a popular way of efficiently representing color images. Y is the luminance (luma) component and can be calculated as a weighted average of R, G and B:

$$Y = k_r * R + k_g * G + k_b * B \quad (1.1)$$

Where k_r , k_g , k_b are weighting factors.

The color information can be represented as *color difference* (chrominance or chroma) components, where each chrominance component is the difference between R, G or B and the luminance Y:

$$\begin{aligned} C_b &= B - Y \\ C_r &= R - Y \\ C_g &= G - Y \end{aligned} \quad (1.2)$$

The complete description of a color image is given by Y (the luminance component) and three color differences C_b , C_r and C_g that represent the difference between the color intensity and the mean luminance of each image sample. In the YCbCr color space, only the luma (Y) and blue and red chroma (C_b , C_r) are transmitted. YCbCr has an important advantage over RGB that is the C_r and C_b components may be represented with a *lower resolution* than Y because the HVS is less sensitive to color than luminance. This reduces the amount of data required to represent the chrominance components without having an obvious effect on visual quality.

YCbCr color space representation can be sampled in various formats. 4:4:4 sampling means that the three components (Y, C_b and C_r) have the same resolution and hence a sample of each component exists at every pixel position. The numbers

indicate the relative sampling rate of each component in the *horizontal* direction, i.e. for every four luminance samples there are four C_b and four C_r samples. 4:4:4 sampling preserves the full fidelity of the chrominance components.

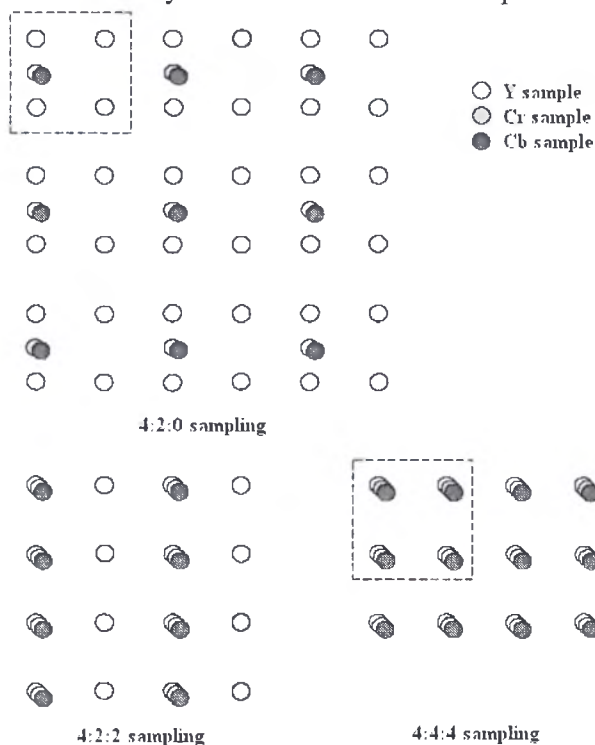


Figure 1.2: 4:2:0, 4:2:2 and 4:4:4 sampling patterns (progressive).

In 4:2:2 sampling (sometimes referred to as YUY2), the chrominance components have the same vertical resolution as the luma but half the horizontal resolution (the numbers 4:2:2 mean that for every four luminance samples in the horizontal direction there are two C_b and two C_r samples). 4:2:2 video is used for high-quality color reproduction.

In the popular 4:2:0 sampling format ('YV12'), C_b and C_r each have half the horizontal and vertical resolution of Y . The term '4:2:0' is rather confusing because the numbers do not actually have a logical interpretation and appear to have been chosen historically as a 'code' to identify this particular sampling pattern and to differentiate it from 4:4:4 and 4:2:2. 4:2:0 sampling is widely used for consumer applications such as video conferencing, digital television and digital versatile disk (DVD) storage. Because each color difference component contains one quarter of the number of samples in the Y component, 4:2:0 YCbCr video requires exactly half as many samples as 4:4:4 (or R:G:B) video.

1.1.2 Video CODEC

Video compression (video coding) is the process of compacting or condensing a digital video sequence into a smaller number of bits. 'Raw' or uncompressed digital video typically requires a large bitrate (approximately 216 Mbits for 1 second of uncompressed TV-quality video) and compression is necessary for practical storage and transmission of digital video. Compression involves a complementary pair of

systems, a compressor (encoder) and a decompressor (decoder). The encoder converts the source data into a compressed form (occupying a reduced number of bits) prior to transmission or storage and the decoder converts the compressed form back into a representation of the original video data. The encoder/decoder pair is often described as a *CODEC* (enCOder/ DECOder)

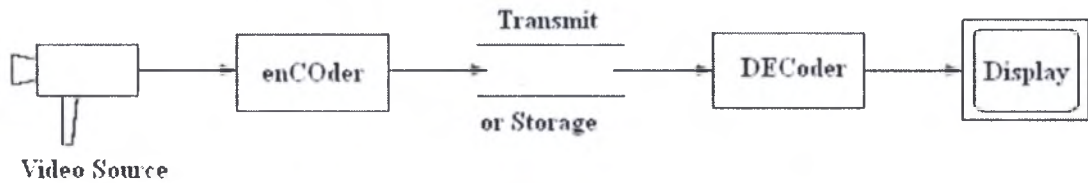


Figure1.3: Video transmission system.

A video CODEC encodes a source image or video sequence into a compressed form and decodes this to produce a copy or approximation of the source sequence. If the decoded video sequence is identical to the original, then the coding process is lossless; if the decoded sequence differs from the original, the process is lossy.

A video encoder consists of three main functional units: a *temporal model*, a *spatial model* and an *entropy encoder*.

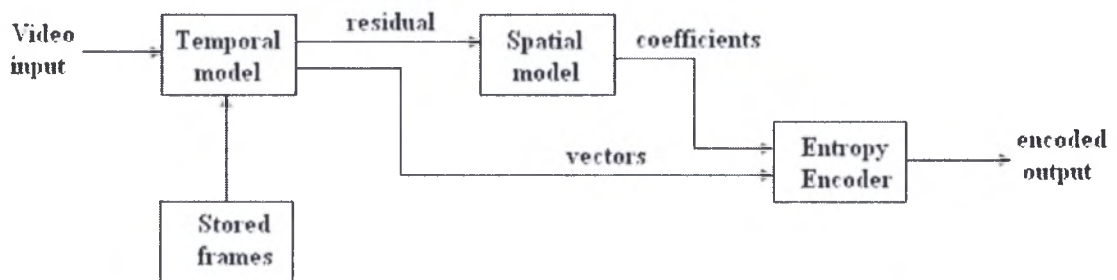


Figure1.4: Video Processing system.

The input to the temporal model is an uncompressed video sequence. The temporal model attempts to reduce temporal redundancy by exploiting the similarities between neighboring video frames, usually by constructing a prediction of the current video frame.

Motion compensation is an example of prediction encoding in which an temporal model creates a prediction of a region of the current frame based on a previous (or future) frame and subtracts this prediction from the current region to form a residual. If the prediction is successful, the energy in the residual is lower than in the original frame and the residual can be represented with fewer bits.

In a similar way, a prediction of an image sample or region may be formed from previously-transmitted samples in the same image or frame and this is called spatial prediction. Prediction encoding was used as the basis for early image compression algorithms. Spatial prediction is sometimes described as ‘Differential Pulse Code Modulation’ (DPCM), a term borrowed from a method of differentially encoding PCM samples in telecommunication systems.

The residual frame resulted from the temporal model forms the input to the spatial model which makes use of similarities between neighboring samples in the residual frame to reduce spatial redundancy. This is achieved by applying a transform to the residual samples and quantizing the results. The output of the spatial model is a set of quantised transform coefficients.

The parameters of the temporal model (typically motion vectors) and the spatial model (coefficients) are compressed by the entropy encoder. This removes statistical redundancy in the data (for example, representing commonly-occurring vectors and coefficients by short binary codes) and produces a compressed bit stream or file that may be transmitted and/or stored. A compressed sequence consists of coded motion vector parameters, coded residual coefficients and header information.

The purpose of the transform stage in spatial model is to convert motion-compensated residual data into another domain (the transform domain). The type of transform depends on a number of criteria:

1. Data in the transform domain should be decorrelated (separated into components minimal inter-dependence) and compact (most of the energy in the transformed data be concentrated into a small number of values).
2. The transform should be reversible.
3. The transform should be computationally tractable (low memory requirement, achievable using limited-precision arithmetic, low number of arithmetic operations, etc.).

Many transforms have been proposed for image and video compression and the most popular transforms tend to fall into two categories: block-based and image-based. Examples of block-based transforms include the Karhunen–Loeve Transform (KLT), Singular Value Decomposition (SVD) and the ever-popular Discrete Cosine Transform (DCT). Each of these operates on blocks of $N \times N$ image or residual samples and hence the image is processed in units of a block. Block transforms have low memory requirements and are well-suited to compression of block-based motion compensation residuals but tend to suffer from artifacts at block edges ('blockiness'). Image-based transforms operate on an entire image or frame (or a large section of the image known as a 'tile'). The most popular image transform is the Discrete Wavelet Transform (DWT or just 'wavelet'). Image transforms such as the DWT have been shown to out-perform block transforms for still image compression but they tend to have higher memory requirements (because the whole image or tile is processed as a unit) and do not 'fit' well with block-based motion compensation.

DCT

The Discrete Cosine Transform (DCT) operates on \mathbf{X} , a block of $N \times N$ samples (typically image samples or residual values after prediction) and creates \mathbf{Y} , an $N \times N$ block of coefficients. The action of the DCT (and its inverse, the IDCT) can be

described in terms of a transform matrix \mathbf{A} . The forward DCT (FDCT) of an $N \times N$ sample block is given by:

$$\mathbf{Y} = \mathbf{A}\mathbf{X}\mathbf{A}^T \quad (1.3)$$

And the inverse DCT (IDCT) by:

$$\mathbf{X} = \mathbf{A}^T\mathbf{Y}\mathbf{A} \quad (1.4)$$

Where \mathbf{X} is a matrix of samples, \mathbf{Y} is a matrix of coefficients and \mathbf{A} is an $N \times N$ transform matrix. The elements of \mathbf{A} are:

$$A_{ij} = C_i \cos \frac{(2j+1)i\pi}{2N} \quad \text{Where } C_i = \sqrt{\frac{1}{N}} \ (i=0), \quad C_i = \sqrt{\frac{2}{N}} \ (i>0)$$

The DCT transform compacts the block energy in a smaller number of coefficients rather than being spreaded in all the block coefficients. This is clear in the figure below.

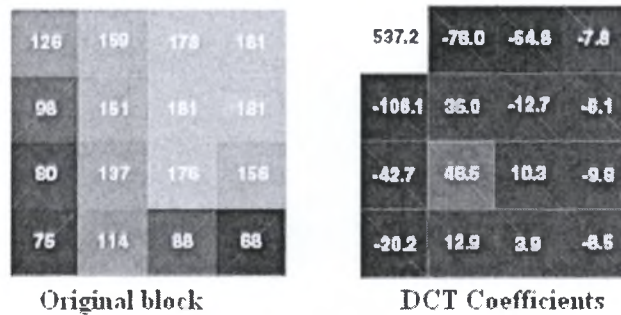


Figure 1.5: Close-up of 4x4 block; DCT coefficients.

The next step after image transformation is usually, the *quantization process*. A quantizer maps a signal with a range of values X to a quantized signal with a reduced range of values Y . It should be possible to represent the quantized signal with fewer bits than the original since the range of possible values is smaller. A *scalar quantizer* maps one sample of the input signal to one quantized output value and a *vector quantizer* maps a group of input samples (a ‘vector’) to a group of quantized values.

A simple example of scalar quantization is the process of rounding a fractional number to the nearest integer, i.e. the mapping is from R (real numbers) to Z (integer numbers). The process is lossy (not reversible) since it is not possible to determine the exact value of the original fractional number from the rounded integer.

A more general example of a uniform quantizer is:

$$FQ = \text{round} \left(\frac{X}{QP} \right) \quad (1.5)$$

$$Y = FQ * QP$$

Where QP is a quantization ‘step size’. The quantized output levels are spaced at uniform intervals of QP (as shown in the following example).

Figure 1.6 shows two examples of scalar quantizer, a linear quantizer (with a linear mapping between input and output values) and a nonlinear quantizer that has a ‘dead zone’ about zero (in which small-valued inputs are mapped to zero).

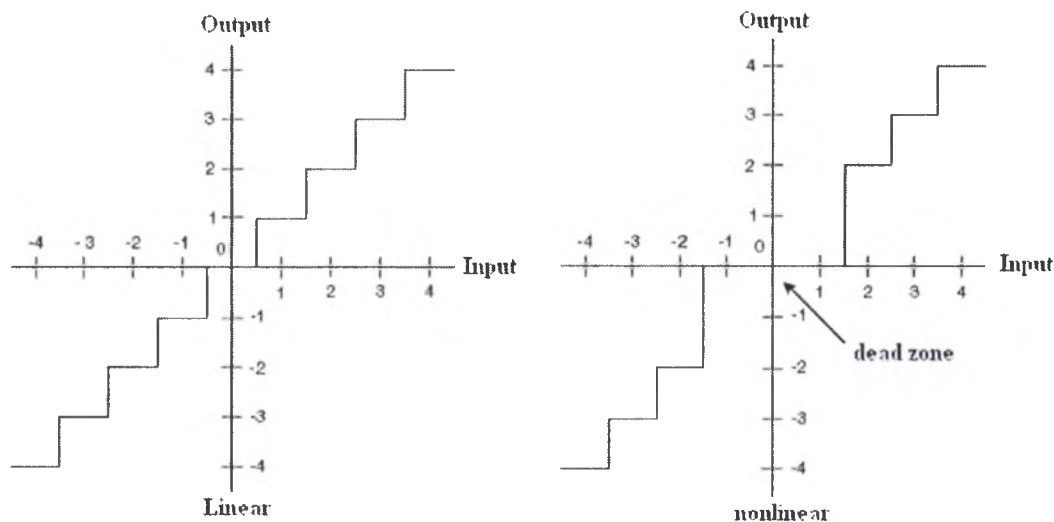


Figure 1.6: Scalar quantizer: linear; nonlinear with dead zone.

Quantization may be used to reduce the precision of image data after applying a transform such as the DCT or wavelet transform removing insignificant values such as near-zero DCT or wavelet coefficients. The forward quantizer in an image or video encoder is designed to map insignificant coefficient values to zero whilst retaining a reduced number of significant, nonzero coefficients. The output of a forward quantizer is typically a ‘sparse’ array of quantized coefficients, mainly containing zeros.

A vector quantizer maps a set of input data (such as a block of image samples) to a single value (codeword) and, at the decoder, each codeword maps to an approximation to the original set of input data (a ‘vector’). The set of vectors are stored at the encoder and decoder in a codebook.

A typical application of vector quantization in image compression is as follows:

1. Part the original image into regions (e.g. $M \times N$ pixel blocks).
2. Select a vector from the codebook that matches the current region as closely as possible.
3. Transmit an index that identifies the chosen vector to the decoder.
4. At the decoder, reconstruct an approximate copy of the region using the selected vector.

The last step before transmitting video is reordering and entropy encoding. Quantized transform coefficients are required to be encoded as compactly as possible prior to storage and transmission. In a transform-based image or video encoder, the output of the quantizer is a sparse array containing a few nonzero coefficients and a large number of zero-valued coefficients. Reordering (to group together nonzero coefficients) and efficient representation of zero coefficients are applied prior to entropy encoding. Reordering process is a scanning operation performed on the quantized transform output blocks. One of the widely used scanning techniques is Zigzag scan order applied for DCT transformed data blocks as shown in figure 1.7.

the H.26L proposal and convert it into an international standard (H.264/MPEG-4 Part 10) published by both ISO/IEC and ITU-T.

The Video Coding Experts Group is a working group of the International Telecommunication Union Telecommunication Standardization Sector (ITU-T). ITU-T develops standards (or ‘Recommendations’) for telecommunication field.

1.2 H.264 Video Encoding Standard

H.264 Standard was developed by the Moving Picture Experts Group and the Video Coding Experts Group (MPEG and VCEG) to provide better compression of video images. The H.264 is entitled ‘Advanced Video Coding’ (AVC) and is published jointly as Part 10 of MPEG-4 and ITU-T Recommendation H.264.

1.2.1 H.264 Structure

The standard includes the following three sets of capabilities, which are referred to as *profiles*, targeting specific classes of applications:

- **Baseline Profile (BP):** Primarily for lower-cost applications with limited computing resources, this profile is used widely in videoconferencing and mobile applications.
- **Main Profile (MP):** Originally intended as the mainstream consumer profile for broadcast and storage applications, the importance of this profile faded when the High profile was developed for those applications.
- **Extended Profile (XP):** Intended as the streaming video profile, this profile has relatively high compression capability and some extra tricks for robustness to data losses and server stream switching.

H.264 supports coding and decoding of 4:2:0 progressive or interlaced video. In the default sampling format, chroma (Cb and Cr) samples are aligned horizontally with every 2nd luma sample and are located vertically between two luma samples. An interlaced frame consists of two fields (a top field and a bottom field) separated in time and with the default sampling format.

A coded picture consists of a number of *macroblocks*, each containing 16×16 luma samples and associated chroma samples (8×8 Cb and 8×8 Cr samples in the current standard). Within each picture, macroblocks are arranged in *slices*, where a slice is a set of macroblocks in raster scan order. An *I slice* may contain only *I* macroblock types, a *P slice* may contain *P* and *I* macroblock types, and a *B slice* may contain *B* and *I* macroblock types.

1.2.2 H.264 CODEC

H.264 standard defines the CODEC structure shown in figure1.8. The H.264 CODEC can be divided into three main parts as in figure1.4: a temporal model, a spatial model and an entropy encoder.

The temporal model includes two dataflow paths, a ‘forward’ path (left to right) and a ‘reconstruction’ path (right to left). An input frame or field F_n is processed in units of a macroblock. Each macroblock is encoded in intra or inter mode and, for each block in the macroblock, a prediction PRED (marked ‘P’) is formed based on reconstructed picture samples. In Intra mode, PRED is formed from samples in the current slice that have been previously encoded, decoded and reconstructed (uF_n in the figures; note that *unfiltered* samples are used to form PRED). In Inter mode, PRED is formed by motion-compensated prediction from one or more reference picture(s).

The reconstruction path in the temporal model, decodes (reconstructs) encoded blocks to provide a reference for further predictions. The coefficients X are scaled (Q^{-1}) and inverse transformed (T^{-1}) to produce a difference block D_n . The prediction block PRED is added to D_n to create a reconstructed block uF_n (a decoded version of the original block; u indicates that it is unfiltered). A filter is applied to reduce the effects of blocking distortion and the reconstructed reference picture is created from a series of blocks F_n .

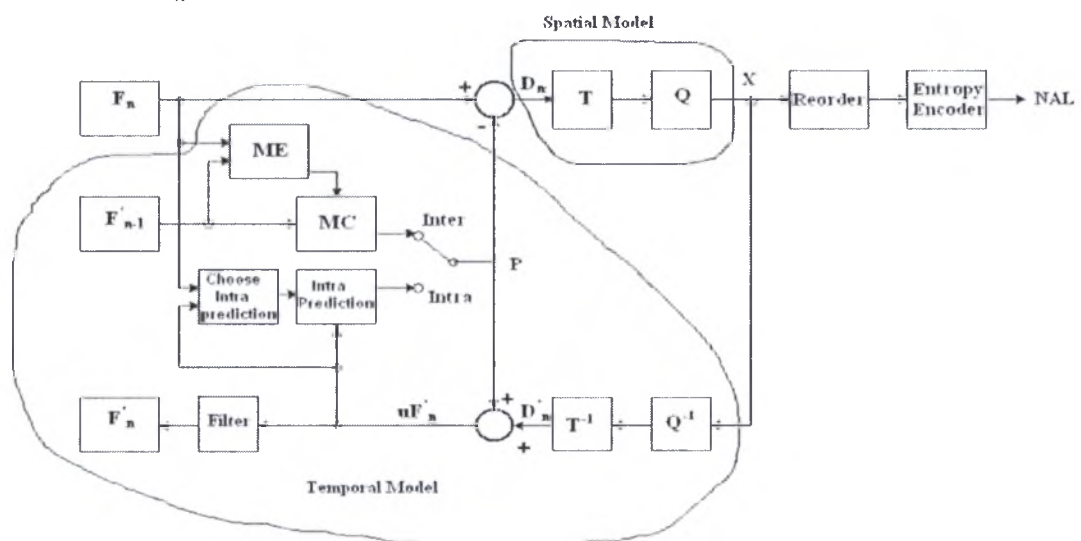


Figure 1.8: H.264 CODEC.

The prediction PRED is subtracted from the current block to produce a residual (difference) block D_n that enters the spatial model. In spatial model, D_n blocks are transformed (using a block transform) and quantised to give X , a set of quantized transform coefficients which are reordered and entropy encoded. The entropy-encoded coefficients, together with side information required to decode each block within the macroblock (prediction modes, quantizer parameter, motion vector information, etc.) form the compressed bitstream which is passed to a Network Abstraction Layer (NAL) for transmission or storage.

Two prediction types are used in the H.264 standard; Inter prediction and Intra prediction modes.

Inter prediction

Inter prediction creates a prediction model from one or more previously encoded video frames or fields using block-based motion compensation. Important differences from earlier standards include the support for a range of block sizes (from 16×16 down to 4×4) and fine subsample motion vectors (quarter-sample resolution in the luma component).

The luminance component of each macroblock (16×16 samples) may be split up in four ways and motion compensated either as one 16×16 *macroblock partition*, two 16×8 partitions, two 8×16 partitions or four 8×8 partitions. If the 8×8 mode is chosen, each of the four 8×8 sub-macroblocks within the macroblock may be split in other 4 ways, either as one 8×8 sub-macroblock partition, two 8×4 sub-macroblock partitions, two 4×8 sub-macroblock partitions or four 4×4 sub-macroblock partitions. These partitions and sub-macroblocks give to a large number of possible combinations within each macroblock. This method of partitioning macroblocks into motion compensated sub-blocks of varying size is known as variable block size or *tree structured motion compensation*.

A separate motion vector is required for each partition or sub-macroblock. Each motion vector must be coded and transmitted and the choice of partition(s) must be encoded in the compressed bitstream. Choosing a large partition size (16×16 , 16×8 , 8×16) means that a small number of bits are required to signal the choice of motion vector(s) and the type of partition but the motion compensated residual may contain a significant amount of energy in frame areas with high detail. Choosing a small partition size (8×4 , 4×4 , etc.) may give a lower-energy residual after motion compensation but requires a larger number of bits to signal the motion vectors and choice of partition(s). The choice of partition size therefore has a significant impact on compression performance. In general, a large partition size is appropriate for homogeneous areas of the frame and a small partition size may be beneficial for detailed areas.

Each chroma component in a macroblock (Cb and Cr) has half the horizontal and vertical resolution of the luminance (luma) component. Each chroma block is partitioned in the same way as the luma component, except that the partition sizes have exactly half the horizontal and vertical resolution (an 8×16 partition in luma corresponds to a 4×8 partition in chroma; an 8×4 partition in luma corresponds to 4×2 in chroma and so on). The horizontal and vertical components of each motion vector (one per partition) are halved when applied to the chroma blocks.

Each partition or sub-macroblock partition in an inter-coded macroblock is predicted from an area of the same size in a reference picture. The offset between the two areas (the motion vector) has quarter-sample resolution for the luma component and one-eighth-sample resolution for the chroma components. The luma and chroma

samples at sub-sample positions do not exist in the reference picture and so it is necessary to create them using interpolation from nearby coded samples.

Intra Prediction

In intra mode a prediction block P is formed based on previously encoded and reconstructed blocks of the same frame and is subtracted from the current block prior to encoding. For the luma samples, P is formed for each 4×4 block or for a 16×16 macroblock. There are a total of nine optional prediction modes for each 4×4 luma block, four modes for a 16×16 luma block and four modes for the chroma components. The encoder typically selects the prediction mode for each block that minimizes the difference between P and the block to be encoded.

The spatial model of H.264 consists of the transform and quantization successive operations. Section 1.3 describes in details the transformation and quantization used in H.264

Table 1.1: MPEG-4 VISUAL and H.264 standards comparison.

Comparison	MPEG-4 Visual	H.264
Supported data types	Rectangular video frames and fields, arbitrary-shaped video objects, still texture and sprites, synthetic or synthetic–natural hybrid video objects, 2D and 3D mesh objects	Rectangular video frames and fields
Number of profiles	19	3
Compression efficiency	Medium	High
Support for video streaming	Scalable coding	Switching slices
Motion compensation minimum block size	8×8	4×4
Motion vector accuracy	Half or quarter-pixel	Quarter-pixel
Transform	8×8 DCT	4 × 4 DCT approximation
Built-in deblocking filter	No	Yes

H.264 has a narrower scope than MPEG-4 Visual and is designed primarily to support efficient and robust coding and transport of rectangular video frames. Its original aim was to provide similar functionality to earlier standards such as H.263+ and MPEG-4 Visual (Simple Profile) but with significantly better compression performance and improved support for reliable transmission.

Target applications include two-way video communication (videoconferencing or videotelephony), coding for broadcast and high quality video and video streaming over packet networks. Support for robust transmission over networks is built in and the standard is designed to facilitate implementation on as wide a range of processor platforms as possible.

Table 1.1 summarizes some of the main differences among the MPEG-4 VISUAL and H.264 standards.

1.3 Transformation and Quantization of H.264 Standard

1.3.1 Transformation

The transformation process applied on predicted residual data includes three different types depending on data components and prediction type as follow:

1. A DCT-based transform for all other 4×4 blocks in the residual data.
2. A Hadamard transform for the 4×4 array of luma DC coefficients in intra macro blocks predicted in 16×16 mode.
3. A Hadamard transform for the 2×2 array of chroma DC coefficients (in any macro block).

1.3.1.1 4×4 Residual Transform

This transform operates on 4×4 blocks of residual data after motion-compensated prediction or Intra prediction. This transform is based on the DCT but with some differences; mainly, it can be implemented by integer addition and shifting, while multiplication is integrated in the quantization process which reduces the number of multiplication units needed.

Let's see how the derivation of this transform from the DCT is:

The 4×4 DCT is given by:

$$Y = AXA^T = \begin{bmatrix} a & a & a & a \\ b & c & -c & -b \\ a & -a & -a & a \\ c & -b & b & -c \end{bmatrix} * \begin{bmatrix} X \end{bmatrix} * \begin{bmatrix} a & b & a & c \\ a & c & -a & -b \\ a & -c & -a & b \\ a & -b & a & -c \end{bmatrix} \quad (1.6)$$

Where

$$a = \frac{1}{2}, \quad b = \sqrt{\frac{1}{2}} \cos\left(\frac{\pi}{8}\right), \quad c = \sqrt{\frac{1}{2}} \cos\left(\frac{3\pi}{8}\right) \quad (1.7)$$

This matrix multiplication can be factorized to the following equivalent form

$$Y = (CXC^T) \otimes E = \left(\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & d & -d & -1 \\ 1 & -1 & -1 & 1 \\ d & -1 & 1 & -d \end{bmatrix} * \begin{bmatrix} X \end{bmatrix} * \begin{bmatrix} 1 & 1 & 1 & d \\ 1 & d & -1 & -1 \\ 1 & -d & -1 & 1 \\ 1 & -1 & 1 & -d \end{bmatrix} \right) \otimes \begin{bmatrix} a^2 & ab & a^2 & ab \\ ab & b^2 & ab & b^2 \\ a^2 & ab & a^2 & ab \\ ab & b^2 & ab & b^2 \end{bmatrix} \quad (1.8)$$

CXC^T is a 'core' 2D transform. E is a matrix of scaling factors and the symbol \otimes indicates that each element of (CXC^T) is multiplied by the scaling factor in the same position in matrix E (scalar multiplication rather than matrix multiplication). The constants a and b are as before, and $d = c/b$ (approximately 0.414).

To simplify the implementation of the transform, d is approximated by 0.5. In order to ensure that the transform remains orthogonal, b also needs to be modified so that:

$$a = \frac{1}{2}, \quad b = \sqrt{\frac{2}{5}}, \quad d = \frac{1}{2} \quad (1.9)$$

The 2nd and 4th rows of matrix \mathbf{C} and the 2nd and 4th columns of matrix \mathbf{C}^T are scaled by a factor of two and the post-scaling matrix \mathbf{E} is scaled down to compensate, avoiding multiplications by half in the ‘core’ transform \mathbf{CXC}^T which could result in loss of accuracy using integer arithmetic. The final forward transform becomes:

$$Y = (\mathbf{C}_f \mathbf{X} \mathbf{C}_f^T) \otimes E_f = \left(\begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & -2 \\ 1 & -1 & -1 & 1 \\ 1 & -2 & 2 & -1 \end{bmatrix} * \begin{bmatrix} X \end{bmatrix} * \begin{bmatrix} 1 & 2 & 1 & 1 \\ 1 & 1 & -1 & -2 \\ 1 & -1 & -1 & 2 \\ 1 & -2 & 1 & -1 \end{bmatrix} \right) \otimes \begin{bmatrix} a^2 & \frac{ab}{2} & a^2 & \frac{ab}{2} \\ \frac{ab}{2} & \frac{b^2}{4} & \frac{ab}{2} & \frac{b^2}{4} \\ \frac{ab}{2} & \frac{ab}{4} & a^2 & \frac{ab}{4} \\ \frac{ab}{2} & \frac{b^2}{4} & \frac{ab}{2} & \frac{b^2}{4} \end{bmatrix} \quad (1.10)$$

This transform is an approximation to the 4×4 DCT but because of the change to factors d and b , the result of the new transform will not be identical to the 4×4 DCT.

The inverse transform is given below:

$$Y = \mathbf{C}_i^T (\mathbf{Z} \otimes \mathbf{E}_i) \mathbf{C}_i = \begin{bmatrix} 1 & 1 & 1 & \frac{1}{2} \\ 1 & \frac{1}{2} & -1 & -1 \\ 1 & -\frac{1}{2} & -1 & 1 \\ 1 & -1 & 1 & -\frac{1}{2} \end{bmatrix} * \left(\begin{bmatrix} Z \end{bmatrix} \otimes \begin{bmatrix} a^2 & ab & a^2 & ab \\ ab & b^2 & ab & b^2 \\ a^2 & ab & a^2 & ab \\ ab & b^2 & ab & b^2 \end{bmatrix} \right) * \begin{bmatrix} 1 & \frac{1}{2} & -\frac{1}{2} & 1 \\ 1 & \frac{1}{2} & -\frac{1}{2} & -1 \\ \frac{1}{2} & -1 & -1 & 1 \\ \frac{1}{2} & -1 & 1 & -\frac{1}{2} \end{bmatrix} \quad (1.11)$$

This time, \mathbf{Y} is *pre-scaled* by multiplying each coefficient by the appropriate weighting factor from matrix \mathbf{E}_i . Note the factors $\pm 1/2$ in the matrices \mathbf{C}_i and \mathbf{C}_i^T which can be implemented by a right-shift without a significant loss of accuracy because the coefficients \mathbf{Y} are pre-scaled.

Example: Compare the output of the 4 × 4 approximate transform with the output of a ‘true’ 4 × 4 DCT, for input block \mathbf{X} :

$$\mathbf{X} = \begin{bmatrix} 7 & 13 & 10 & 11 \\ 8 & 4 & 6 & 19 \\ 2 & 5 & 3 & 11 \\ 22 & 16 & 1 & 18 \end{bmatrix}$$

DCT output

$$\mathbf{Y} = \mathbf{A} \mathbf{X} \mathbf{A}^T = \begin{bmatrix} 39.000 & 1.32900 & 10.000 & -8.5870 \\ 2.6350 & -4.5668 & -6.161 & 2.23240 \\ 10.000 & 9.51100 & -1.000 & -3.1670 \\ -7.392 & -0.4762 & -7.712 & 0.18430 \end{bmatrix}$$

Approximate transform output:

$$\mathbf{Y}' = (\mathbf{CXC}^T) \otimes E_f = \begin{bmatrix} 39.000 & -3.4785 & 10.000 & -5.6580 \\ -2.5298 & -6.4000 & -6.957 & 3.8000 \\ 10.0000 & 9.1706 & -1.000 & -2.5298 \\ -7.5895 & -1.2000 & -8.2219 & 0.4000 \end{bmatrix}$$

The difference between the two output blocks is

$$Y - Y' = \begin{bmatrix} 0 & 4.8075 & 0 & 0.2674 \\ 5.1648 & 1.8332 & 0.7960 & -1.5676 \\ 0 & 0.3404 & 0 & -0.6372 \\ 0.1975 & 0.7238 & 0.5099 & -0.2157 \end{bmatrix}$$

1.3.1.2 4×4 Luma DC Coefficient Transform (16×16 Intra-mode only)

If the macro block is encoded in 16×16 Intra prediction mode (i.e. the entire 16×16 luma component is predicted from neighboring samples), each 4×4 residual block is first transformed using the ‘core’ transform described above ($C_f X C_f^T$). The DC coefficient of each 4×4 block is then transformed again using a 4×4 Hadamard transform:

$$Y_D = \left(\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} * \begin{bmatrix} W_D \end{bmatrix} * \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} \right) / 2 \quad (1.12)$$

W_D is the block of the 4 × 4 DC coefficients and Y_D is the block after transformation.

At the decoder, an inverse Hadamard transform is applied *followed by* rescaling (note that the order is not reversed as might be expected):

$$W_D = \left(\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} * \begin{bmatrix} Z_D \end{bmatrix} * \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} \right) \quad (1.13)$$

1.3.1.3 2×2 Chroma DC Coefficient Transform

Each 4×4 block in the chroma components is transformed as described in Section 1.3.1.1, then the DC coefficients of each 4×4 block of chroma coefficients are grouped in a 2×2 block (W_D) and are further transformed prior to quantization:

$$W_{QD} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} * \begin{bmatrix} W_D \end{bmatrix} * \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \quad (1.14)$$

As with the Intra luma DC coefficients, the extra transform helps to de-correlate the 2×2 chroma DC coefficients and improves compression performance.

1.3.2 Quantization

As H.264 standard support different transformations, a corresponding quantization process for each type exists. However the quantization process is completed one level, and is performed on the final result of the transformation process, using different process for each of the transformation categories described in the previous section.

1.3.2.1 4×4 Residual Quantization

All the 4×4 Residual data blocks which do not undergo another transformation type after DCT based one, will be quantized as follow:

$$Z_{ij} = \text{round}\left(\frac{Y_{ij}}{Qstep}\right) \quad (1.15)$$

Where Y_{ij} is a coefficient of the transform described above, $Qstep$ is a quantizer step size and Z_{ij} is a quantized coefficient. The rounding operation here (and throughout this section) need not round to the nearest integer; for example, biasing the ‘round’ operation towards smaller integers can give perceptual quality improvements.

A total of 52 values of $Qstep$ are supported by the standard, indexed by a Quantization Parameter, QP (Table 1.2). $Qstep$ doubles in size for every increment of 6 in QP . The wide range of quantizer step sizes makes it possible for an encoder to control the tradeoff between bit rate and quality accurately and flexibly. The values of QP can be different for luma and chroma. Both parameters are in the range 0–51 and the default is that the chroma parameter.

Table1.2: QP values and corresponding $Qsteps$.

QP	0	1	2	3	4	5	6	7	8	9	10	11	12	...
$Qstep$	0.625	0.6875	0.8125	0.875	1	1.125	1.25	1.375	1.625	1.75	2	2.25	2.5	...
QP	...	18	...	24	...	30	...	36	...	42	...	48	...	51
$Qstep$...	5	...	10	...	20	...	40	...	80	...	160	...	224

The post-scaling factor a^2 , $ab/2$ or $b^2/4$, is incorporated into the forward quantizer. First, the input block \mathbf{X} is transformed to give a block of unscaled coefficients $\mathbf{W}=\mathbf{CXC}^T$. Then, each coefficient W_{ij} is quantized and scaled in a single operation:

$$Z_{ij} = \text{round}\left(w_{ij} \frac{PF}{Qstep}\right) \quad (1.16)$$

PF is a^2 , $ab/2$ or $b^2/4$ depending on the position (i, j) as in the table1.3 below:

Table1.3: PF values

Position	PF
(0,0), (2,0), (0,2) or (2,2)	a^2
(1,1), (1,3), (3,1) or (3,3)	$b^2/4$
Others	$ab/2$

In order to simplify the arithmetic, the factor $(PF/Qstep)$ is implemented as a multiplication by a factor MF and a right-shift, avoiding any division operations:

$$Z_{ij} = \text{round}\left(w_{ij} \frac{MF}{2^{qbits}}\right) \quad (1.17)$$

Where

$$\frac{MF}{2^{qbits}} = \frac{PF}{Qstep} \quad (1.18)$$

And

$$qbits = 15 + \text{floor}\left(\frac{QP}{6}\right) \quad (1.19)$$

In integer arithmetic, Equation 1.12 can be implemented as follows:

$$\begin{aligned} (Z_{ij}| &= (W_{ij}|MF + f) \gg qbits \\ sign(Z_{ij}) &= sign(W_{ij}) \end{aligned} \quad (1.20)$$

Where \gg indicates a binary shift right, and, f is $\frac{2^{qbits}}{3}$ for Intra blocks or $\frac{2^{qbits}}{6}$ for Inter blocks. The values are given in table 1.4 below:

Table1.4: MF multiplication factor.

QP	Positions (0,0), (2,0), (0,2) or (2,2)	Positions (1,1), (1,3), (3,1) or (3,3)	Other positions
0	13107	5243	8066
1	11916	4660	7490
2	10082	4194	6554
3	9362	3647	5825
4	8192	3355	5243
5	7282	2893	4559

For QP values ≥ 6 , we find the result of $(QP \bmod 6)$ which will be in the range (0–5) and look in the corresponding row in the table 1.4 to get MF values.

The rescaling or inverse quantization process is applied by the following equation:

$$\hat{Y}_{ij} = Z_{ij} * Qstep \quad (1.21)$$

The pre-scaling factor for the inverse transform is incorporated in this operation, together with a constant scaling factor of 64 to avoid rounding errors:

$$\hat{W}_{ij} = Z_{ij} * Qstep * PF * 64 \quad (1.22)$$

\hat{W}_{ij} Is a scaled coefficient which is transformed by the inverse transforming core $C^T_i W C_i$ (Equation 1.6). The values at the output of the inverse transform are divided by 64 to remove the scaling factor (this can be implemented using only an addition and a right shift).

The H.264 standard does not specify Qstep or PF directly. Instead, the parameter $V=(Qstep.PF.64)$ is defined for $0 \leq QP \leq 5$ and for each coefficient position so the scaling operation becomes

$$\hat{W}_{ij} = Z_{ij} * V_{ij} * 2^{\text{floor}(QP/6)} \quad (1.23)$$

The values of V defined in the standard for $0 \leq QP \leq 5$ are shown in Table1.5

Table1.5: V scaling factor.

QP	Positions (0,0), (2,0), (0,2) or (2,2)	Positions (1,1), (1,3), (3,1) or (3,3)	Other positions
0	10	16	13
1	11	18	14
2	13	20	16
3	14	23	18
4	16	25	20
5	18	26	23

1.3.2.2 4×4 Luma DC Coefficient Quantization

What was described in the previous section for the 4×4 Luma DC Coefficients with a simple modification

$$\begin{aligned} \left(Z_{D(i,j)} \right) &= \left(Y_{D(i,j)} \cdot MF_{(0,0)} + 2f \right) \gg (qbits + 1) \\ \text{sign}(Z_{D(i,j)}) &= \text{sign}(Y_{D(i,j)}) \end{aligned} \quad (1.24)$$

$MF_{(0,0)}$ is the multiplication factor for position(0,0) in Table1.2 and f , $qbits$ are defined as before.

The inverse quantization operation is

$$\begin{aligned} \widehat{W}_{D(i,j)} &= W_{QD(i,j)} * V_{(0,0)} * 2^{\text{floor}(QP/6)-2} & QP \geq 12 \\ \widehat{W}_{D(i,j)} &= \left(W_{QD(i,j)} * V_{(0,0)} + 2^{1-\text{floor}(QP/6)} \right) \gg (2 - \text{floor}(QP/6)) & QP < 12 \end{aligned} \quad (1.25)$$

$V_{(0,0)}$ is the scaling factor V for position (0,0) in Table1.3. Because $V_{(0,0)}$ is constant throughout the block, rescaling and inverse transformation can be applied in any order. The specified order (inverse transform first, then scaling) is designed to maximize the dynamic range of the inverse transform.

The rescaled DC coefficients \widehat{W}_D are inserted into their respective 4×4 blocks and each 4×4 blocks of coefficients is inverse transformed using the DCT-based inverse transform core ($C_i^T \widehat{W} C_i$). In a 16×16 intra-coded macro block, much of the energy is concentrated in the DC coefficients of each 4×4 block which tend to be highly correlated. After this extra transform, the energy is concentrated further into a small number of significant coefficients.

1.3.2.3 2×2 Chroma DC Coefficient Quantization

The Quantization operation is performed by

$$\begin{aligned} \left(Z_{D(i,j)} \right) &= \left(Y_{D(i,j)} \cdot MF_{(0,0)} + 2f \right) \gg (qbits + 1) \\ \text{sign}(Z_{D(i,j)}) &= \text{sign}(Y_{D(i,j)}) \end{aligned} \quad (1.26)$$

The rescaling process is done before inverse transform core according to the equations below:

$$\begin{aligned} \widehat{W}_{D(i,j)} &= W_{QD(i,j)} * V_{(0,0)} * 2^{\text{floor}(QP/6)-1} & QP \geq 6 \\ \widehat{W}_{D(i,j)} &= \left(W_{QD(i,j)} * V_{(0,0)} \right) \gg 1 & QP < 6 \end{aligned} \quad (1.27)$$

1.3.3 Reforming Quantization process

H.264 standard suggests equations (1.20), (1.24) and (1.26) as an implementation of the quantization process as stated in equation (1.13). It is clear from these equations that the quantization process is performed as follow:

1. Take the absolute value of \mathbf{W} matrix elements (W_{ij}).
2. Multiply by the MF factors.
3. Add f value which is necessary for correct rounding.
4. Applying right shift with q bits digits.
5. Resign the elements to obtain the correct values of \mathbf{Z} matrix.

The absolute and resigning stages add an overhead on the quantization process, which can be avoided with clever selection of f values according to the sign of rounded value.

1.3.3.1 Integer division implementation

The output of integer division process is an integer number rounded from a floating point result. In this section we will deal with division on numbers of the form 2^n .

Division with integer number of the form 2^n can be implemented simply by right shift operation with n digits. The right shift process always works as *floor* function (i.e. round to the nearest smallest integer), see the example below:

Example:

$(7 / 4) \rightarrow$ shift right (111) by 2 digits \rightarrow result (001) \rightarrow (1).

$(7 / 4) \rightarrow 1\frac{3}{4} \rightarrow 1.75$

In this example we see that the $\frac{3}{4}$ are ignored and the result is rounded to 1. This is done because by right shifting 2 bits; we discard those bits which represent the value on the numerator (which is for our example (11) \rightarrow 3).

So, how we can round to the nearest largest integer if we want? As for example when we want to round numbers which their decimal digit is larger than or equal to (0.5)?

A simple way is to multiply the divisor by $(1 - 0.5)$ and add the result to the number before performing the right shift operation. Why this way? See below

$$1 - 0.5 = 0.5 \rightarrow 0.5 * 4 = 2 \text{ (10)} \rightarrow 7 + 2 = 9 \text{ (1001)}$$

$$\text{Shr (1001) by 2 digits} \rightarrow 0010 \rightarrow 2$$

Rounding occurred here because by adding 2 the floating point result is: $2\frac{1}{4}$ which is

larger than 2 and by shifting, the $(1/4)$ will be ignored.

Generally, if we want to round to the nearest largest integer for numbers which their decimal digit is larger than or equal to 0.x we just replace 0.5 above by 0.x and go forward. Adding this value will result a floating point number which is equal to or greater than the nearest integer, if the decimal digit is larger than or equal to 0.x, so when performing shifting we will go down to the new nearest smallest integer which is actually the nearest largest integer we want.

$$\frac{N}{2^k} = n.x \quad (\text{division is implemented as below})$$

For Positive Integers

$$(\text{floor}((1-0.x) * 2^k) + N) \gg k$$

What happens in the case of negative numbers?

Example:

$$(-5/4) \rightarrow \text{Shift right (1011) by 2 bits} \rightarrow (1110) \rightarrow -2$$

$$(-7/4) \rightarrow \text{Shift right (1001) by 2 bits} \rightarrow (1110) \rightarrow -2$$

$$(-5/4) \rightarrow -1\frac{1}{4} \rightarrow -1.25$$

$$(-7/4) \rightarrow -1\frac{3}{4} \rightarrow -1.75$$

The rounding here also produces the nearest smallest integer (-2 less than -1), even for very small decimal digits.

Now what we should do if we want to round value such -1.25 to -1 and not to -2, or in other words; how can we make rounding to the nearest largest integer, for example, when the decimal digit is smaller or equal to (0.5).

There are two ways:

1. Take the absolute value, work on it as for positive integers, and finally do resigning.
2. Multiply the divisor by (0.x), add the result to the number, and finally perform right shifting.

Let us check the two ways for (-5/4) and (-7/4):

Way#1:

$$(-5/4):$$

$$(1 - 0.5 = 0.5) \rightarrow (0.5 * 4 = 2) \rightarrow (2 + \text{abs}(-5) = 7) \rightarrow (7/4) \rightarrow \text{Shr (0111) 2bits} \rightarrow (0001) \rightarrow (1111) \rightarrow (-1).$$

$$(-7/4):$$

$$(1 - 0.5 = 0.5) \rightarrow (0.5 * 4 = 2) \rightarrow (2 + \text{abs}(-7) = 9) \rightarrow (9/4) \rightarrow \text{Shr (1001) 2bits} \rightarrow (0010) \rightarrow (1110) \rightarrow (-2).$$

Way#2:

$$(-5/4):$$

$$(0.5 * 4 = 2 \rightarrow (2 + (-5) = -3) \rightarrow (-3/4) \rightarrow \text{Shr (101) 2bits} \rightarrow (111) \rightarrow (-1).$$

$$(-7/4):$$

$$(0.5 * 4 = 2 \rightarrow (2 + (-7) = -5) \rightarrow (-5/4) \rightarrow \text{Shr (1011) 2bits} \rightarrow (1110) \rightarrow (-2).$$

The first way works as the case of positive integers. It rounds to the nearest integer according to (0.x).

The second way works as follow: by adding (0.5 * 4) to (-5) the value goes to (-3) which is less than 4 or, the floating point result is (-0.75) which by right shifting goes to (-1). However, for (-7) the value is still over 4, or the floating point result is (-

1.25) which by right shift goes to (-2). It is clear that values with decimal digit small than 0.5 are rounded to the nearest largest integer, and values with decimal digits larger than 0.5 are rounded to the nearest smallest integer.

The problem with way#2 is when the decimal digit is exactly 0.5 the result will be error. See below for (-6/4):

Way#1:

$(1 - 0.5 = 0.5) \rightarrow (0.5 * 4 = 2) \rightarrow (2 + \text{abs}(-6) = 8) \rightarrow (8/4) \rightarrow \text{Shr}(1000) \text{ 2bits} \rightarrow (0010) \rightarrow (1110) \rightarrow (-2).$

Way#2:

$(0.5 * 4 = 2) \rightarrow (2 + (-6) = -4) \rightarrow (-4/4) \rightarrow \text{Shr}(1100) \text{ 2bits} \rightarrow (1111) \rightarrow (-1).$

In general, way#2 does not work when the decimal digit is exactly (0.x). However, for our case as you will see below we will ensure that the case of 0.x will never occur.

$$\frac{N}{2^k} = n.x$$

For Negative Integers

way#1: $(\text{floor}((1 - |0.x|) * 2^k) + |N|) \gg k$

way#2: $(\text{floor}(|0.x| * 2^k) + N) \gg k$

1.3.3.2 Modification of f for negative Numbers

The rounding process in (1.17) is implemented by adding the f value after taking the absolute values of W coefficients. Remember that f takes the values $(2^{\text{qbits}})/3$ for intra mode prediction and $(2^{\text{qbits}})/6$ for inter mode prediction. Let us consider $w_{ij} * MF_{ij}$ as a whole signed integer equal to K . This integer needs to be divided by 2^{qbits} . Performing right shift of K by $qbits$ places, will always round to the nearest smallest integer as we've seen previously. In order to see more clearly how we use some examples:

Example 1: $w = 54, MF = 1, qbits = 5, \text{Intra mode}.$

$$54 / 32 = 1 \frac{22}{32} = 1.6875$$

$$f = 2^5 / 3 = 10 \text{ (always floor for } f).$$

$$54 * 1 + 10 = 64 \rightarrow (64/32) \rightarrow \text{Shr}(1000000) \text{ 5bits} \rightarrow (0000010) \rightarrow 2.$$

Example 2: $w = 53, MF = 1, qbits = 5, \text{Intra mode}.$

$$53 / 32 = 1 \frac{21}{32} = 1.65625$$

$$f = 2^5 / 3 = 10 \text{ (always floor for } f).$$

$$53 * 1 + 10 = 63 \rightarrow (63/32) \rightarrow \text{Shr}(0111111) \text{ 5bits} \rightarrow (0000001) \rightarrow 1.$$

The addition of f before right shifting allows us to round to the nearest largest integer as appears in example1. If we look carefully to the value of f we could know

the decimal point above it the standard prefers to round to the nearest largest integer. Look to the back calculations of the addend value f below:

$$f = \left(\left(2^{\text{qbits}} \right) / 3 \right) = (1 - 0.x) * \left(2^{\text{qbits}} \right)$$

$$\left(2^{\text{qbits}} \right) * 0.x = \left(2^{\text{qbits}} - \frac{2^{\text{qbits}}}{3} \right) = \left(2^{\text{qbits}} \right) * \frac{2}{3}$$

So, if the decimal point is larger than or equal to 0.666 we should round to the nearest smallest integer as the standard prefer.

For the case of inter mode prediction, the value of f is $(2^{\text{qbits}}/6)$, which means we round to the nearest largest integer when the decimal digit is equal to or larger than 0.8333 (or 5/6). Let us see the example below:

Example 3: $w = 59$, $MF = 1$, $\text{qbits} = 5$, Inter mode.

$$59 / 32 = 1 \frac{27}{32} = 1.84375$$

$$f = 2^5/6 = 5 \text{ (always floor for } f\text{).}$$

$$59 * 1 + 5 = 64 \rightarrow (64/32) \rightarrow \text{Shr (1000000) 5bits} \rightarrow (000010) \rightarrow 2.$$

Example 4: $w = 58$, $MF = 1$, $\text{qbits} = 5$, Inter mode.

$$58 / 32 = 1 \frac{26}{32} = 1.8125$$

$$f = 2^5/6 = 5 \text{ (always floor for } f\text{).}$$

$$58 * 1 + 5 = 63 \rightarrow (63/32) \rightarrow \text{Shr (0111111) 5bits} \rightarrow (000001) \rightarrow 1.$$

Now let us see how the standard deals with negative numbers.

Example 5: $w = -54$, $MF = 1$, $\text{qbits} = 5$, Intra mode.

$$-54 / 32 = -1 \frac{22}{32} = -1.6875$$

$$f = 2^5/3 = 10 \text{ (always floor for } f\text{).}$$

$$\text{abs}(-54) * 1 + 10 = 64 \rightarrow (64/32) \rightarrow \text{Shr (1000000) 5bits} \rightarrow (000010) \rightarrow 2 \rightarrow -2$$

Example 6: $w = -53$, $MF = 1$, $\text{qbits} = 5$, Intra mode.

$$-53 / 32 = -1 \frac{21}{32} = -1.65625$$

$$f = 2^5/3 = 10 \text{ (always floor for } f\text{).}$$

$$\text{abs}(-53) * 1 + 10 = 63 \rightarrow (63/32) \rightarrow \text{Shr (0111111) 5bits} \rightarrow (000001) \rightarrow 1 \rightarrow -1.$$

At first, it is clear that the standard adapted the way#1 which presented in the previous section for the case of negative integers. Secondly, we can see clearly the rounding to the nearest largest integer (-1 here) is done when the decimal digit is less than 0.6666 (or 2/3) so the value of f is $(1/3)$ the divisor (2^{qbits}) . The same thing can be seen for inter mode, where rounding to the nearest largest integer occurs when the decimal digit is less than 0.8333 (or 5/6).

Example 7: $w = -59$, $MF = 1$, $\text{qbits} = 5$, Inter mode.

$$-59 / 32 = -1 \frac{27}{32} = -1.84375$$

$$f = 2^5/6 = 5 \text{ (always floor for } f\text{)}.$$

$$\text{abs}(-59) * 1 + 5 = 64 \rightarrow (64/32) \rightarrow \text{Shr}(1000000) \text{ 5bits} \rightarrow (0000010) \rightarrow 2 \rightarrow -2.$$

Example 8: $w = -58$, $MF = 1$, $qbits = 5$, Inter mode.

$$-58 / 32 = -1 \frac{26}{32} = -1.8125$$

$$f = 2^5/6 = 5 \text{ (always floor for } f\text{)}.$$

$$\text{abs}(-58) * 1 + 5 = 63 \rightarrow (63/32) \rightarrow \text{Shr}(0111111) \text{ 5bits} \rightarrow (0000001) \rightarrow 1 \rightarrow -1.$$

The change we propose here is to use way#2 for the case of negative integers, so we can remove two stages from the quantization process: absolute operation stage, and resigning stage.

Again, Way#2 works as follow:

Multiply the divisor by (0.x), then add the result to the number, and perform right shifting. Here $0.x = (2/3)$ for Intra mode, and $0.x = (5/6)$ for Inter mode.

Let us check this for our examples:

Example 9: $w = -54$, $MF = 1$, $qbits = 5$, Intra mode.

$$-54 / 32 = -1 \frac{22}{32} = -1.6875$$

$$f = 2 * (2^5/3) = 21 \text{ (2/3 of the divisor } 2^5\text{)}.$$

$$-54 * 1 + 21 = -33 \rightarrow (-33/32) \rightarrow \text{Shr}(1011111) \text{ 5bits} \rightarrow (1111110) \rightarrow -2$$

Example 10: $w = -53$, $MF = 1$, $qbits = 5$, Intra mode.

$$-53 / 32 = -1 \frac{21}{32} = -1.65625$$

$$f = 2 * (2^5/3) = 21 \text{ (2/3 of the divisor } 2^5\text{)}.$$

$$-53 * 1 + 21 = -32 \rightarrow (-32/32) \rightarrow \text{Shr}(100000) \text{ 5bits} \rightarrow (111111) \rightarrow -1.$$

Example 11: $w = -59$, $MF = 1$, $qbits = 5$, Inter mode.

$$-59 / 32 = -1 \frac{27}{32} = -1.84375$$

$$f = 5 * (2^5/6) = 26 \text{ (5/6 of the divisor } 2^5\text{)}.$$

$$-59 * 1 + 26 = -33 \rightarrow (-33/32) \rightarrow \text{Shr}(1011111) \text{ 5bits} \rightarrow (1111110) \rightarrow -2$$

Example 12: $w = -58$, $MF = 1$, $qbits = 5$, Inter mode.

$$-58 / 32 = -1 \frac{26}{32} = -1.8125$$

$$f = 5 * (2^5/6) = 26 \text{ (5/6 of the divisor } 2^5\text{)}.$$

$$-58 * 1 + 26 = -32 \rightarrow (-32/32) \rightarrow \text{Shr}(100000) \text{ 5bits} \rightarrow (111111) \rightarrow -1.$$

So from the previous discussion we can say again that: for the case of Intra mode prediction $0.x = 2/3$ while for Inter mode $0.x = 5/6$

Now, what about the problem of way#2 for decimal digits exactly equal (0.x)?

Since we deal with integer numbers and with divisor of the form 2^n , the case $0.x = 2/3$ and $0.x = 5/6$ will never occur.

Simply the values $((2/3) * 2^n)$ and $((5/6) * 2^n)$ are not integer values, and since we deal only with integer values (i.e. $w_{ij} * MF_{ij}$ is always integer number) there will be no errors. Nevertheless, if it is necessary we should take the floor result of those values.

From the above discussion, equation (1.15) can be rewritten as follow:

$$Z_{ij} = (W_{ij} \cdot MF + f) \gg qbits \quad (1.28)$$

Where f here take the following values:

1. For Positive values of W_{ij} predicted in Intra mode, f takes the value $\frac{2^{qbits}}{3}$.
2. For Negative values of W_{ij} predicted in Intra mode, f takes the value $2 * \frac{2^{qbits}}{3}$.
3. For Positive values of W_{ij} predicted in Inter mode, f takes the value $\frac{2^{qbits}}{6}$.
4. For Negative values of W_{ij} predicted in Inter mode, f takes the value $5 * \frac{2^{qbits}}{6}$.

For the case of Chroma and luma Intra16 mode data blocks, the difference in quantization process is just by adding $2f$, and right shifting with $(qbits+1)$ digits.

The same model used above can be used here but by considering the divisor as $2^{(qbits+1)}$ (since we make right shift of $(qbits+1)$), also we multiply previous f values by 2 so; $2^{(qbits)}/3 \rightarrow 2^{(qbits+1)}/3$ and $2^{(qbits)}/6 \rightarrow 2^{(qbits+1)}/6$, which means we perform rounding here exactly as previously. So equations (1.20) and (1.22) can be rewritten as follow:

$$Z_{D(i,j)} = (Y_{D(i,j)} \cdot MF_{(0,0)} + 2f) \gg (qbits + 1) \quad (1.29)$$

Where f takes the same values mentioned above.

1.4 Previous Work

Core transform-quantization chip architecture synthesis is presented in [1]. The architecture is developed to be used in high-resolution applications such as High Definition Television (HDTV) and Digital Cinema. The developed architecture is prototyped and simulated using ModelSim 5.4®. It is synthesized using Leonardo Spectrum®.

Two levels of butterfly-adder blocks are used to implement the core transform. The butterfly structure represents the best way to minimize area as it processes an entire 4×4 block at the same time, and eliminates the need for memory units or any pipeline hardware.

The synthesis result was a chip of 359 ports and with critical path of 28.3 ns, i.e. a clock frequency of 34.8MHz is used. The authors concluded that the speed of the proposed architecture speed is redundant for the current video rates; this means that taking the input serially may provide the necessary speed with less area and power, or

integrating other functions and processes on the chip will make use of the speed redundancy.

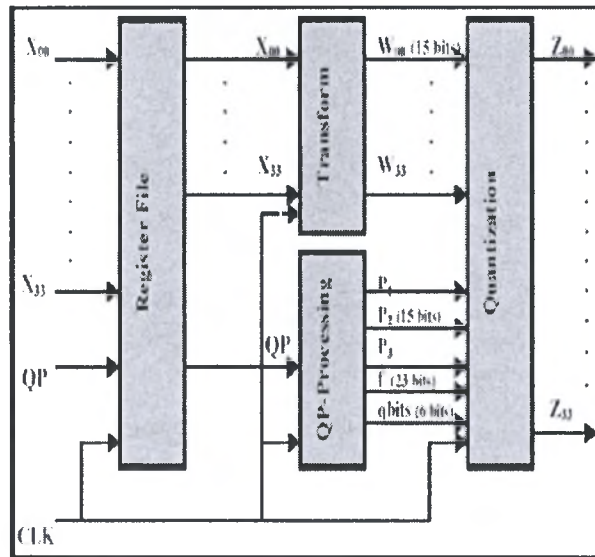


Figure 1.9: presented architecture in [1].

In [2], architecture of 2×2 hadamard transform and quantization is presented. The architecture is prototyped and simulated using ModelSim 5.4®. It is synthesized using Leonardo Spectrum®. Again the butterfly-adder block is used to build the transform on the 2×2 DC coefficients block. The proposed architecture is designed to perform pipelined operations on the 4×4 DC coefficients block coming out from the core transform process. The target technology is the FPGA device (2V3000fg676) from the Virtex-II family of Xilinx®.

The critical path is estimated be 23.68 ns, which is equivalent to a maximum operating frequency of 42.4 MHz. The chip outputs a whole 2×2 coded block with each clock pulse (except for the first block).

2 RTL Level Design of Transform and Quantization

Digital system design flow starts from a mathematical model of the system problem. The mathematical equations and representations can be partitioned into simpler expressions, which can be represented in simple functional units. This design flow is called *hierarchical design flow*. Partitioning of the mathematical model continues until representing the whole process as a structure of basic operators' functional units, (like addition, subtraction, shifting...). This level of system representation using basic operators' functional units is called *RTL level system design*. Afterwards, a second level of logic and gate design follows the RTL level. At the end, a layout and transistor level design is performed to get the fabricatable system chip.

A design on register level components (adders, multiplexers, shifters ...) is done in this chapter. All possible optimizations for delay and area are considered, with the priority of delay optimization. Power consumption optimization in this level can be considered as proportional for area and delay optimization since the above optimizations in this level are based on minimizing the logic circuits and computation levels rather than gates and wiring sizing, which will be considered in the next chapter when gate level design is considered.

2.1 Transform

- The entire transform process in H.264 can be represented in a block diagram as the one shown in figure2.1. Transform process can be implemented on a single chip or can be integrated with quantization phase.

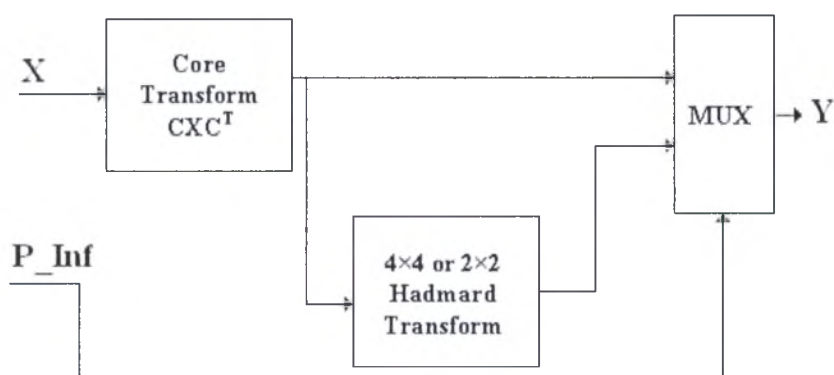


Figure2.1: Transform Stage Block Diagram

The block diagram shows that two input and one output busses are required. A detailed list of the inputs and outputs of the transform circuit is shown below:

- Input Busses:
 1. **X** buss: **X** is a 4×4 residual block data, with each entry being a 9-bits signed integer (the most left bit is the sign bit). So a bus of 144 lines is needed.
 2. **P_Inf** bus: **P_Inf** carry the information about component (chroma or luma) and Prediction types. It consists of two bits as follow:
 - Chroma component (regardless of prediction type): this case represented by the string “00” on the P_Inf bus.
 - Luma component (Intra 16×16 prediction mode only): this case represented by the string “01” on the P_Inf bus.
 - Other modes of prediction for Luma Components: this case represented by the string “11” on the P_Inf bus.
 3. Output Buss **Y**: This is a 4×4 block transformed data with each entry being a 15-bit signed integer (the most left bit is the sign bit). A buss of 240 lines is needed.

The transform process has two forward flows. The result of one of them will be taken at the output. The first flow passes through core transform CXC^T (1.5) and directly to the output through the multiplexer. The second forward flow passes from the core transform through Hadmard transform (1.12 or 1.14) and then to the output buss through the multiplexer. The selection of the taken path depends on the data block type as described by P_Inf.

2.1.1 Core Transform

The core transform process shown in equation (1.10) can be written as below:

$$W = (C_r X C_r^T) = \left(\begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & -2 \\ 1 & -1 & -1 & 1 \\ 1 & -2 & 2 & -1 \end{bmatrix} * \begin{bmatrix} x_{00} & x_{01} & x_{02} & x_{03} \\ x_{10} & x_{11} & x_{12} & x_{13} \\ x_{20} & x_{21} & x_{22} & x_{23} \\ x_{30} & x_{31} & x_{32} & x_{33} \end{bmatrix} * \begin{bmatrix} 1 & 2 & 1 & 1 \\ 1 & 1 & -1 & -2 \\ 1 & -1 & -1 & 2 \\ 1 & -2 & 1 & -1 \end{bmatrix} \right) \quad (2.1)$$

The first two matrix multiplication can be re-written in the following way:

$$\begin{bmatrix} (x_{00} + x_{10} + x_{20} + x_{30}) & (x_{01} + x_{11} + x_{21} + x_{31}) & (x_{02} + x_{12} + x_{22} + x_{32}) & (x_{03} + x_{13} + x_{23} + x_{33}) \\ (2x_{00} + x_{10} - x_{20} - 2x_{30}) & (2x_{01} + x_{11} - x_{21} - 2x_{31}) & (2x_{02} + x_{12} - x_{22} - 2x_{32}) & (2x_{03} + x_{13} - x_{23} - 2x_{33}) \\ (x_{00} - x_{10} - x_{20} + x_{30}) & (x_{01} - x_{11} - x_{21} + x_{31}) & (x_{02} - x_{12} - x_{22} + x_{32}) & (x_{03} - x_{13} - x_{23} + x_{33}) \\ (x_{00} - 2x_{10} + 2x_{20} - x_{30}) & (x_{01} - 2x_{11} + 2x_{21} - x_{31}) & (x_{02} - 2x_{12} + 2x_{22} - x_{32}) & (x_{03} - 2x_{13} + 2x_{23} - x_{33}) \end{bmatrix} \quad (2.2)$$

This can be renamed as

$$S = \begin{bmatrix} s_{00} & s_{01} & s_{02} & s_{03} \\ s_{10} & s_{11} & s_{12} & s_{13} \\ s_{20} & s_{21} & s_{22} & s_{23} \\ s_{30} & s_{31} & s_{32} & s_{33} \end{bmatrix} \quad (2.3)$$

A Butterfly block arrangement can be used to implement this first matrix multiplication as shown in the figure2.2:

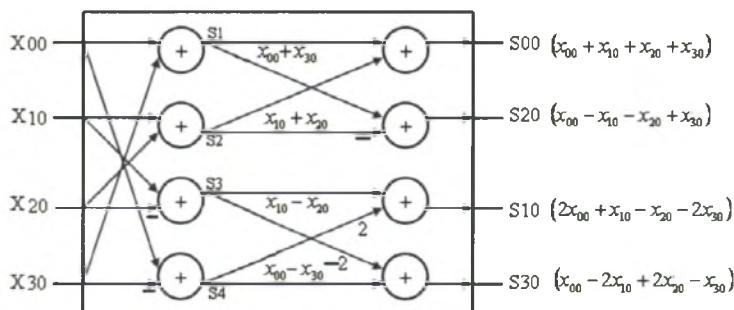


Figure 2.2: first stage of core transform butterfly.

The matrix multiplication of the result matrix S with C^T will result in the following matrix:

$$\begin{bmatrix} (s_{00} + s_{01} + s_{02} + s_{03}) & (2s_{00} + s_{01} - s_{02} - 2s_{03}) & (s_{00} - s_{01} - s_{02} + s_{03}) & (s_{00} - 2s_{01} + 2s_{02} - s_{03}) \\ (s_{10} + s_{11} + s_{12} + s_{13}) & (2s_{10} + s_{11} - s_{12} - 2s_{13}) & (s_{10} - s_{11} - s_{12} + s_{13}) & (s_{10} - 2s_{11} + 2s_{12} - s_{13}) \\ (s_{20} + s_{21} + s_{22} + s_{23}) & (2s_{20} + s_{21} - s_{22} - 2s_{23}) & (s_{20} - s_{21} - s_{22} + s_{23}) & (s_{20} - 2s_{21} + 2s_{22} - s_{23}) \\ (s_{30} + s_{31} + s_{32} + s_{33}) & (2s_{30} + s_{31} - s_{32} - 2s_{33}) & (s_{30} - s_{31} - s_{32} + s_{33}) & (s_{30} - 2s_{31} + 2s_{32} - s_{33}) \end{bmatrix} \quad (2.4)$$

Which can be also realized using the upper structure of butterfly as shown below:

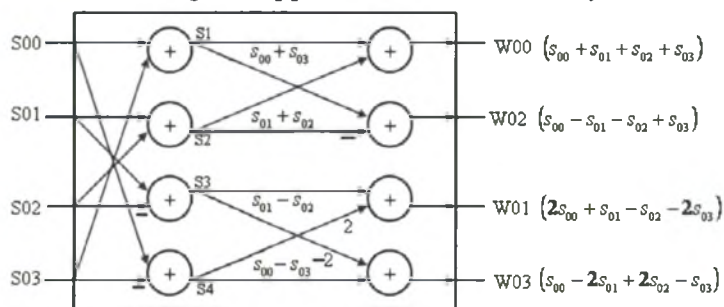


Figure 2.3: second stage core transform butterfly.

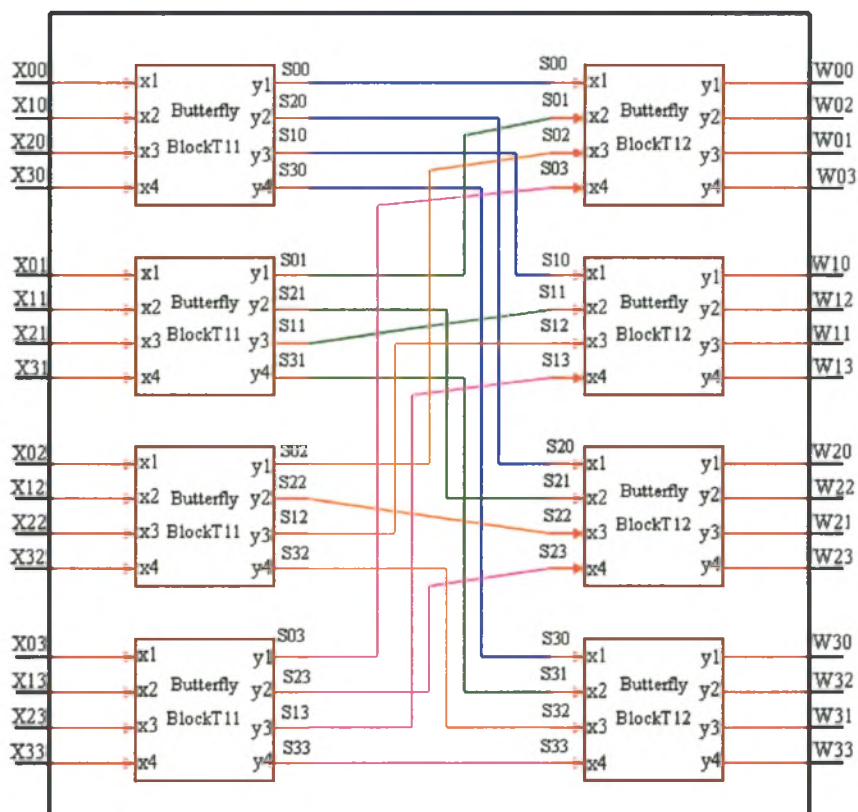


Figure 2.4: Core Transform Structure.

From the above two stages of butterflies, we can construct the core transform for 4×4 residual data blocks. Figure 2.4 shows the structure of core transform stage.

It is clear from the figures that we have two types of Butterfly; one with 9-bits per input line and second with 12-bits per input line. The multiply by 2 operation which is necessary in the butterfly block can be implemented using left shift by 1-bit operation.

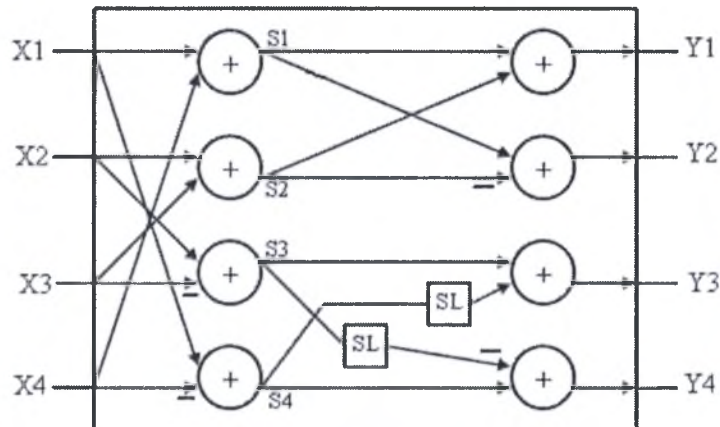


Figure 2.5: structure of butterfly block used in core transform.

Butterfly T11:

Butterfly type 1 has 4 input buses each with 9-bits, and 4 output buses each with 12-bits. So it uses four 9-bits signed adders, and four 11-bits signed adders. Furthermore, Butterfly block uses a left shift by 1-bit block to perform multiplication by 2. Left shift block takes 10-bits as input and outputs 11 bits for the 11-bits signed adder.

Notes: Lines with no SL blocks, fed the eleventh line as sign extension. Minus signs performed by a block of inverters and a feed of carry one to the adder.

Optimization Hints: it is not necessary to use 11-bits adder for the first two adders in the second level in the butterfly since there is no shift blocks on their inputs, and sign extension is done later for the 12th bit on the output bus. So, a 10-bits adder is better.

Butterfly T12:

Butterfly type 2 has 4 input buses each with 12-bits, and 4 output buses each with 15-bits. So it uses four 12-bits signed adders, and four 14-bits signed adders. As the previous one, this Butterfly block uses left shift by 1-bit block to perform multiplication by 2. Left shift block takes 13-bits as input and outputs 14 bits for the 14-bits signed adder.

Notes: Lines with no SL blocks, fed the fourteenth line as sign extension. Minus signs performed by block of inverters and a feed of carry one to the adder.

Optimization Hints: it is not necessary to use 14-bits adder for the first two adders in the second level in the butterfly since there is no shift blocks on their inputs,

and sign extension is done later for the 15th bit on the output bus. So, a 13-bits adder is better.

Estimation of Necessary Hardware:

From previous description we can estimate the physical hardware which is necessary as listed below:

1. 16 blocks of 9-bits signed adders.
2. 8 blocks of 10-bits signed adders.
3. 8 blocks of 11-bits signed adders.
4. 16 blocks of 12-bits signed adders.
5. 8 blocks of 13-bits signed adders.
6. 8 blocks of 14-bits signed adders.
7. 8 blocks of 9-bits inversion.
8. 8 blocks of 12-bits inversion.
9. 4 blocks of 10-bits inversion.
10. 4 blocks of 11-bits inversion.
11. 4 blocks of 13-bits inversion.
12. 4 blocks of 14-bits inversion.

2.1.2 Hadmard 4×4/2×2 Transform

Hadmard transform will work on DC coefficients which resulted from the core transform process.

The first matrix multiplication in equation (1.12) can be decomposed into the result below:

$$\begin{bmatrix} (w_{00} + w_{10} + w_{20} + w_{30}) & (w_{01} + w_{11} + w_{21} + w_{31}) & (w_{02} + w_{12} + w_{22} + w_{32}) & (w_{03} + w_{13} + w_{23} + w_{33}) \\ (w_{00} + w_{10} - w_{20} - w_{30}) & (w_{01} + w_{11} - w_{21} - w_{31}) & (w_{02} + w_{12} - w_{22} - w_{32}) & (w_{03} + w_{13} - w_{23} - w_{33}) \\ (w_{00} - w_{10} - w_{20} + w_{30}) & (w_{01} - w_{11} - w_{21} + w_{31}) & (w_{02} - w_{12} - w_{22} + w_{32}) & (w_{03} - w_{13} - w_{23} + w_{33}) \\ (w_{00} - w_{10} + w_{20} - w_{30}) & (w_{01} - w_{11} + w_{21} - w_{31}) & (w_{02} - w_{12} + w_{22} - w_{32}) & (w_{03} - w_{13} + w_{23} - w_{33}) \end{bmatrix} \quad (2.5)$$

This can be renamed as

$$S = \begin{bmatrix} s_{00} & s_{01} & s_{02} & s_{03} \\ s_{10} & s_{11} & s_{12} & s_{13} \\ s_{20} & s_{21} & s_{22} & s_{23} \\ s_{30} & s_{31} & s_{32} & s_{33} \end{bmatrix} \quad (2.6)$$

The same structure of butterfly used in the core transform can be used here but without multiplication by 2.

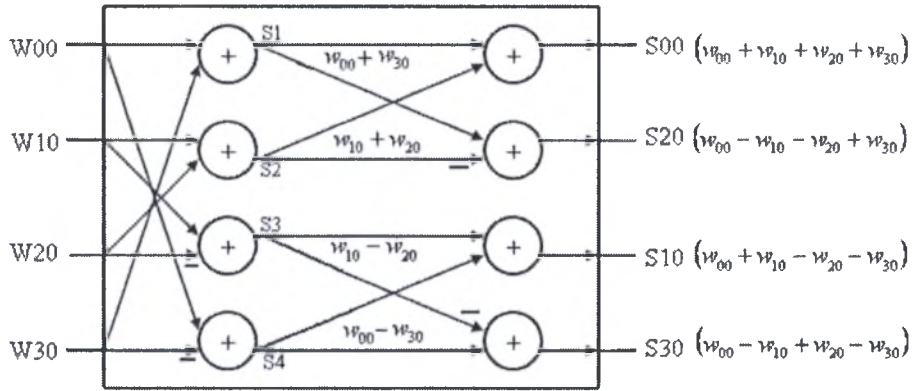


Figure2.6: first stage butterfly of 4×4 hadmard transform.

The same thing applies for the second matrix multiplication, where the second butterfly structure used in core transform is used here but without the multiplication by 2 operation.

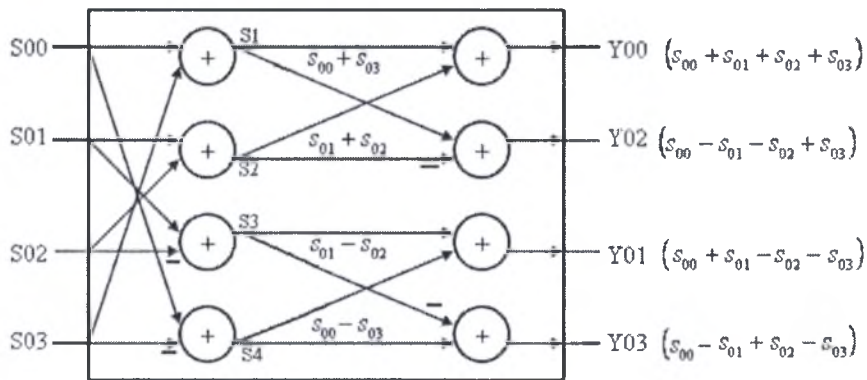


Figure2.7: second stage butterfly of 4×4 hadmard transform.

Before viewing the whole structure for the 4×4 Hadmard transform lets see how we can implement the 2×2 Hadmard transform.

The transform in equation (1.14) can be written for the whole 4×4 W matrix as follow:

$$\begin{bmatrix} y_{00} & y_{01} \\ y_{10} & y_{11} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} * \begin{bmatrix} w_{00} & w_{01} \\ w_{10} & w_{11} \end{bmatrix} * \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \quad \begin{bmatrix} y_{02} & y_{03} \\ y_{12} & y_{13} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} * \begin{bmatrix} w_{02} & w_{03} \\ w_{12} & w_{13} \end{bmatrix} * \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \quad (2.7)$$

$$\begin{bmatrix} y_{20} & y_{21} \\ y_{30} & y_{31} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} * \begin{bmatrix} w_{20} & w_{21} \\ w_{30} & w_{31} \end{bmatrix} * \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \quad \begin{bmatrix} y_{22} & y_{23} \\ y_{32} & y_{33} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} * \begin{bmatrix} w_{22} & w_{23} \\ w_{32} & w_{33} \end{bmatrix} * \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

The result of the first matrix multiplication can be seen below:

$$\begin{bmatrix} y_{00} & y_{01} \\ y_{10} & y_{11} \end{bmatrix} = \begin{bmatrix} (w_{00} + w_{10} + w_{01} + w_{11}) & (w_{00} + w_{10} - w_{01} - w_{11}) \\ (w_{00} - w_{10} + w_{01} - w_{11}) & (w_{00} - w_{10} - w_{01} + w_{11}) \end{bmatrix} \quad (2.8)$$

A similar butterfly structure as the one used for the 4×4 Hadmard transform, can be used here, with the only difference that in this case we need one stage and the inputs' order to the butterfly is different.

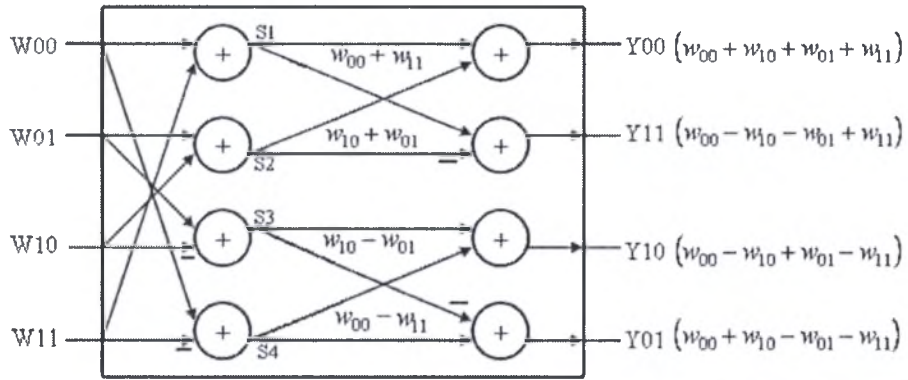


Figure 2.8: butterfly block used for 2x2 Hadamard transform.

Four blocks of this butterfly will be enough to implement the 2x2 Hadamard transform.

From the above discussion of Hadamard transform implementation, we can see that there are two different implementations of Hadamard transforms stage. This means that we should implement two separated units for each of the 4x4 and 2x2 hadamard transforms.

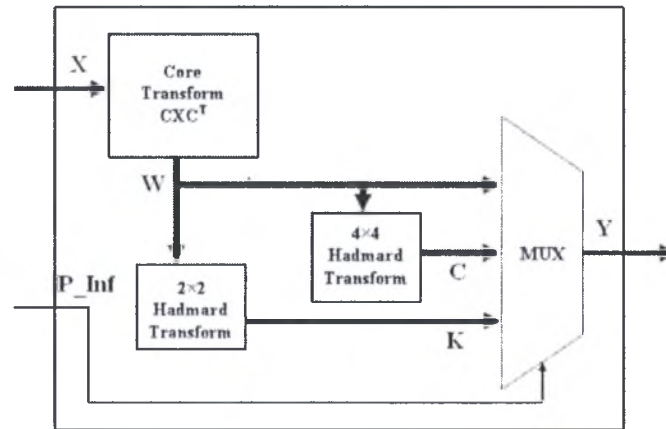


Figure 2.9: Implementation of hadamard transforms using separated 2x2 and 4x4 blocks.

The drawback of this method is the duplicated hardware which increases the design area and consumed power.

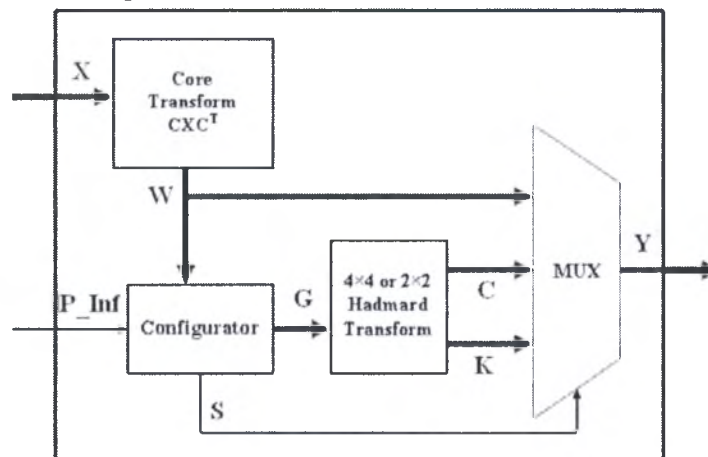


Figure 2.10: Implementation of hadamard transforms using configurator.

One of the best solutions for small area and power is to use the first butterfly level used for the 4x4 Hadamard transform to implement the 2x2 Hadamard transform. However a Configurator before the Hadamard transform level is necessary to pass the appropriate inputs according to data type (chroma or luma).

The only drawback of this implementation scheme is the delay overhead from the configurator. However, this overhead can be minimized by a good synthesis.

The figure below show the Hadamard transform stage used for the 4x4 and 2x2 Hadamard transforms.

The used butterfly blocks are described as follow:

- *Butterfly BlockT21:* It has four 15-bit input busses, and four 16-bit output busses. So we need four 15-bits signed adders, and four 16-bits signed adders.
- *Butterfly BlockT22:* It has four 16-bit input busses, and four 16-bit output busses. So we need four 16-bits signed adders, and four 17-bits signed adders.

Note: The division operation by 2 (in the 4x4 Hadamard transform) is implemented in the second level addition in butterfly block T22, where we ignore the least significant bit of the adder output and take the remaining 16 bits.

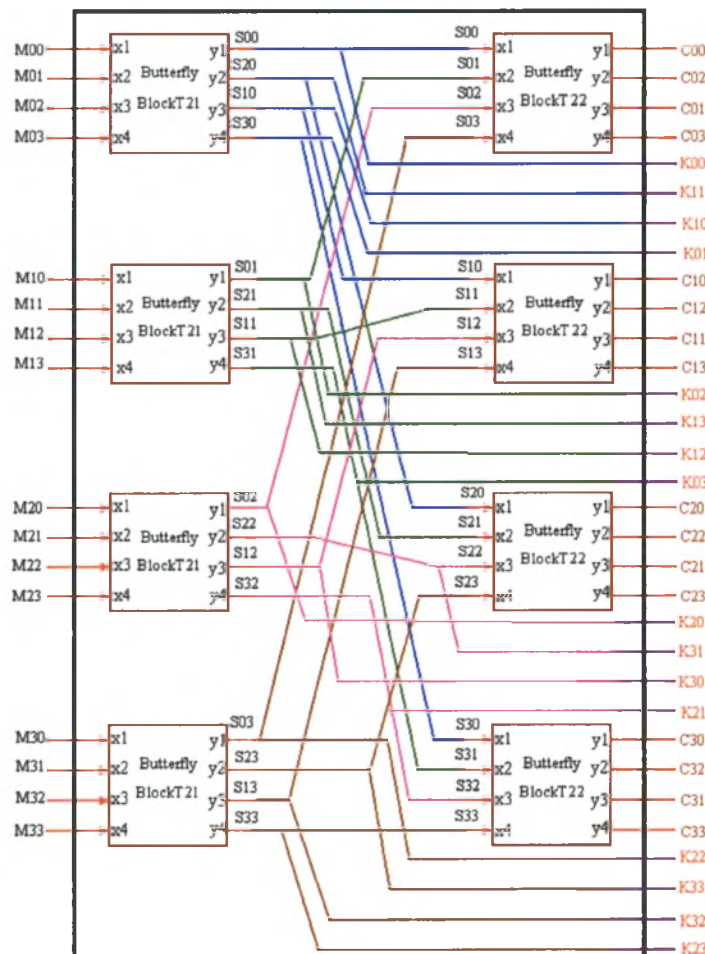


Figure2.11: 2x2 and 4x4 hadamard transforms block used with configurator.

Estimation of Necessary Hardware:

From the previous description we can estimate the physical hardware which is necessary as listed below:

1. 64 15-bits signed adders.
2. 8 blocks of 15-bits inversion.
3. 16 blocks of 16-bits inversion.
4. 8 blocks of 17-bits inversion.

2.1.3 Configurator

As already mentioned, the first level in the 4x4 Hadmard transform Block can be used to perform the 2x2 Hadmard transform of chroma components. The problem with this solution is that the input DC coefficients in the case of 4x4 Hadmard transform is different of that in the case of 2x2 Hadmard transform (see the figure below) so, a Configurator is used here to solve this problem.

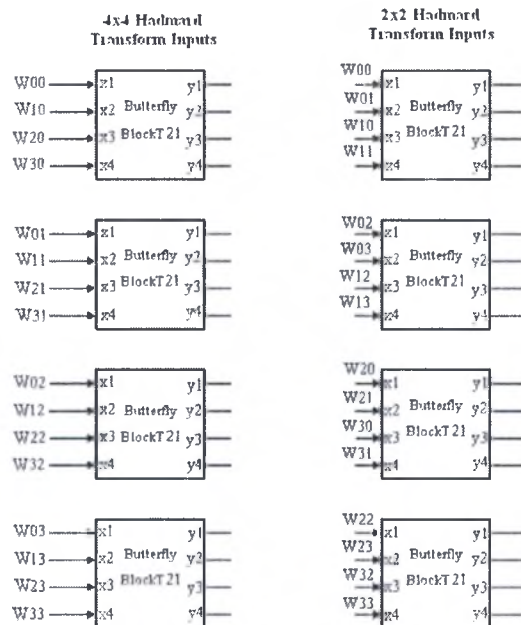


Figure2.12: Order of inputs to 2x2 Hadmard transform and 4x4 Hadmard transform

The structure of the Configurator is shown in figure2.13. It can be considered as a 2x1 Multiplexer, with the two inputs are the 4x4 DC coefficients block **W**, but each input has different order of the 16-data elements of **W**. Two levels is necessary, ANDing level, then ORing level. The total number of necessary gates is around 420 2-input AND gates, and around 210 2-input OR gates. The selection signals are generated just by three 2-input AND gates, as described at the beginning from **P_INF**, (see Inputs description of Transform stage section 2.1).

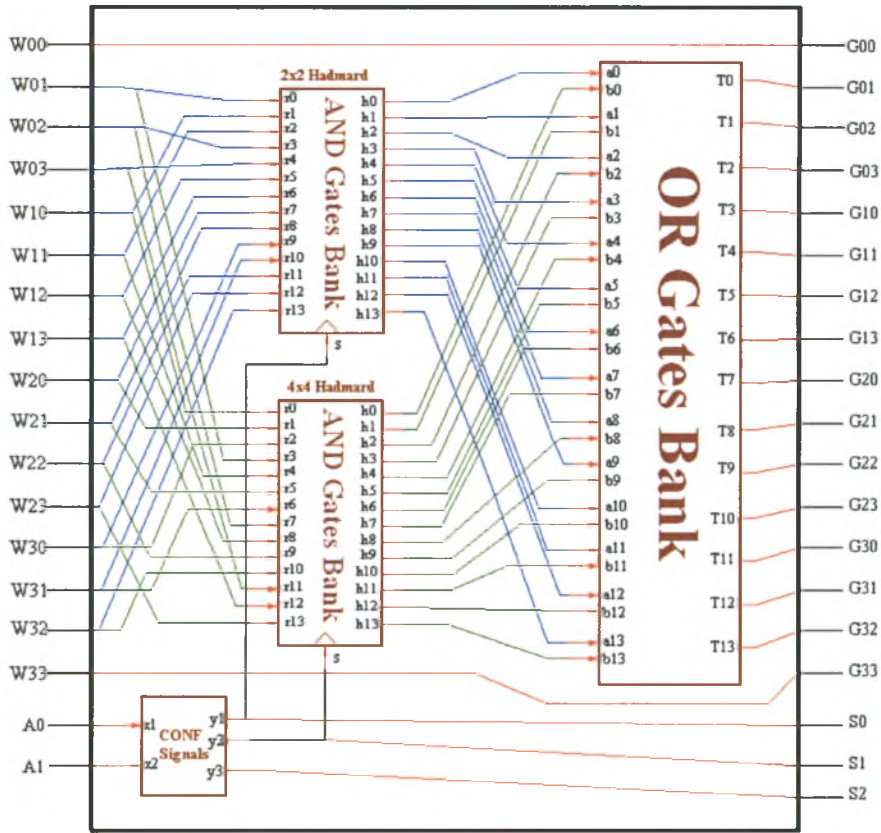


Figure 2.13: Configurator Structure.

2.1.4 Faster Implementation of 4×4 Hadamard Transform

From the above implementation, with the core transform and the 4×4 hadamard transform being sequentially, we will get 8 levels of adders on the critical path. A robust environment will enforce us to implement 4×4 hadamard transform independently from core transform output. To investigate this possibility lets go back to equation (1.11) by embedding W from equation (2.1) as below:

$$Y_D = \left(\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} * \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & -2 \\ 1 & -1 & -1 & 1 \\ 1 & -2 & 2 & -1 \end{bmatrix} * \begin{bmatrix} X \end{bmatrix} * \begin{bmatrix} 1 & 2 & 1 & 1 \\ 1 & 1 & -1 & -2 \\ 1 & -1 & -1 & 2 \\ 1 & -2 & 1 & -1 \end{bmatrix} * \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} \right) / 2 \quad (2.9)$$

This will be on the new form:

$$Y_D = \left(\begin{bmatrix} 5 & -1 & 1 & -1 \\ 1 & 5 & -1 & -1 \\ -1 & -1 & 5 & 1 \\ -1 & 1 & -1 & 5 \end{bmatrix} * \begin{bmatrix} x_{00} & x_{01} & x_{02} & x_{03} \\ x_{10} & x_{11} & x_{12} & x_{13} \\ x_{20} & x_{21} & x_{22} & x_{23} \\ x_{30} & x_{31} & x_{32} & x_{33} \end{bmatrix} * \begin{bmatrix} 5 & 1 & -1 & -1 \\ -1 & 5 & -1 & 1 \\ 1 & -1 & 5 & -1 \\ -1 & -1 & 1 & 5 \end{bmatrix} \right) / 2 \quad (2.10)$$

From the first look, the existence of 5 indicates difficulties and necessity for multiplication, since shift left process will not be enough. Nevertheless, the existence of 5 on the diagonal simplifies the implementation, and avoids multiplication or complexity. The new transformation matrices on the left and right hands of \mathbf{X} matrix can be decomposed as follow:

$$\begin{bmatrix} 5 & -1 & 1 & -1 \\ 1 & 5 & -1 & -1 \\ -1 & -1 & 5 & 1 \\ -1 & 1 & -1 & 5 \end{bmatrix} = \begin{bmatrix} 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ -1 & -1 & 1 & 1 \\ -1 & 1 & -1 & 1 \end{bmatrix} + \begin{bmatrix} 4 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 4 \end{bmatrix}$$

and (2.11)

$$\begin{bmatrix} 5 & 1 & -1 & -1 \\ -1 & 5 & -1 & 1 \\ 1 & -1 & 5 & -1 \\ -1 & -1 & 1 & 5 \end{bmatrix} = \begin{bmatrix} 1 & 1 & -1 & -1 \\ -1 & 1 & -1 & 1 \\ 1 & -1 & 1 & -1 \\ -1 & -1 & 1 & 1 \end{bmatrix} + \begin{bmatrix} 4 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 4 \end{bmatrix}$$

By inserting the decomposed forms in equation (2.10) and performing multiplications we get the following form for $2*Y_D$:

$$Y_D = \begin{bmatrix} 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ -1 & -1 & 1 & 1 \\ -1 & 1 & -1 & 1 \end{bmatrix} * \begin{bmatrix} X \\ X \\ X \\ X \end{bmatrix} + \begin{bmatrix} 1 & 1 & -1 & -1 \\ -1 & 1 & -1 & 1 \\ 1 & -1 & 1 & -1 \\ -1 & -1 & 1 & 1 \end{bmatrix} * \begin{bmatrix} X \\ X \\ X \\ X \end{bmatrix} + \begin{bmatrix} 4 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 4 \end{bmatrix} +$$
(2.12)

$$+ \begin{bmatrix} 4 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 4 \end{bmatrix} * \begin{bmatrix} X \\ X \\ X \\ X \end{bmatrix} + \begin{bmatrix} 1 & 1 & -1 & -1 \\ -1 & 1 & -1 & 1 \\ 1 & -1 & 1 & -1 \\ -1 & -1 & 1 & 1 \end{bmatrix} + \begin{bmatrix} 4 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 4 \end{bmatrix} * \begin{bmatrix} X \\ X \\ X \\ X \end{bmatrix} + \begin{bmatrix} 4 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 4 \end{bmatrix}$$

From the above decomposition we can conclude that:

1. The matrix which has only nonzero elements on the diagonal is Identity matrix which can be ignored. So, the matrices with their diagonals have values of 4 can be replaced by a scalar value of 4 only.
2. The second term on the right side is the same as the first part of the first term so, one implementation is necessary for the both.
3. Third term needs separate implementation, with similar structure to the second part of the first term.
4. The last term is simply a multiplication of input data elements by 16, which can be implemented by left shift of 4 places.

First Term

$$\begin{bmatrix} 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ -1 & -1 & 1 & 1 \\ -1 & 1 & -1 & 1 \end{bmatrix} * \begin{bmatrix} x_{00} & x_{01} & x_{02} & x_{03} \\ x_{10} & x_{11} & x_{12} & x_{13} \\ x_{20} & x_{21} & x_{22} & x_{23} \\ x_{30} & x_{31} & x_{32} & x_{33} \end{bmatrix} * \begin{bmatrix} 1 & 1 & -1 & -1 \\ -1 & 1 & -1 & 1 \\ 1 & -1 & 1 & -1 \\ -1 & -1 & 1 & 1 \end{bmatrix}$$
(2.13)

The first two matrix multiplication can be written as below:

$$\begin{bmatrix} (x_{00} - x_{10} + x_{20} - x_{30}) & (x_{01} - x_{11} + x_{21} - x_{31}) & (x_{02} - x_{12} + x_{22} - x_{32}) & (x_{03} - x_{13} + x_{23} - x_{33}) \\ (x_{00} + x_{10} - x_{20} - x_{30}) & (x_{01} + x_{11} - x_{21} - x_{31}) & (x_{02} + x_{12} - x_{22} - x_{32}) & (x_{03} + x_{13} - x_{23} - x_{33}) \\ (-x_{00} - x_{10} + x_{20} + x_{30}) & (-x_{01} - x_{11} + x_{21} + x_{31}) & (-x_{02} - x_{12} + x_{22} + x_{32}) & (-x_{03} - x_{13} + x_{23} + x_{33}) \\ (-x_{00} + x_{10} - x_{20} + x_{30}) & (-x_{01} + x_{11} - x_{21} + x_{31}) & (-x_{02} + x_{12} - x_{22} + x_{32}) & (-x_{03} + x_{13} - x_{23} + x_{33}) \end{bmatrix} \quad (2.14)$$

This can be renamed as

$$S = \begin{bmatrix} s_{00} & s_{01} & s_{02} & s_{03} \\ s_{10} & s_{11} & s_{12} & s_{13} \\ s_{20} & s_{21} & s_{22} & s_{23} \\ s_{30} & s_{31} & s_{32} & s_{33} \end{bmatrix}$$

From the matrix we note that the third row is the minus of the second row, and the fourth row is the minus of the first row, which will simplify the size of the butterfly adders block and reduce it to the half as shown in the figure below:

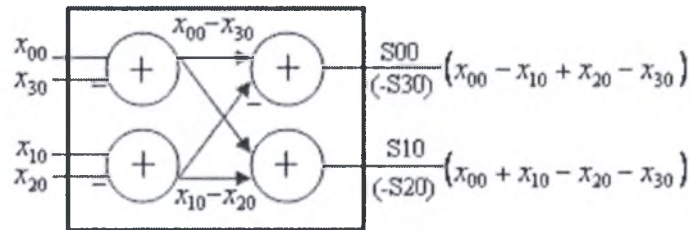


Figure2.14: Butterfly1_1N

In the same way the second matrix multiplication can be done as

$$\begin{bmatrix} (s_{00} - s_{01} + s_{02} - s_{03}) & (s_{00} + s_{01} - s_{02} - s_{03}) & (-s_{00} - s_{01} + s_{02} + s_{03}) & (-s_{00} + s_{01} - s_{02} + s_{03}) \\ (s_{10} - s_{11} + s_{12} - s_{13}) & (s_{10} + s_{11} - s_{12} - s_{13}) & (-s_{10} - s_{11} + s_{12} + s_{13}) & (-s_{10} + s_{11} - s_{12} + s_{13}) \\ (s_{20} - s_{21} + s_{22} - s_{23}) & (s_{20} + s_{21} - s_{22} - s_{23}) & (-s_{20} - s_{21} + s_{22} + s_{23}) & (-s_{20} + s_{21} - s_{22} + s_{23}) \\ (s_{30} - s_{31} + s_{32} - s_{33}) & (s_{30} + s_{31} - s_{32} - s_{33}) & (-s_{30} - s_{31} + s_{32} + s_{33}) & (-s_{30} + s_{31} - s_{32} + s_{33}) \end{bmatrix} \quad (2.15)$$

This can also be built using the same structure of butterfly adders' block as shown in figure below:

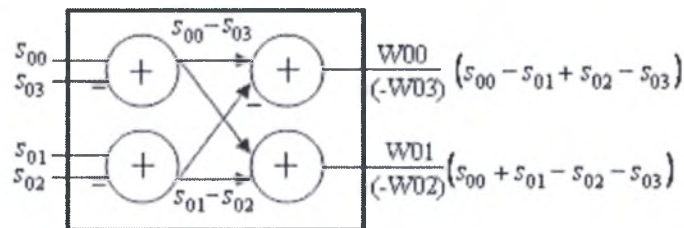


Figure2.15: Butterfly1_2N

So, the first term implementation can be done using two levels of butterflies with the structures and connections discussed above. This results to the structure shown in figure2.16

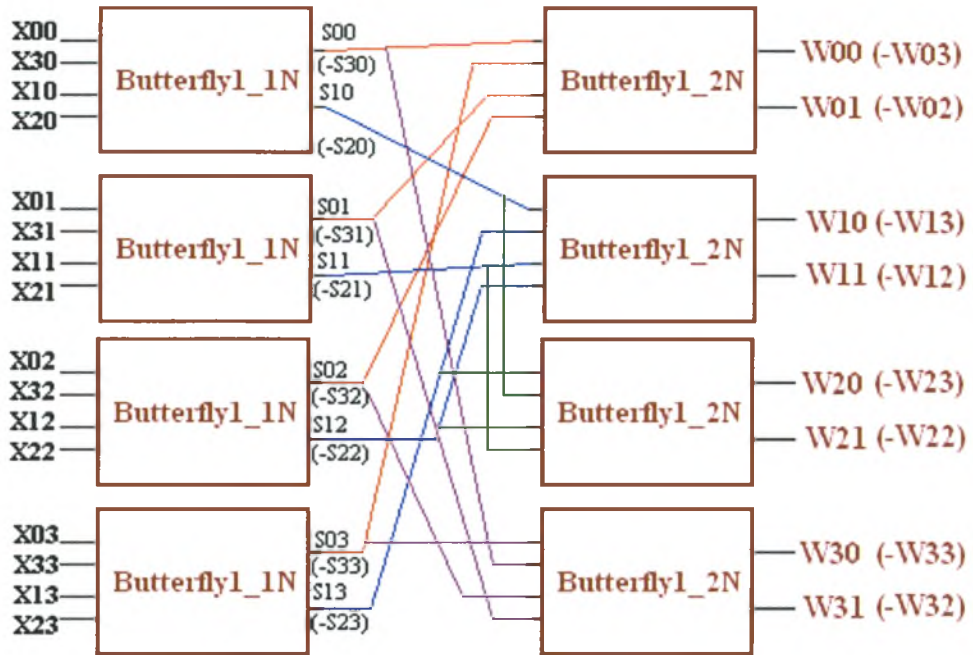


Figure2.16: First Term of equation (2.12) implementation.

Third Term

Third term can be calculated using the same structure of butterfly used described above with different inputs. The decomposition of the matrix multiplication in third term is

$$\begin{bmatrix} (x_{00} - x_{01} + x_{02} - x_{03}) & (x_{00} + x_{01} - x_{02} - x_{03}) & (-x_{00} - x_{01} + x_{02} + x_{03}) & (-x_{00} + x_{01} - x_{02} + x_{03}) \\ (x_{10} - x_{11} + x_{12} - x_{13}) & (x_{10} + x_{11} - x_{12} - x_{13}) & (-x_{10} - x_{11} + x_{12} + x_{13}) & (-x_{10} + x_{11} - x_{12} + x_{13}) \\ (x_{20} - x_{21} + x_{22} - x_{23}) & (x_{20} + x_{21} - x_{22} - x_{23}) & (-x_{20} - x_{21} + x_{22} + x_{23}) & (-x_{20} + x_{21} - x_{22} + x_{23}) \\ (x_{30} - x_{31} + x_{32} - x_{33}) & (x_{30} + x_{31} - x_{32} - x_{33}) & (-x_{30} - x_{31} + x_{32} + x_{33}) & (-x_{30} + x_{31} - x_{32} + x_{33}) \end{bmatrix} \quad (2.16)$$

This can be renamed as

$$M = \begin{bmatrix} M_{00} & M_{01} & M_{02} & M_{03} \\ M_{10} & M_{11} & M_{12} & M_{13} \\ M_{20} & M_{21} & M_{22} & M_{23} \\ M_{30} & M_{31} & M_{32} & M_{33} \end{bmatrix}$$

This can be implemented using the following butterfly structure:

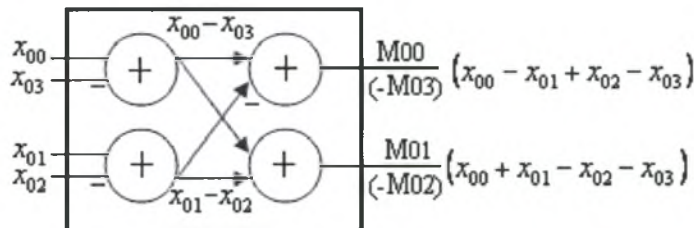


Figure2.17: Butterfly1_1N used for Third term calculation.

This stage runs in parallel with the first stage of butterflies of first term calculation.

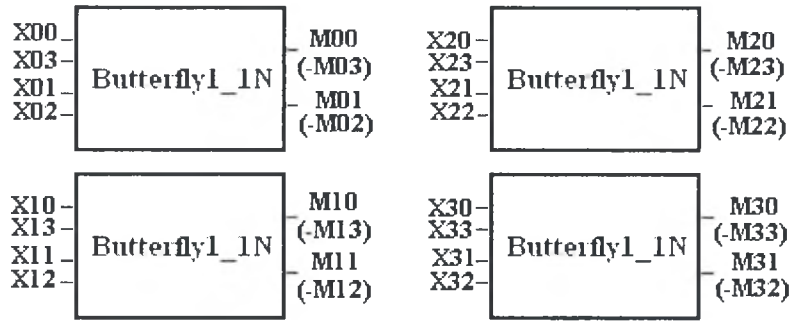


Figure2.18: Third Term in equation (2.12) implementation.

Second, Third, and Fourth Terms Addition

The previous analysis showed that the calculation of S matrix elements and M matrix elements occur in parallel with each calculation having two stages of adders. Moreover, for second term calculation what is only needed is a multiplication of matrix S by 4 (left shift by 2 places). Also, the fourth term is the input data elements multiplied by 16 (left shift by 4 places). So, the three terms are ready after the first level of butterfly blocks, and we can run a block of two addition stages to add the three terms while the second level of butterfly block, used for first term calculation is running. Figure2.19 shows how three types of blocks are used to perform addition of the last three terms:

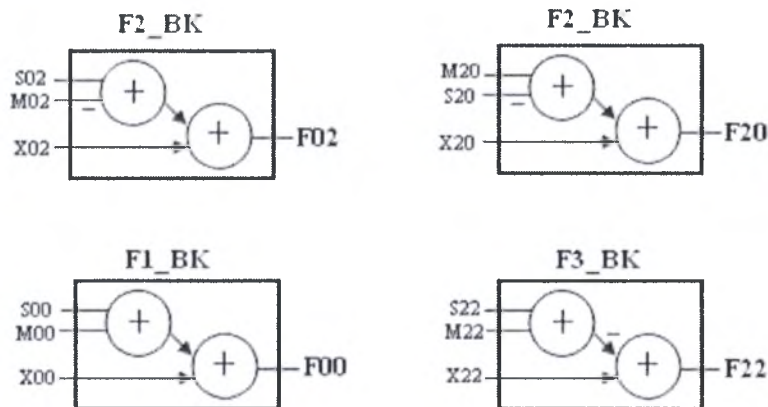


Figure2.19: Blocks structures used to add the last three terms in equation (2.12).

These three different instances are obligatory since for some elements, we have the negative value of them, as we stated above (the output S_{00} is also $-S_{30}$), so, when one of the two added M and S elements is negative we use $F2_BK$ structure. If both are negative we use $F3_BK$, else (all positive) we use the normal structure $F1_BK$. The figure gives some examples of inputs of each block type to give a hint how connections are done.

The stage of addition of the last three terms finishes simultaneously with the second level of butterfly blocks used for first term since each has two stages of adders.

The last step, before the 4×4 hadmard transform output is ready, is the addition of W (output of first term calculation) matrix with F (output of addition of last three terms) which cost a stage of adders, and a shift right one place to realize the division by 2.

In total, 5-stages of adders will result in the 4×4 hadmard transform and save three stages of adders. However the 2×2 hadmard transform needs 2 stages of adders after the core transform, which result in 6-stages of adders on the critical path. This means that 2-stages of adders of 15-bit size are saved by implementing 4×4 hadmard transform on the described structure above.

Hardware Estimation:

1. 16 9-bit signed adders.
2. 28 10-bit signed adders.
3. 24 11-bit signed adders.
4. 12 12-bit signed adders.
5. 8 13-bit signed adders.
6. 8 15-bit signed adders.
7. 16 9-bit inverting blocks.
8. 16 10-bit inverting blocks.
9. 8 11-bit inverting blocks.
10. 8 12-bit inverting blocks.
11. 8 15-bit inverting blocks.

2.2 Standard Quantization Implementation

Quantization phase can be implemented in two main structures; the first one calculates the quantization parameters which include value of f , multiplication factors MF and $qbits$, which is necessary for shift operation. The second one will implement equation (1.20) using the calculated parameters.

2.2.1 Quantization parameters

Quantization parameters include f value, multiplication factors which differ according to the location of the data element in the data block, and the $qbits$ value which decides how much shift will be done. The calculation of all these parameters depends on the value of QP (other factors will be compensated in the second structure) which may be different for each data block. Figure 2.20 shows the block diagram of the quantization parameters calculation structure.

The calculation process can be separated into three independent paths (not for the case of F value calculation).

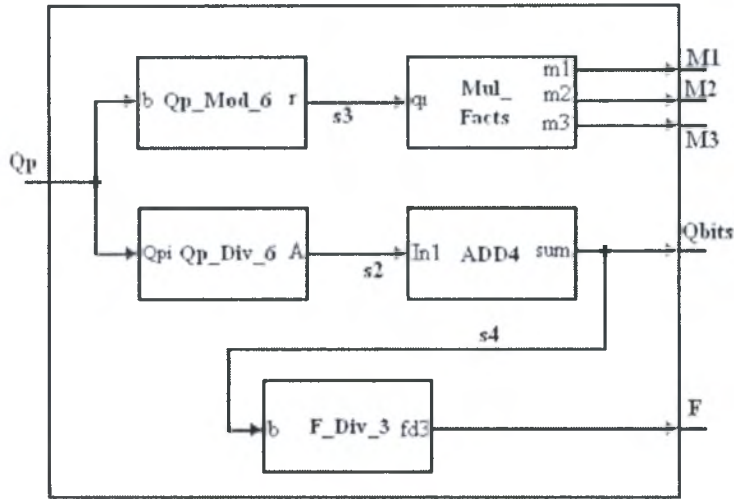


Figure2.20: Quantization parameters calculation structure.

Multiplication Factors:

The multiplication factors M1, M2, and M3 are those that appear in the three columns of table 1.2 respectively. The values in the table are calculated for QP values from 0 to 5. For greater values of QP we calculate (QP mod 6) and map the result to the table. So what is done in this structure is to calculate QP mod 6 and then use the result to generate the corresponding number on each multiplication factor buss. Each multiplication factor buss consists of 14 lines.

Note: The effective information in the multiplication factors M2, and M3 needs 12 lines-bus since M2 has 2-bits with value ‘0’ always, and M3 has 1-bit with value ‘0’ always and the second with value ‘1’.

Qbits Calculation:

Qbits value is calculated as in equation (1.19), where firstly the result of the division of QP by 6 is calculated, with floor process, and then it is increased by 15 using a simple adder.

Since maximum value of QP is 51 the max division result is 8, hence the max value of qbits is 23 which needs only 5-lines buss.

Calculation of F:

F takes the values $\frac{2^{qbits}}{3}$ for Intra blocks or $\frac{2^{qbits}}{6}$ for Inter blocks. Furthermore, it should be multiplied by 2, for the case of chroma and luma intra16x16 data blocks. However, in this stage we will calculate only the base value $\frac{2^{qbits}}{3}$, and then in the second stage (second structure of the quantization implementation) it will be changed according to the prediction type. A 22-bit buss is necessary.

2.2.2 Quantization phase

The second stage in quantization process goes in a forward flow from left to right as shown in figure2.21. The first thing to perform is the calculation of the absolute

values, then multiplication, addition, right shift, and at the end, resigning of the result to the sign of the input number.

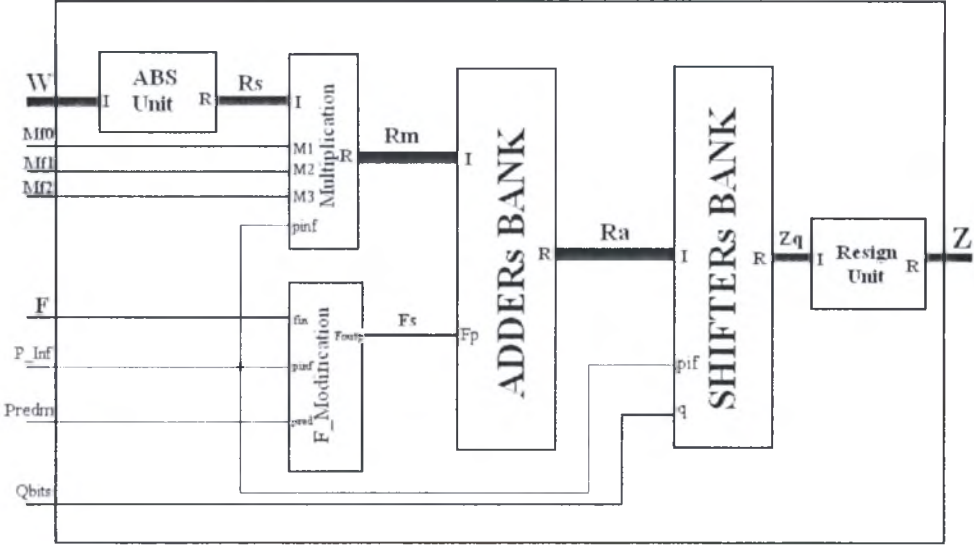


Figure2.21: Standard quantization data flow.

The details of the inputs and outputs ports of the Quantization block are described below:

- **W**: 4×4 Transformed Coefficients, with each 15-bit signed integer, with the most significant bit (bit15) is the sign bit.
- **Mf0, Mf1, and Mf2**: Multiplication factors for the three columns in table1.4 respectively. Each is represented with 14-bits positive integer.
- **P_Inf**: Prediction information as described in Transform part.
- **F**: The Addend value, represented in 22-bit positive integer.
- **Predm**: a bit that indicated prediction type, Intra (takes the value ‘0’) or Inter (takes the value ‘1’).
- **Qbits**: The shift size to be done, it is represented using 5-bits positive integer.

2.2.2.1 Absolute Value

Absolute value process can be realized simply as 2’s complement process, which can be implemented as an inverting stage followed by addition of 1. The inverting process should be controlled by the sign bit since for positive number there is no necessity to make 2’s complement.

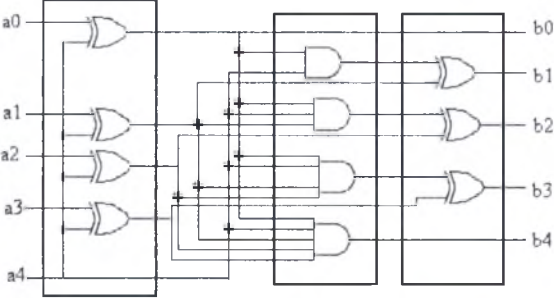


Figure2.22: absolute value implementation.

Consequently the positive numbers pass this stage without any change. Figure 2.22 shows the implementation of absolute value operation for 5-bits signed number where the MSB (a_4) is the sign number.

The first XORs block works as controlled inverter by the sign bit; when $a_4 = 1$ XOR gates work as inverters, else pass the value. The second stage represents the addition process, where the carry is calculated and passed to the second XORs block. Note here that for $a_4 = 0$ the AND gates result '0' which pass the value in the second XORs block, so positive values pass without any effect. It is clear that the critical path passes through the last AND gate, the fan-in of which increases with the increase of input size.

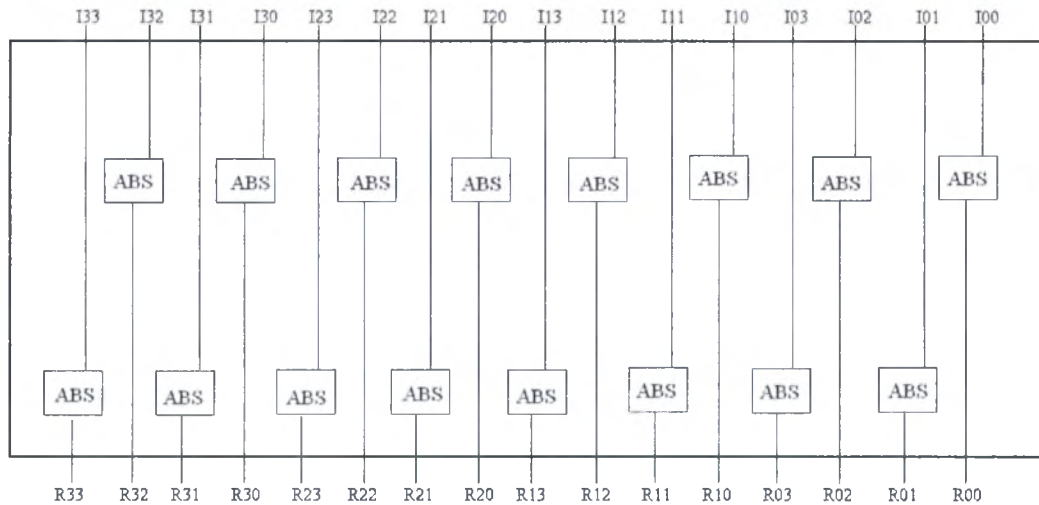


Figure 2.23: Absolute value Unit.

To perform the absolute value of the entire 4×4 transformed block, 16 absolute value units of the above structure are necessary.

2.2.2.2 Multiplication

Multiplication of 15-bit by 14-bit numbers is necessary here. A multiplexing stage is necessary to select appropriate multiplication factor according to the data element location as already mentioned.

In figure 2.24, M_1 is the multiplication factor of position (0,0). As shown we saw in equations (1.24) and (1.26) M_1 is used for all positions of the matrix W , so a total of 12 multiplexers are needed to pass M_1 if we have chroma or luma intra 16×16 components, or pass M_1, M_2, M_3 according to the element position in the data block.

A variety of multiplier architectures can be used to implement multiplication operation. In this section we will discuss three architectures of widely used multipliers.

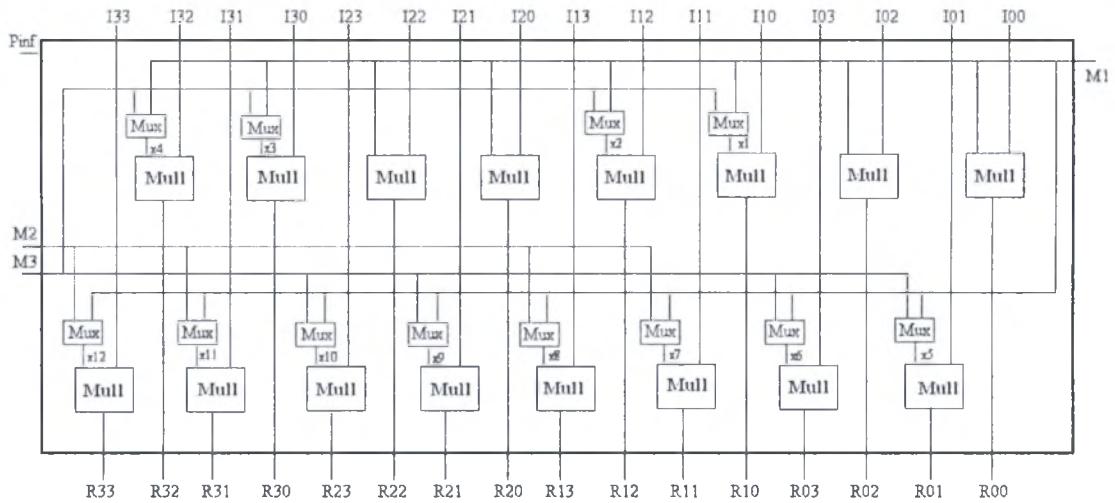


Figure2.24: Multiplication Unit structure.

Array multiplier:

The array multiplier can be represented as in the figure2.25, where a 4×15 multiplier is shown.

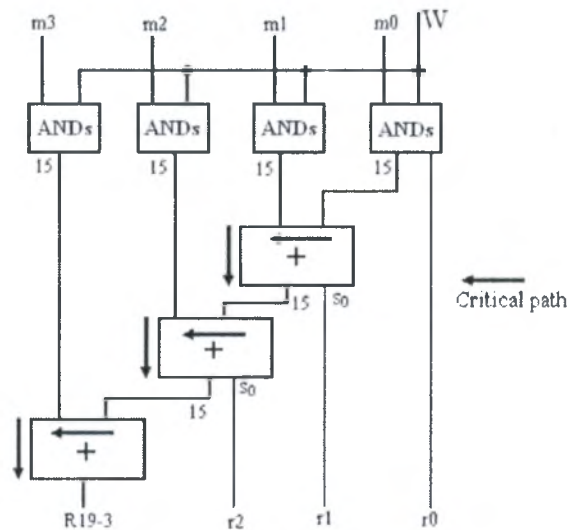


Figure2.25: Array multiplier.

The outputs of the ANDs blocks are the partial products which can be summed using 3 15-bit adders as shown in the figure.

In general, $M \times N$ array multiplier will need $(M - 1)$ stages of N -bits adder, and M blocks, each of N AND gates. Array multiplier is the slowest multiplier architecture. The critical path delay can be calculated as

$$t_{mult} = ((N - 1) + (M - 2))t_{carry} + t_{and} + (M - 1)t_{sum}$$

Tree Multiplier:

In tree multiplier, the addition of the partial products is done in parallel. Figure2.26 shows an example 4×15 multiplier.

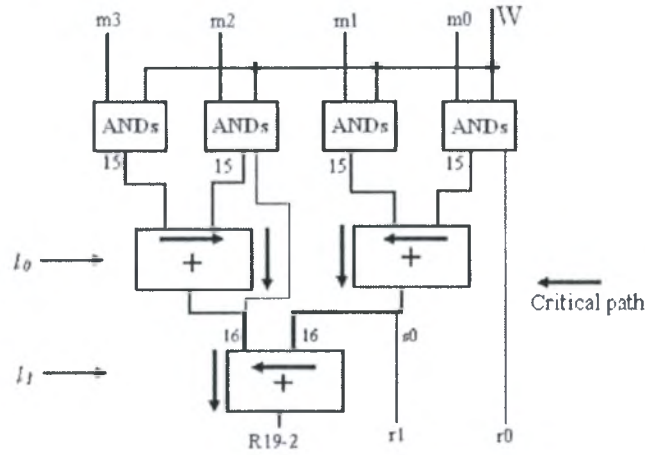


Figure 2.26: Tree Multiplier.

Tree multiplier decreases the number of levels necessary to sum the partial products from $(M - 1)$ to “ $\text{ceil}(\log_2 M)$ ”.

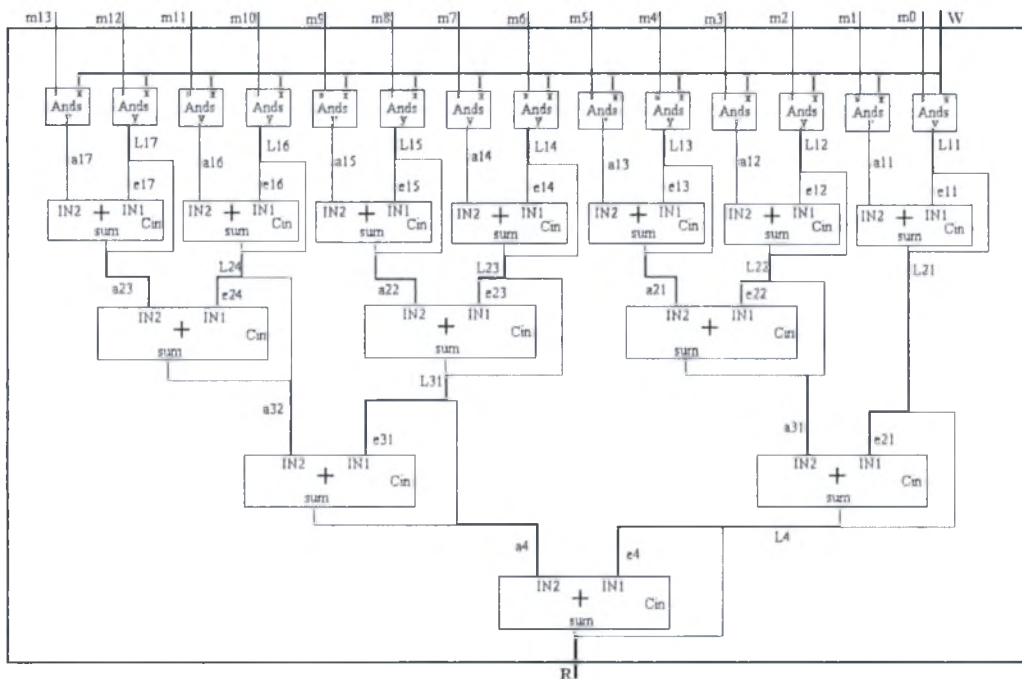


Figure 2.27: 14x15 Tree multiplier architecture.

However the size of adders increases as we go down in summation levels in the form $(N + L*2)$ where L is the level number. Despite this increase in size of the adders, the overall critical path length is smaller. The overhead in this architecture is the increase of multiplier area. Figure 2.27 shows the architecture of 14x15 multiplier.

The critical path delay of the tree multiplier can be calculated as follow:

$$t_{mult} = (3\text{ceil}(\log_2 M) + N - 3)t_{carry} + t_{and} + (\text{ceil}(\log_2 M))t_{sum}$$

Carry Save Multiplier:

Carry save multiplier has the same structure with the array multiplier, but carry-save adders are used, instead, this implementation releases the carry propagate

overhead in each addition stage. Only one ripple adder is necessary at the end to sum the carry and sum outputs of the last carry save adder stage as shown in figure2.28.

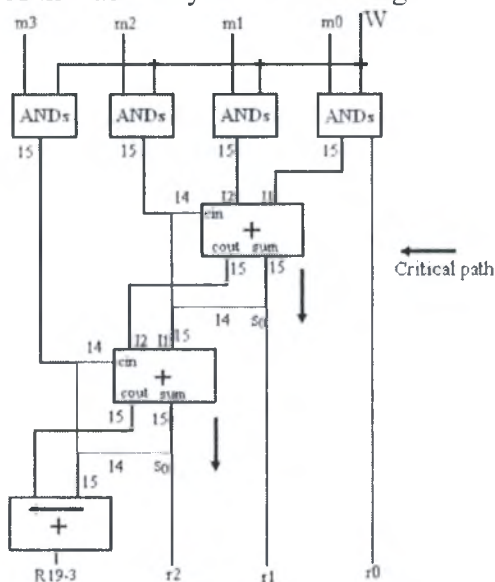


Figure2.28: Carry-save multiplier architecture.

In $M \times N$ multiplier, the critical path delay can be calculated as follow:

$$t_{mult} = (N - 1)t_{carry} + t_{and} + (M - 1)t_{sum}$$

2.2.2.3 f-Modification Stage

The value calculated in quantization parameters calculation phase is a base value $\frac{2^{qbits}}{3}$ which should be modified for inter-prediction mode, since we need the value to be $\frac{2^{qbits}}{6}$, or we need to multiply it by 2 for the case of chroma or luma intra 16×16 components.

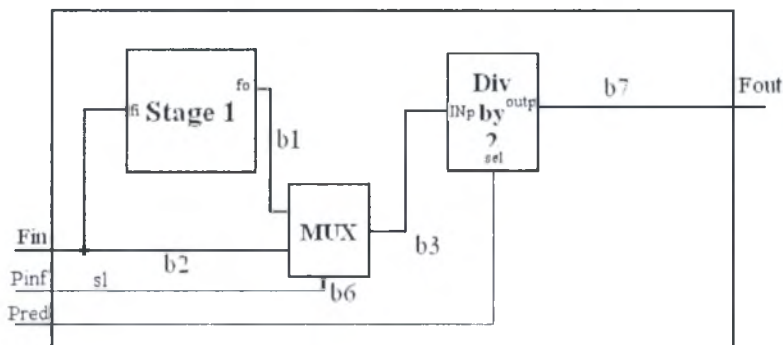


Figure2.29: f-modification structure.

The value $\frac{2^{qbits}}{6}$ can be obtained simply by division of f value by 2, which can be easily implemented as a right shift by 1-digit. Figure2.29 shows the f -modification phase.

Stage1 performs the multiplication by 2 and the **Mux** selects, according to the component type, to pass the original value or the multiplied one. Then at the division by 2, according to the prediction signal we perform the division or pass the value.

Multiplication by 2 consists of two operations; at first left shift by one bit, then, addition of value '1' is done according to f -value. Addition by one is necessary for compensating the loss occurred in the division by 3 and flooring process. To understand this lets see examples for calculating f value:

$$\begin{aligned} \left(2^{15}/3\right) &= 10922\frac{2}{3} & \left(2^{16}/3\right) &= 21845\frac{1}{3} \\ \left(2^{17}/3\right) &= 43690\frac{2}{3} & \left(2^{18}/3\right) &= 87381\frac{1}{3} \end{aligned}$$

From the examples it is clear that for odd powers (qbits = 15, 17, ...) there is a decimal digit 0.6666 which ignored in division process, while for even powers (qbits = 16, 18, ...) the decimal digit is 0.3333, so the loss of 0.6666 when we make floor process, should be covered by addition of 1 since when we multiply it by 2 the result will be 1.3333, while this is not the case for decimal digits 0.3333 which by multiply by 2 still below 1 which again will be ignored.

The problem that arises, is when we know if the addition with 1 is necessary, and how it will be implemented without using an adder??

We can note that the value of f has the following property: the values are even for odd powers (10922, 43690, ...) and odd for even powers (21845, 87381, ...). By checking the first bit of f we can understand if the corresponding number is even or odd ('0' means even and '1' means odd). So the addition of 1 after left shift process is implemented simply by passing '1' on the first bit when addition is needed or '0' when it is not necessary.

2.2.2.4 Addition of f and Left Shift process

The addition stage consists of a 29-bit adder. A carry lookahead adder is implemented for this purpose. There are two reasons why we should implement special adder:

1. One of the inputs is 22-bits size and extended with 8-bits of '0' value. This information can be used to remove the necessary AND gates needed to calculate g values (generate carry), and save the XOR gates needed to calculate p values (propagate carry).
2. Since the right shift process will be at least 15- digits, there is no necessity to calculate the first 15-bits of sum, which reduces the output bus to 15 bits only. Also the information of zero input carry can be used to remove necessity to propagate '0' value.

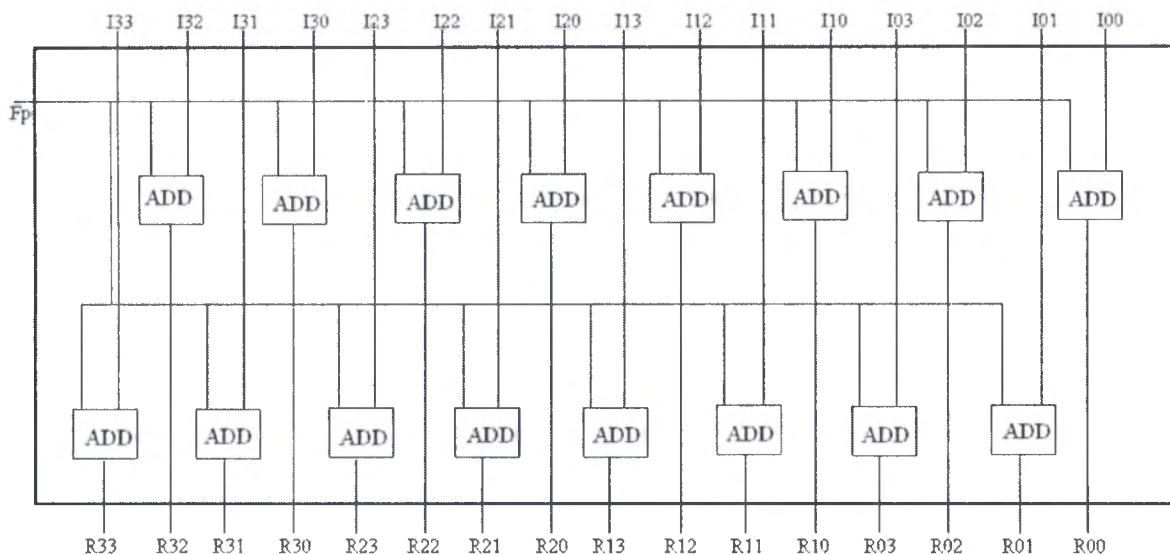


Figure2.30: Addition Unit Structure.

Right shift process can be implemented using funnel shifter structure as shown in figure2.31 for 4-bits input and output. '0' extension is used here to fill the left output bits. S bits are the select bits that decide the number of shift digits.

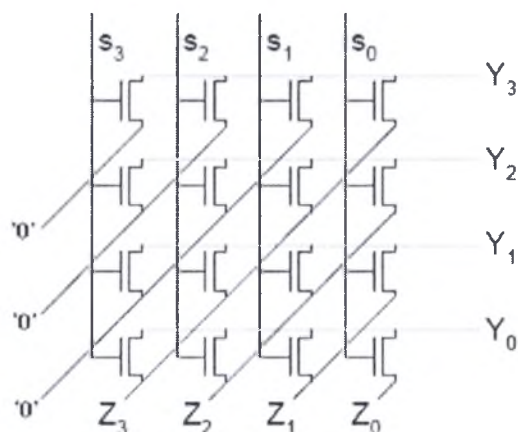


Figure2.31: 4-bits funnel shifter.

In our shifter the maximum number of shifts is 9 digits, since *qbits* range is 15-23 and in the case of chroma and luma intra16×16 components we add 1.

The table2.1 shows the shifter table.

Table2.1: funnel shifter map table.

Shift digits	y14	y13	y12	Y11	Y10	y9	y8	y7	y6	y5	y4	y3	y2	y1	y0
15	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15
16	0	29	28	27	26	25	24	23	22	21	20	19	18	17	16
17	0	0	29	28	27	26	25	24	23	22	21	20	19	18	17
18	0	0	0	29	28	27	26	25	24	23	22	21	20	19	18
19	0	0	0	0	29	28	27	26	25	24	23	22	21	20	19
20	0	0	0	0	0	29	28	27	26	25	24	23	22	21	20
21	0	0	0	0	0	0	29	28	27	26	25	24	23	22	21
22	0	0	0	0	0	0	0	29	28	27	26	25	24	23	22
23	0	0	0	0	0	0	0	0	29	28	27	26	25	24	23
24	0	0	0	0	0	0	0	0	0	29	28	27	26	25	24

The first row is the 15-bit output row, note that we use in the table, index of the 30-bit adder output, to indicate bits locations and movements.

The shift digits (*qbits*) and an additional bit (*piif*), which indicates if chroma and luma intra16×16 components or others is used to calculate the select bits used to control the funnel shifter. In figure2.32 the enable block is responsible of calculating the select bits that distributed to all the 16- shifting units.

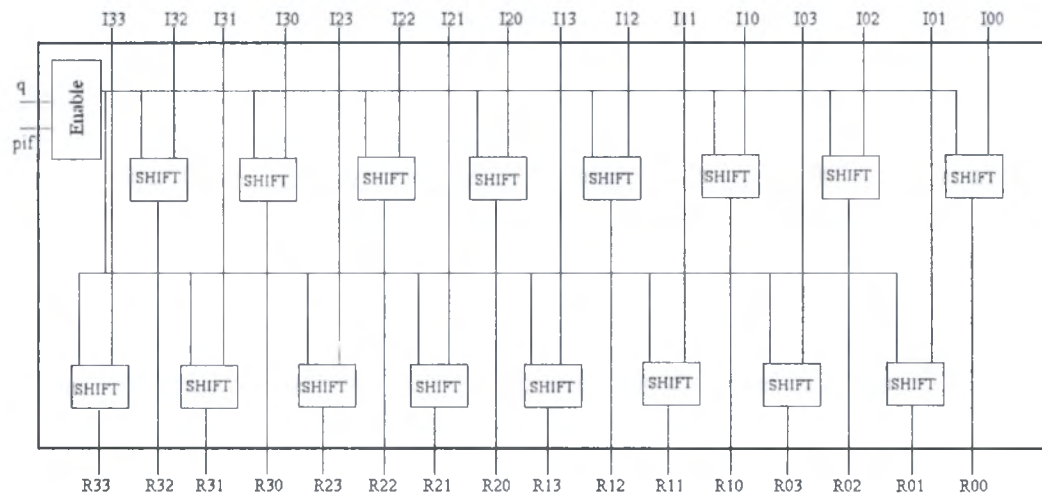


Figure2.32: Shift Unit structure.

2.2.2.5 Resigning Phase

Resigning phase can be implemented as absolute value process by applying 2's complement on the shifter output according to the sign bit of the original input value.

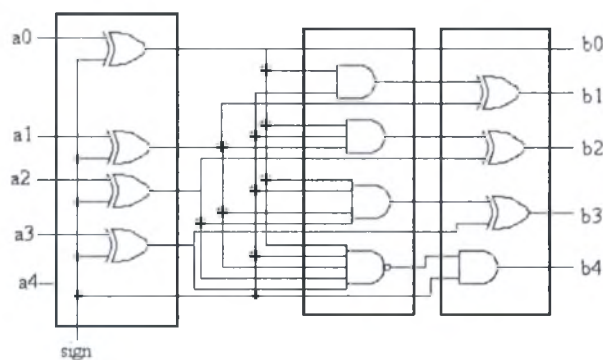


Figure2.33: resign function structure.

However, in resigning phase, the sign bit controls the inversion and carry propagation process. The result size is 15-bits, like the input (15-bits shifter output) where the last bit is ignored since it is always '0'. To prove this, let us see the maximum number of shifter output:

$$\left(\frac{|-2^{15}| * (2^{13} - 1) + (2^{22} - 1)}{2^{15}} \right) = 8319$$

The values represents, in order, the max values of w_{ij} , multiplication factor, and f -value, and 2^{15} represent the minimum number of shift digits. The result (8319) is represented in 15-bits binary form as (010000001111111), so the last bit is always '0'

and can be ignored. The sign bit of the result (b4) is calculated as shown in the figure to avoid propagating the sign bit (for negative inputs) if the shift result equal zero.

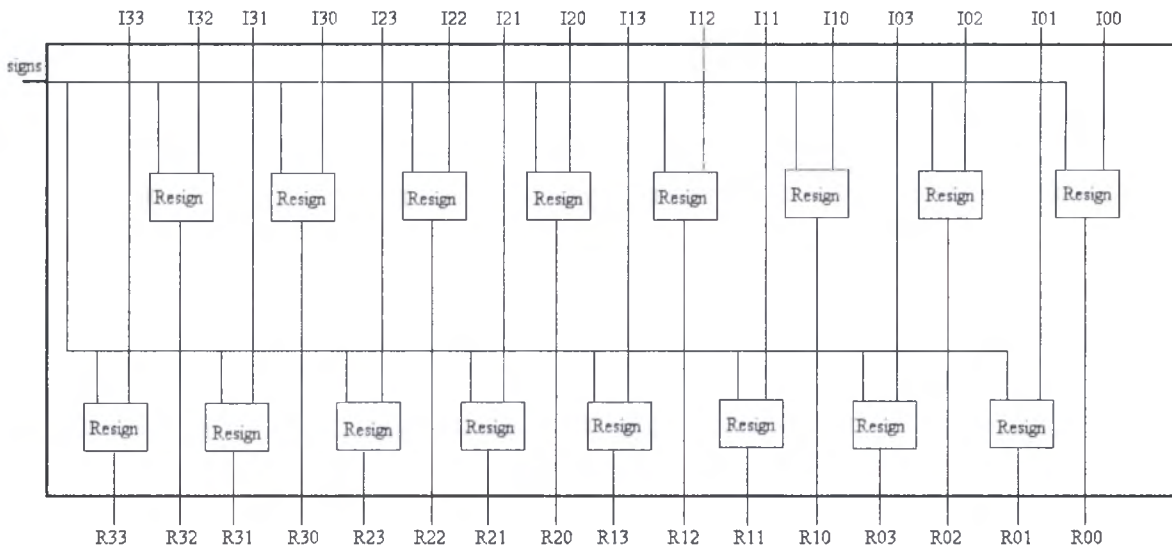


Figure2.34: Resigning Unit structure.

2.3 Quantization using f -Modification Method

Absolute value and resigning operations, as implemented in the previous section, add an overhead on the computation process. The quantization process actually consists of multiplication, addition which is necessary for correct rounding, and finally a right shifting phase which performs the division process. The absolute value and resigning phase are used to perform correct rounding for negative values with the same value for f in case of positive numbers.

The new form of quantization which appears in equations (1.28) and (1.29) is based on the modification of f -value according to the sign of data elements, so we can perform quantization, while avoiding the absolute value and resigning stages. The structure of the new suggested quantization process appears in figure2.35.

The suggested modification saves area and time, since two values for f are calculated; one for the positive and another for the negative case. A multiplexer, at the addition stage selects the appropriate value according to the sign bit.

The same quantization parameters calculating stage used for standard quantization process can be used here.

The differences for this new quantization form are:

1. Using signed multipliers and adders.
2. f -modification stage has some additional blocks.

Multiplexers' stage is used before the addition unit.

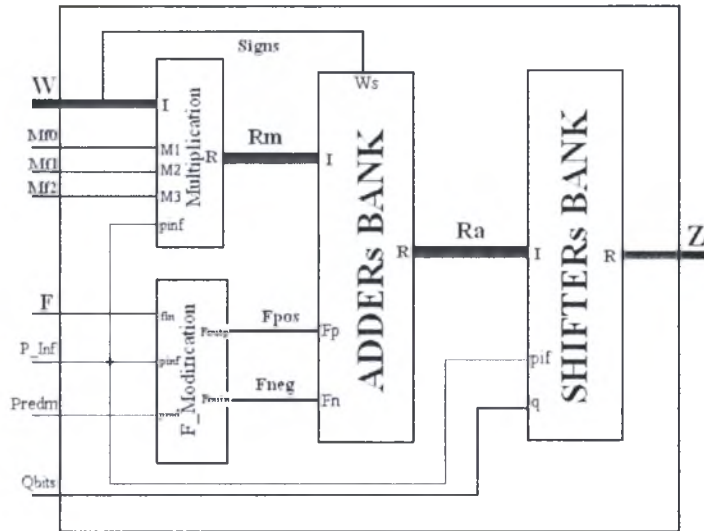


Figure2.35: Quantization using f -Modification Method structure.

2.3.1 F-Modification Stage

Figure2.36 shows the block diagram of the f -modification stage. This stage gives us 2 values of f ; one for positive data elements and the other for negative data elements. At the input of the adders a multiplexer selects which one will be used, according to the sign of the multiplication result.

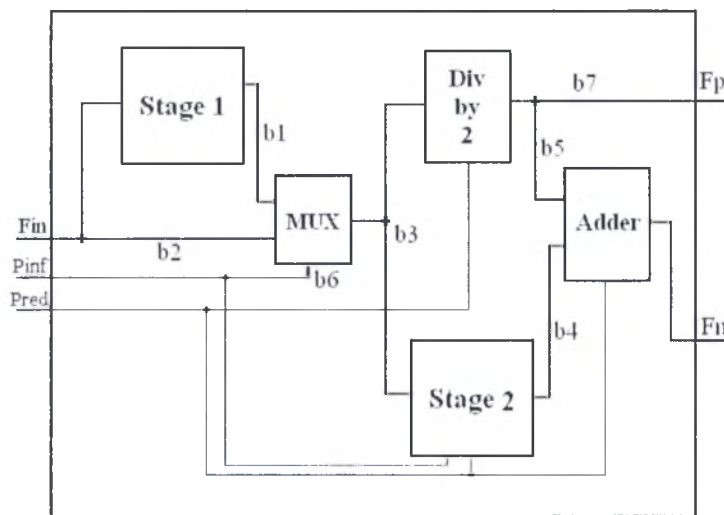


Figure2.36: F-Modification Stage structure.

The block takes the basic value of f ($2^{qbits}/3$) which was calculated in quantization parameters calculation stage, and a set of modifications occur according to the prediction mode (Intra or Inter: carried by signaled Pred input) and components information (chroma or luma: signaled by Pinf input).

I/O Details:

1. Fin: 22bits buss carries the value of $f(2^{qbits}/3)$.
2. Pinf: 1bit line tells if chroma or luma intra16 mode case (Pinf = '0') or if other modes (Pinf = '1').

3. Pred: 1bit line tells if Intra mode prediction (Pred = '0') or Inter mode prediction (Pred = '1').
4. Fp: 23bits buss carries the value of f for positive integers.
5. Fn: 25bits buss carries the value of f for negative integers.

Stage 1 & MUX:

Stage 1 performs multiplication by 2 on F_{in} so we have $2f$ in case that the 4×4 data block is chroma components or luma components predicted in the Intra16 mode. The MUX will select according to P_{inf} whether to pass f or $2f$. In stage1 we add 1 after multiplication by 2 to recover the lost 1 by flooring of f as we discussed in details in section (2.2.2.3).

DIV by 2:

Div by 2 block gives $2^{q_{bits}}/6$ simply by performing right shift with 1bit. However it is controlled by Pred input, because for intra mode it should not perform division, since we need value of f at the positive output to be $2^{q_{bits}}/3$ and not $2^{q_{bits}}/6$.

In case of Inter prediction, the result of "Div by 2" block will be necessary to create the negative case value of $f = 5 * (2^{q_{bits}}/6)$ as follow:

$$5 * (2^{q_{bits}}/6) = (4 + 1) * (2^{q_{bits}}/6) = 4 * (2^{q_{bits}}/6) + (2^{q_{bits}}/6) = 2 * (2^{q_{bits}}/3) + (2^{q_{bits}}/6) \quad (2.16)$$

Also we need $2 * (2^{q_{bits}}/3)$ for the case of Intra mode.

Stage 2 & Adder:

Stage 2 performs multiplication by 2 on the incoming value of f (which can be either f or $2f$) since we need it as indicated in equation (2.1) and for Intra mode. Inside stage2 an addition with 1 is done after multiplication in order to recover lost 1 from flooring depending on Pred and P_{inf} values as follow:

1. In case of Pred = '1' (means Inter mode) we always add 1.
2. In case of Pred = '0' and P_{inf} = '0' we add 1 when we don't add 1 in stage 1.
3. In case of Pred = '0' and P_{inf} = '1' we add 1 according the same conditions of adding 1 in stage 1.

2.3.2 Addition Stage

As we mentioned above at each adder input a multiplexer should be used to select f -value of positive integers or f -value of negative integers. This multiplexer is controlled by the sign bit in the multiplication output value. One may claim that the multiplexing process add an overhead on data flow path. However, the area overhead from multiplexers is negligible compared with the area overhead of 16 absolute value units and 16 resigning units.

As for the time overhead, this is also negligible, since each multiplexer adds only ANDing level and ORing level, compared with the absolute value and resigning units where each one contain the carry propagation hardware over 15-bits in addition to 2-XORing levels.

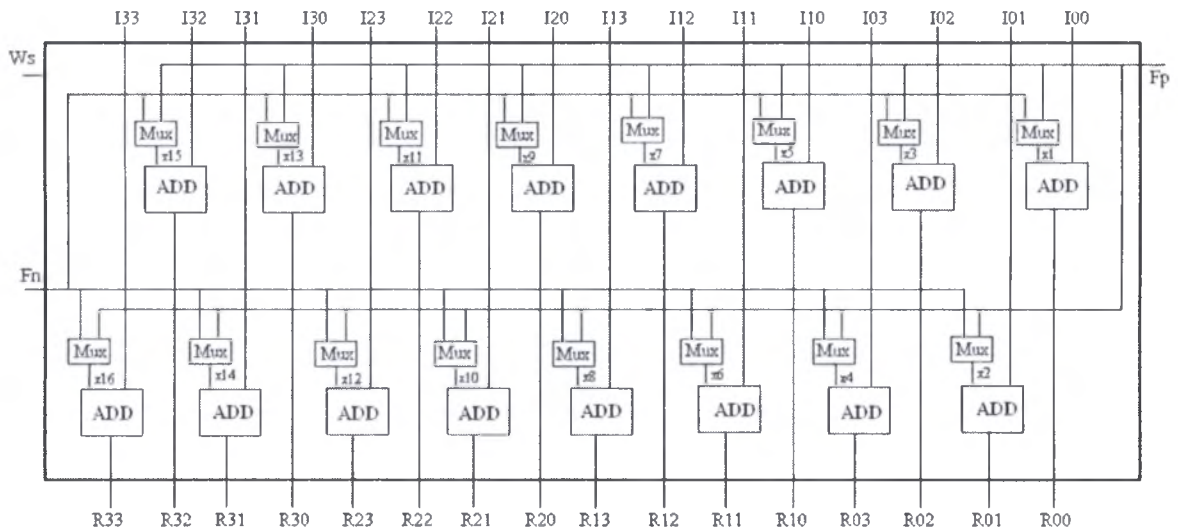


Figure2.37: Addition Unit structure.

Regardless of the above discussion, the time overhead of multiplexers can be removed completely by forwarding the sign bits of all the data elements that enter the multiplication unit and use them to control the multiplexers. This is possible since we know that the sign of the multiplier output result is the same as the sign of the input data element. In figure2.37, Ws is the sign bits buss (16-bits) which used to control the 16-multiplexer.

2.4 Inverse Transform and Quantization

Inverse process can be described in four stages as follow:

1. Inverse 4×4 and 2×2 hadmard transforms are applied on the input 4×4 data block. Then, a multiplexer selects to pass the input 4×4 data block or one of the 4×4 data blocks resulting from inverse hadmard transforms units.
2. Rescale (quantization inverse) process is performed on the 4×4 data block from the previous step. Equations (1.23), (1.25) and (1.27) describe the rescale process for residual components, luma intra 16×16 mode components, and chroma components respectively.
3. Inverse core transform process as described in equation (1.11).
4. Divide over 64 and rounding the result.

Inputs/Outputs Description:

- **Z:** 4×4 data block with each element is 15-bit size.
- **P_INF:** 2-bits bus carry information about the data components type, as described in section 2.1
- **QP:** quantization parameter controls the quantization steps as appropriate. It takes values from 0 up 51.
- **X:** 4×4 reconstructed data block with each element having 10-bit size. Since the original data elements coming out from prediction phase, are 9-bits signed integers, and the inverse process results in approximated values, we add

additional bit represents the data elements resulting from inverse process by 10-bits signed integers. For example; -16 is represented by 5 bits while -17 needs 6 bits so to avoid errors we incur additional bit.

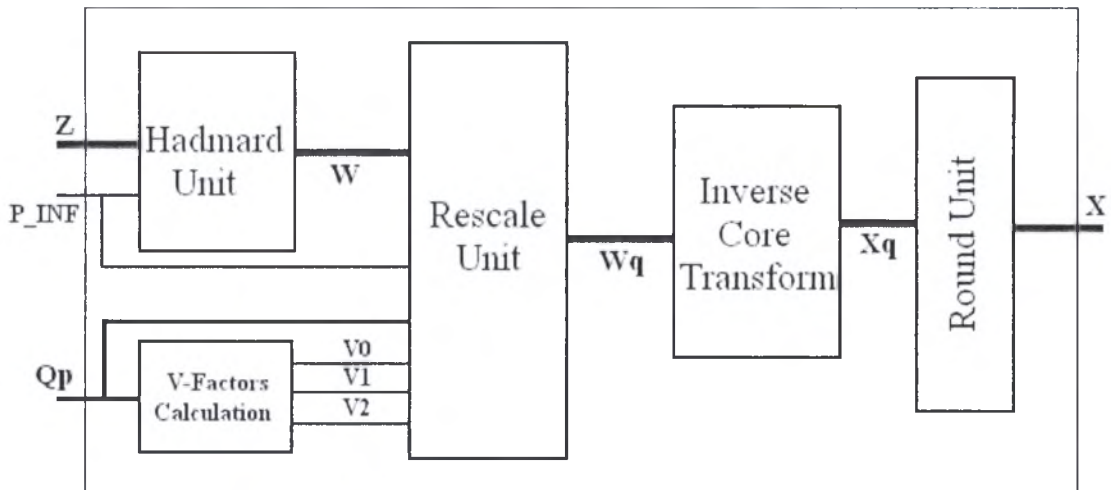


Figure2.38: Inverse transform and quantization structure.

2.4.1 Inverse 4×4 and 2×2 Hadmard Transforms

Equations (1.13) and (1.14) describe 4×4 and 2×2 inverse hadmard transforms respectively. Figure2.39 shows the structure of first stage.

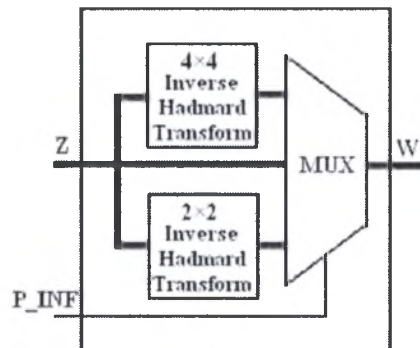


Figure3.39: Inverse Hadmard transform unit.

Inverse 4×4 hadmard transform matrices have the same form as the forward 4×4 hadmard transform since they are symmetric, so the same structure built for the forward 4×4 hadmard transform can be used here by changing only the size of the adders. On the first level of butterflies' blocks we use 15-bit and 16-bit lookahead adders, and on the second level of butterflies' blocks we use 17-bit and 18-bit lookahead adders.

As we stated in the implementation of forward 2×2 hadmard transform, the first level of butterflies' blocks used for the 4×4 inverse hadmard transform can be used to implement the 2×2 inverse hadmard transform, so to avoid configurator here we duplicate the first level of butterflies' blocks in 4×4 inverse hadmard transform to build 2×2 inverse hadmard transform.

Hardware Estimation: From the above description of used hardware we can state estimation of the used cells:

1. 32 15-bit signed adders.
2. 32 16-bit signed adders.
3. 16 17-bit signed adders.
4. 16 18-bit signed adders.
5. 16 15-bit inverters' blocks.
6. 16 16-bit inverters' blocks.
7. 8 17-bit inverters' blocks.
8. 8 18-bit inverters' blocks.

2.4.2 V-Factors Calculation

Table 1.5 in section (1.3.2.1) lists the values of V-scaling factors used in equation (1.23). The values occur in the range from 10 up to 26, so 5-bits are necessary to represent the values of V-factors. V-factors depend on $(QP \bmod 6)$ result.

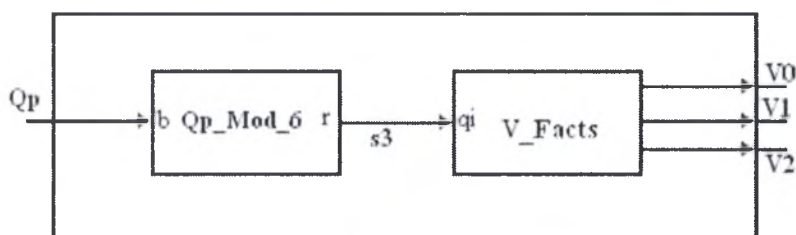


Figure2.40: V-Factors calculation.

2.4.3 Rescale Unit

Rescale unit realizes the inverse quantization process. Rescaling process mainly consist of multiplication and shifting stages.

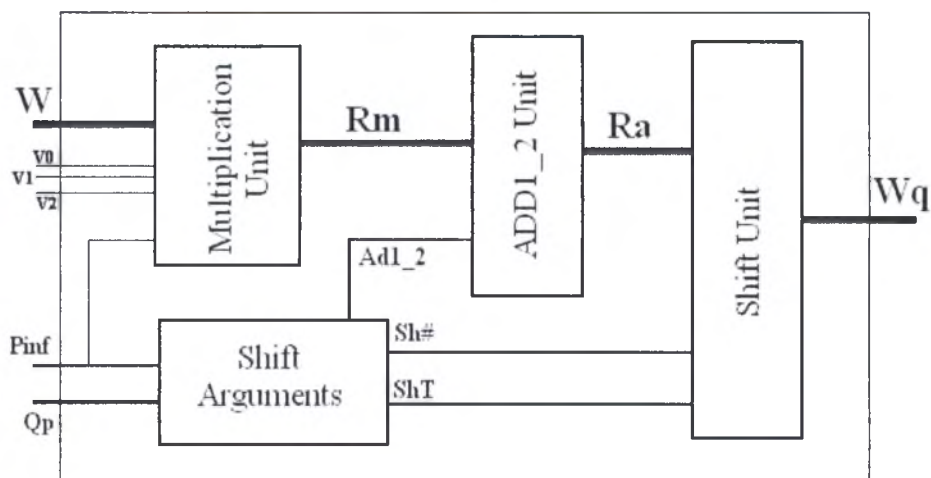


Figure2.41: Rescale Unit Structure.

A 5×19 tree multiplier is implemented. Only three levels of adders are necessary in addition to the partial products calculation stage. 4 adders are used; two 19-bit signed adders, one 21-bit signed adder, and the last one is 23-bit signed adder. After multiplication stage, addition of 1 or 2 before shifting is necessary for special cases as stated below:

1. Addition of '1' is necessary when $6 \leq Qp < 12$ and data components are of luma type predicted in 16×16 mode. This is clear from equation (1.24) where we have $(+2^{1-\text{floor}(QP/6)})$ for $QP < 12$, which give us $+2^{(1-1)}$, when $6 \leq Qp < 12$.
2. Addition of '2' is necessary when $Qp < 6$ since $(+2^{1-\text{floor}(QP/6)})$ will give us $+2^{(1-0)} = 2$

So **Pinf**, and **Qp** inputs control the value of **Ad1_2** bus running from **shift arguments unit** to **ADD1_2Unit**, **Ad1_2** is 2-bit bus carries value '0', '1' or '2'.

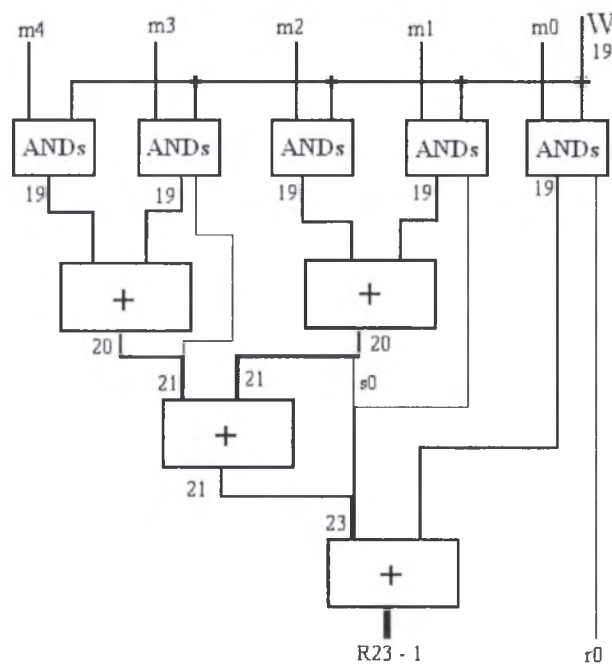


Figure2.42: 5×19 tree multiplier used in rescale unit.

Shift arguments unit also calculates the shift type; left or right, and number of shift bits which ranges from (0 – 8) for left shift, and (0 – 2) for right shift.

2.4.4 Inverse Core Transform

Inverse core transform is based on a butterfly block of adders which has the structure shown in figure2.43. The **SR** blocks are shift right by 1-bit blocks to realize the division by 2 as we will see below.

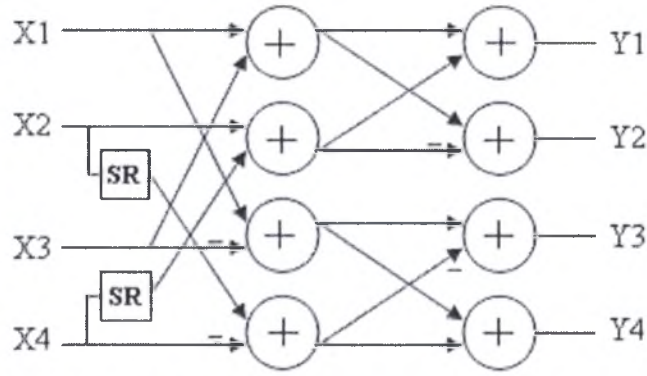


Figure 2.43: Butterfly adders' block structure used in inverse core transform.

The core transform process shown in equation (1.11) can be written as below:

$$X' = (C_i^T W' C_i) = \begin{pmatrix} 1 & 1 & 1 & \frac{1}{2} \\ 1 & \frac{1}{2} & -1 & -1 \\ 1 & -\frac{1}{2} & -1 & 1 \\ 1 & -1 & 1 & -\frac{1}{2} \end{pmatrix} * \begin{bmatrix} w_{00} & w_{01} & w_{02} & w_{03} \\ w_{10} & w_{11} & w_{12} & w_{13} \\ w_{20} & w_{21} & w_{22} & w_{23} \\ w_{30} & w_{31} & w_{32} & w_{33} \end{bmatrix} * \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \frac{1}{2} & -\frac{1}{2} & -1 \\ 1 & -1 & -1 & 1 \\ \frac{1}{2} & -1 & 1 & -\frac{1}{2} \end{bmatrix} \quad (2.17)$$

The first two matrix multiplication can be written as:

$$\begin{pmatrix} w_{00} + w_{10} + w_{20} + \frac{w_{30}}{2} \\ w_{00} + \frac{w_{10}}{2} - w_{20} - w_{30} \\ w_{00} - \frac{w_{10}}{2} - w_{20} + w_{30} \\ w_{00} - w_{10} + w_{20} - \frac{w_{30}}{2} \end{pmatrix} \begin{pmatrix} w_{01} + w_{11} + w_{21} + \frac{w_{31}}{2} \\ w_{01} + \frac{w_{11}}{2} - w_{21} - w_{31} \\ w_{01} - \frac{w_{11}}{2} - w_{21} + w_{31} \\ w_{01} - w_{11} + w_{21} - \frac{w_{31}}{2} \end{pmatrix} \begin{pmatrix} w_{02} + w_{12} + w_{22} + \frac{w_{32}}{2} \\ w_{02} + \frac{w_{12}}{2} - w_{22} - w_{32} \\ w_{02} - \frac{w_{12}}{2} - w_{22} + w_{32} \\ w_{02} - w_{12} + w_{22} - \frac{w_{32}}{2} \end{pmatrix} \begin{pmatrix} w_{03} + w_{13} + w_{23} + \frac{w_{33}}{2} \\ w_{03} + \frac{w_{13}}{2} - w_{23} - w_{33} \\ w_{03} - \frac{w_{13}}{2} - w_{23} + w_{33} \\ w_{03} - w_{13} + w_{23} - \frac{w_{33}}{2} \end{pmatrix} \quad (2.18)$$

This can be renamed as

$$S = \begin{bmatrix} s_{00} & s_{01} & s_{02} & s_{03} \\ s_{10} & s_{11} & s_{12} & s_{13} \\ s_{20} & s_{21} & s_{22} & s_{23} \\ s_{30} & s_{31} & s_{32} & s_{33} \end{bmatrix}$$

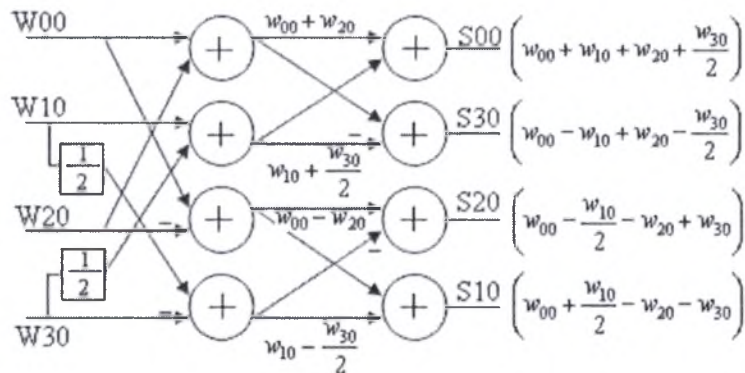


Figure 2.44: First level butterfly adders' block used in inverse core transform.

A Butterfly block arrangement can be used to implement this first matrix multiplication as shown in figure2.44. The adders used are 17-bit look ahead signed adders.

The matrix multiplication of the result matrix S with C_i will result in the following matrix:

$$\begin{bmatrix} \left(s_{00} + s_{01} + s_{02} + \frac{s_{03}}{2} \right) & \left(s_{00} + \frac{s_{01}}{2} - s_{02} - s_{03} \right) & \left(s_{00} - \frac{s_{01}}{2} - s_{02} + s_{03} \right) & \left(s_{00} - s_{01} + s_{02} - \frac{s_{03}}{2} \right) \\ \left(s_{10} + s_{11} + s_{12} + \frac{s_{13}}{2} \right) & \left(s_{10} + \frac{s_{11}}{2} - s_{12} - s_{13} \right) & \left(s_{10} - \frac{s_{11}}{2} - s_{12} + s_{13} \right) & \left(s_{10} - s_{11} + s_{12} - \frac{s_{13}}{2} \right) \\ \left(s_{20} + s_{21} + s_{22} + \frac{s_{23}}{2} \right) & \left(s_{20} + \frac{s_{21}}{2} - s_{22} - s_{23} \right) & \left(s_{20} - \frac{s_{21}}{2} - s_{22} + s_{23} \right) & \left(s_{20} - s_{21} + s_{22} - \frac{s_{23}}{2} \right) \\ \left(s_{30} + s_{31} + s_{32} + \frac{s_{33}}{2} \right) & \left(s_{30} + \frac{s_{31}}{2} - s_{32} - s_{33} \right) & \left(s_{30} - \frac{s_{31}}{2} - s_{32} + s_{33} \right) & \left(s_{30} - s_{31} + s_{32} - \frac{s_{33}}{2} \right) \end{bmatrix} \quad (2.19)$$

Which can be also realized using the upper structure of butterfly as shown in figure2.45. The adders used are 16-bit look ahead signed adders.

Estimated hardware: From the above description of used hardware we can estimate the cells needed for this implementation:

1. 32 17-bit signed adders.
2. 32 16-bit signed adders
3. 8 17-bit inverters' blocks.
4. 8 16-bit inverters' blocks.

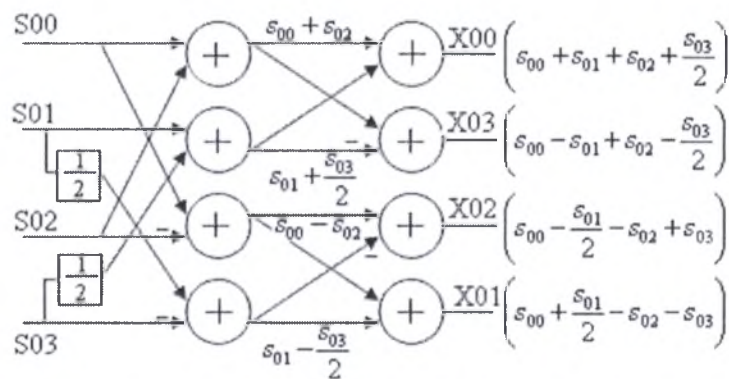


Figure2.45: Second level butterfly adders' block used in inverse core transform.

2.4.5 Round Unit

The last step before getting the reconstructed 4×4 data block is a division by 64 and a rounding. The division by 64 is implemented by shift right of 6 places process. Rounding is done after the shift process by adding '1' to the shifter output according to the least significant 6-bits of the shifter 16-bit input data. Round process is done as follows:

- The round process is done if the decimal digit is larger than or equal to (0.5) for positive numbers, and if the decimal digit is larger than or equal to (-0.5) for negative numbers.

- (0.5) and (-0.5) decimal digits mean that the least significant 6-bits of the number represent (32) and (-32) for positive and negative numbers respectively.
- From the above two conditions, a test of the sixth bit will tell us if the decimal digit is larger than (0.5) and (-0.5) or not. If the sixth bit is '1' then we add '1' since it indicated decimal digit larger than (0.5) or (-0.5), else addition of '0' is done.

Figure2.46 shows the implementation of division and round process after inverse core transform.

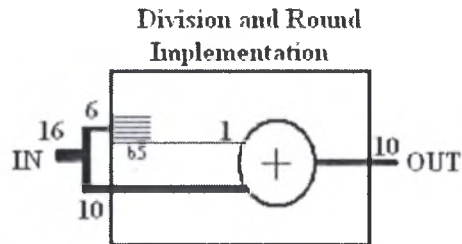


Figure2.46: Division by 64 and round implementation.

3 Optimization and Synthesis

The theoretical designs and paper written concepts will not be useful until their *realization* and *fabrication*. A good written design must take care to the available capabilities and real life degradations or *constraints*. To jump from paper written design to a working fabricated design, you should pass two processes; *optimization* and *synthesis*. The best possible design performance while meeting the design constraints can be obtained by optimization process, while synthesis process can be seen as implementing the design in gate level and getting a fabricatable design. However, the two processes are not separated as appear, really they are mixed in some level, and each depends on the other results.

An unsynthesizable design is that one can't be realized with available capabilities, or in other words, its synthesis result didn't meet its constraints (i.e. *constraints violation*). The designer should be aware that there are always limitations and so its goals should be more real, so it can be realized.

3.1 Design Optimization

Design optimization means getting the best performance design while meeting the design constraints. The design performance meter or optimization targets include delay, consumed power, area, and design complexity; other targets like cost, life time, flexibility, etc, also can be considered. However, the most critical optimization parameters are the firstly mentioned three parameters; delay, power, and area.

Design with best performance is application related meter. Some applications are interested more in minimizing consumed power rather than delay and area, like battery based processors used in kids' toys. Others are interested more in delay/power product minimization, so get the minimum delay while consuming low power like processors used in portable computing machines. In other applications a high speed chips are very necessary like parallel processors desktops. In general area meter works always as a constraint which can be violated, however, the minimum possible area while meeting higher priority constraints can be calculated.

3.1.1 Modeling of optimization problem

An optimization problem can be mathematically formulated in the following form:

minimize Z
according to the constraints

$$c_i \leq f_{\max} \quad i = 0, 1, \dots, n.$$

Z is an expression which models the optimization target while satisfying the constraints set c_i . Examples of optimizations problems are:

Example1:

$$\begin{aligned} & \text{Minimize } D \times P \\ & \text{While} \\ & \quad A \leq A_{\max}. \\ & \quad t_r \leq t_{\min} \end{aligned} \quad (3.1)$$

Example2:

$$\begin{aligned} & \text{Minimize } D \\ & \text{While} \\ & \quad A \leq A_{\max}. \\ & \quad P \leq P_{\max} \\ & \quad t_r \leq t_{\min} \end{aligned} \quad (3.2)$$

One of the widely used ways to model optimization problems is geometric programming.

A *geometric program* (GP) is an optimization problem of the form

$$\begin{aligned} & \text{Minimize } f_0(x) \\ & \text{Subject to } f_i(x) \leq 1 \quad \text{where } i = 1, \dots, m. \\ & \quad g_i(x) = 1 \quad \text{where } i = 1, \dots, p. \end{aligned} \quad (3.3)$$

Where f_i are posynomial functions, g_i are monomials functions, and x_i are the optimization variables (there is implicit constraint that x_i variable are positive).

A posynomial function is the sum of one or more monomials functions, and monomial function is the one of the form

$$f(x) = cx_1^{a_1} x_2^{a_2} \dots x_n^{a_n} \quad (3.4)$$

Where $c > 0$ and $a_i \in \mathbf{R}$.

A mathematically modeled optimization problem can be solved, using appropriate type of solver. One of the widely used solvers is the sparse solver.

3.1.2 Optimizations Levels

Design optimization process starts from the mathematical modeling of the design problem, passing through the RTL level till reaching gates design level.

A good mathematical modeling of the design problem can enable avoiding unnecessary steps in minimizing logic. Example of this, the DCT based transformation supported by the H.264 standard, enables implementation of transformation phase using integer hardware and avoids using of multipliers. Also, good modeling of the design problem will make design partitioning easier, and makes separated tasks appear clearly.

RTL level optimization includes:

- Sharing resources.
- Factorizing expressions.
- Parallel processing.

Sharing resources can be done on a time spaced operations, where the same block can perform all the operations if they are time spaced enough. Like in figure below, since we take one addition result at a time, we can reorder the process so we use one adder and select the appropriate inputs.

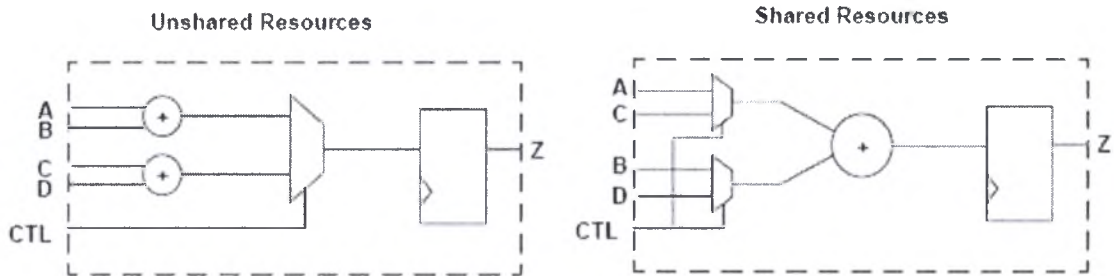


Figure3.1: Sharing resources.

Factorizing of expressions can be explained as follow:

$$Y = X_1 * Z + X_1 * K + X_2 * Z + X_2 * K \quad (3.5)$$

Can be rewritten as

$$Y = (X_1 + X_2) * (Z + K) \quad (3.6)$$

Figure3.2 shows how this factorization saves time and area; the original expression requires 4 multipliers and 3 adders, and output the result after 3 levels of calculation, while the factorization result requires 2 adders and 1 multiplier, and performs calculation in two levels.

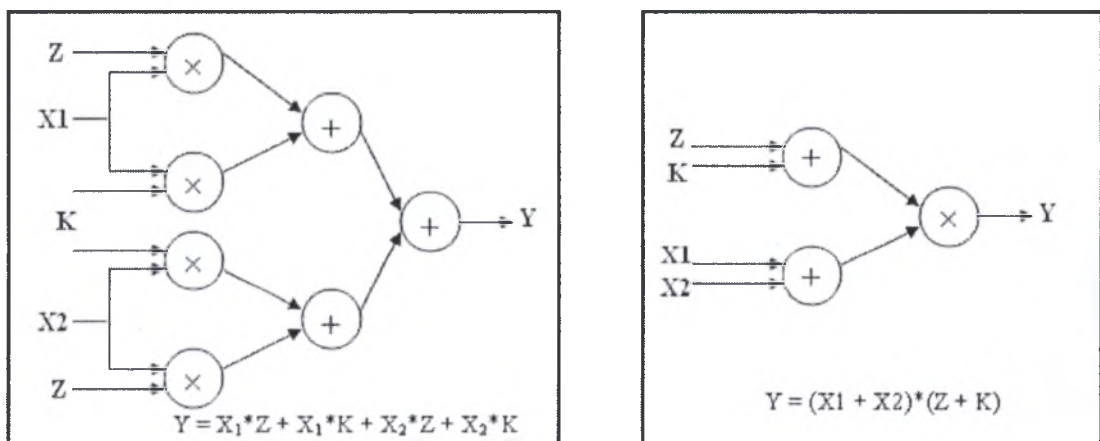


Figure3.2: Factorizing expressions.

Parallel processing also can be used to obtain faster calculation of results, as shown in the example shown in figure3.3, grouping of expressions show the parallelism in the expressions.

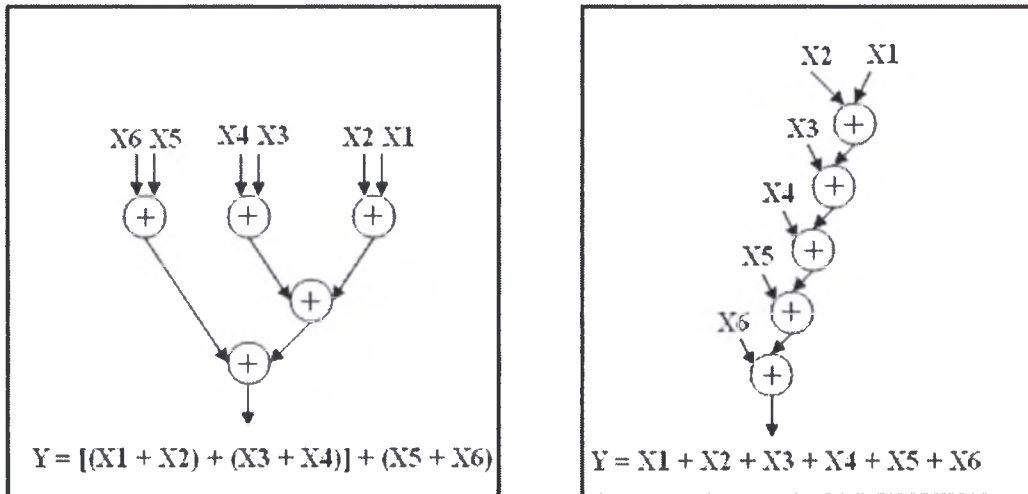


Figure3.3: Parallel processing.

Logic level optimization deals with implementation of RTL level components, like registers, adders, comparators, sequential logic, etc.

This level creates a generic netlist that realize the function of each component. Logic implementation of a function starts from truth table representation, from truth table a minimization process can be ran and logical expressions that realize the function are obtained. More over optimizations can be done by sharing logic subexpressions, factorizing logic expressions, and using Demorgan's laws. Sharing and factorizing of logic expressions is the same as what we saw for RTL level, but here we deal with logic operators than arithmetic operators. A factorization example is shown below.

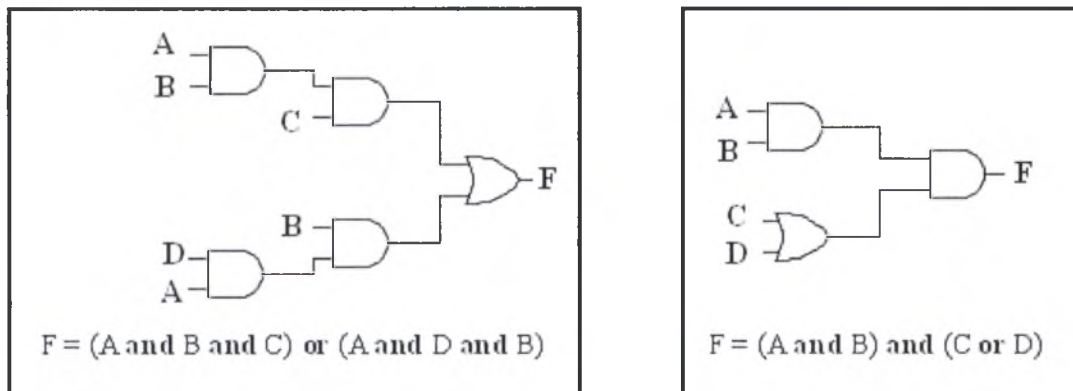


Figure3.4: logic level optimizations.

Demorgan's laws also could enable us to minimize the expressions in some way. As in the simple example shown below, F can be implemented using 1 OR and 1 NOT gates rather than 1 AND and 2 NOT gates.

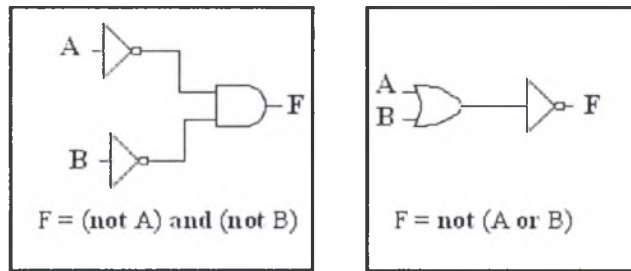


Figure3.5: minimizations using Demorgan's laws.

In *gate level optimization* the created generic netlist in the logic level is realized, where the gate implementation style is selected; static, or dynamic. Also transistors sizing occurs in this level, nets layers and sizes also are decided. At this level actual calculation of timing, area and power parameters occur, and the sizing and implementation process is iterated until the constraints are met.

3.2 Design Synthesis

Implementation of digital design will be using digital gates and transistors, beside the appropriate set of interconnections, resistors, capacitors, and power supply resources. The process of creating gates netlist and deciding the sizes of all the elements of design (gates, transistors, capacitors, resistors, interconnections, pins) and creating design layout and meeting design constraints is called *synthesis process*.

Really, synthesis process is done by Computer Aided Design tool, since it is not realistic to perform synthesis process of design with thousands of gates. Any synthesis process will base on three main elements:

1. Design files.
2. Technology and Synthetic library.
3. Design constraints and environment.

3.2.1 Synthesis Setup

Design files

The system design should be written in a hardware description language (VHDL, Verilog...), or any other appropriate language for the synthesis tool. Also the written design should be organized in files according to the design hierarchy. It is better to partition the design into separated subdesigns arranged in hierarchal way, which simplify the synthesis process, and save effort.

The simple example of partitioning design is the simple calculator example appears in figure 3.6, the partitioning process is better to continue until you reach the arithmetic and logical operators design, and then any structure can be built from those simple files.

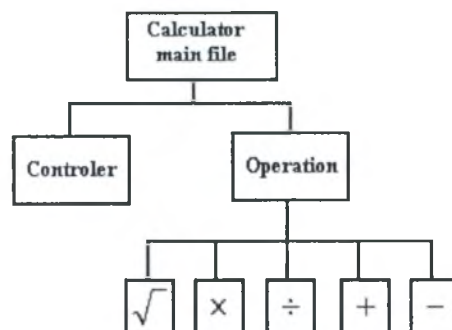


Figure3.6: Design files Hierarchy.

Technology and Synthetic library

Technology libraries contain information about the characteristics and functions of each cell provided in a semiconductor vendor's library. Semiconductor vendors maintain and distribute the technology libraries.

Cell characteristics include information such as cell names, pin names, area, delay arcs, and pin loading. The technology library also defines the conditions that must be met for a functional design (for example, the maximum transition time for nets). These conditions are called design rule constraints.

Examples of cells are those implements functions like AND, OR, XOR, NAND, NOT, and many others. Each technology library supports only one semiconductor technology like for example 180nm, or 250 nm technologies.

In addition to cell information and design rule constraints, technology libraries specify the operating conditions and wire load models specific to that technology.

Synthetic library (or Design Ware library) is a collection of reusable circuit-design building blocks (components) that implement many of the built-in HDL operators. These operators include +, -, *, <, >, <=, >=, and the operations defined by *if* and *case* statements. Most of these operators' implementations are provided by the synthesis tools. For example, a variety of adders' implementations can be found for the addition operator in the synthetic library.

However, you can develop additional Design Ware components and provide them to the synthesis tool.

Design Constraints and Environment

The real life will have no meaning without limitations.

Before a design can be optimized, you must define the environment in which the design is expected to operate. You define the environment by specifying operating conditions, wire load models, and system interface characteristics.

Operating conditions include temperature, voltage, and process variations. Wire load models estimate the effect of wire length on design performance. System interface characteristics include input drives, input and output loads, and Fanout loads. The environment model directly affects design synthesis results.

The design application for example may operate in a high temperature environment, or in a very cold one. The environment can be considered also as the around working chips and their effects of its input/output driving and loading, and create a variable voltage level environment.

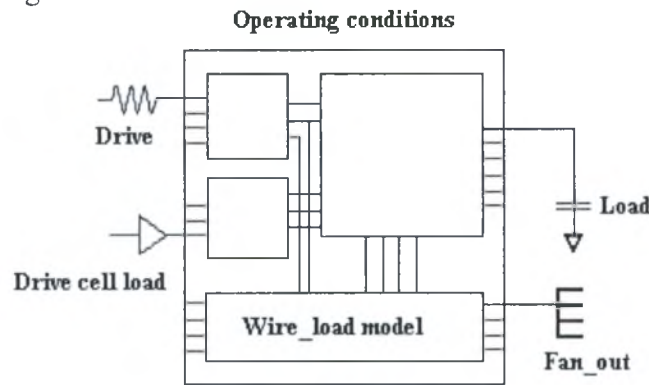


Figure3.7: Design environment parameters.

So the designer should have a good understanding of the design application environment, since the environment of the design may change also according to the place of application.

In addition to specifying the design environment, you must set design constraints before synthesizing the design. There are two categories of design constraints:

- Design rule constraints
- Design optimization constraints

Design rule constraints are supplied in the technology library you specify. They are referred to as the *implicit* design rules. These rules are established by the library vendor, and, for the proper functioning of the fabricated circuit, they must not be violated. You can, however, specify stricter design rules if appropriate. The rules you specify are referred to as the *explicit* design rules.

Design optimization constraints define timing and area optimization goals for synthesis tool and they are user-specified. Synthesis tool optimizes the synthesis of the design, in accordance with these constraints, but not at the expense of the design rule constraints. That is, synthesis tool attempts never to violate the higher-priority design rules.

Design rule constraints include:

- Transition time of a net, which is the time required for its driving pin to change logic values.
- Maximum Fanout load for a net, which is the maximum number of loads the net can drive.
- Maximum Capacitance of a net.

The design optimization constraints include the area constraints, power constraints, and timing constraints. A maximum area of the design is specified by the

designer. Also the consumed power can be restricted by the designer. The most important constraints normally are the timing constraints. Timing constraints include:

- Clock period.
- Maximum clock skew.
- Propagation or not of clock skew.
- Input and output delay relative to clock signal.
- Maximum path delay in combinational parts.

3.2.2 Synthesis process

After the designer set the synthesis setup elements, the synthesis tool can start synthesis process. Synthesis process flow includes two main steps; *netlist creation*, *netlist mapping*.

RTL design files provided to the synthesis tool are implemented in logic level, using generic gates. The RTL components are translated into Boolean equations which implemented using generic gates (like AND, OR, XOR, NOT, NOR, XNOR, NAND...). Before creating boolean equations, all possible RTL optimizations are performed by the synthesis tool, after that, boolean equations are created. A second level of optimizations is done on the boolean equations to obtain the minimal netlist, the applied constraints by the designer affect highly the boolean optimizations.

The second step after all those optimizations and netlist creation is mapping created generic gates to corresponding cells in the technology library.

As we discussed in optimization section, the last optimization step is sizing of gates and gate implementation style, which selected to meet the design constraints. However, an additional constraint on the design is that we don't have the freedom to select any sizes and implementation styles, a semiconductor vendor restrict the designers with the set of cells and gates he can fabricate successfully, so the semiconductor vendor, provides the designers with the set of all cells and gates he fabricates in a technology library as we saw in section (3.2.1). So what synthesis tool does is to map the netlist gates to corresponding ones in the technology library. For example; NAND gate in the technology library is implemented in 10 different sizes in static style and 5 different sizes in dynamic style, then the synthesis tool will map a NAND gate in the netlist to one of the 15 NAND implementations in the technology library, the selected implementation is affected by the design constraints and the library defined implementation characteristics.

A first run mapping is done to check the violation amounts and then mapping process is repeated to meet the constraints. If the first mapping process meets the design constraints, the synthesis tool finishes and the design is ready, else the mapping process is iterated around the critical paths, and repeated until constraints are met. If after amount of time mapping process always violates design constraints, the synthesis tool repeats optimization of RTL level and gates netlist, then restart

mapping. The designer could control the synthesis tool behaviour and synthesis process so he can get the best possible results.

3.3 Transform and Quantization Synthesis

In this section we will describe the synthesis flow and parameters we used in synthesizing Transform and Quantization phases, also we present the synthesis results and their analysis.

Synopsys design compiler is used to perform synthesis process. See appendix A for details about Synopsys design compiler.

The synthesis process target was to investigate the best possible implementation of the suggested ones as described in chapter2. A set of top-level design files are created for each different implementation, table3.1 shows the different top-level design file.

Design Files

Table3.1: Created top-level design files

Top-level File	Description
Transform_CONF.vhd	Implementation of transform unit using configurator.
Transform_No_CONF.vhd	Implementation of transform stage using separate 2×2 and 4×4 hadmard transform units.
Transform_N_HAD	Implementation of transform unit by implementing 4×4 hadmard transform independent from core transform, as described in section 2.1.4
Quantization_FM_TRM.vhd	Implementation of quantization stage using <i>f</i> -modification method and tree multiplier.
Quantization_FM_CSM.vhd	Implementation of quantization stage using <i>f</i> -modification method and carry-save multiplier.
Quantization_STM_TRM.vhd	Implementation of quantization stage using standard method and tree multiplier.
Quantization_STM_CSM.vhd	Implementation of quantization stage using standard method and carry-save multiplier.

The first three files are three different implementations of transform phase, the target is to see how much are the trade off parameters between transform implementation using configurator and another transform implementation by duplicating the first butterfly level of the 4×4 hadmard transform, and a third one with implementing 4×4 hadmard transform independently from core transform.

Created design files hierarchy for the first two implementations are shown in the figures3.8 and 3.9 respectively, for Third implementation we replace 4×4 hadmard transform subdesign in figure 3.9 with the one described in figure 3.10. The main difference between the two design files sets is the replacement of configurator structure by the 2×2 hadmard transform structure.

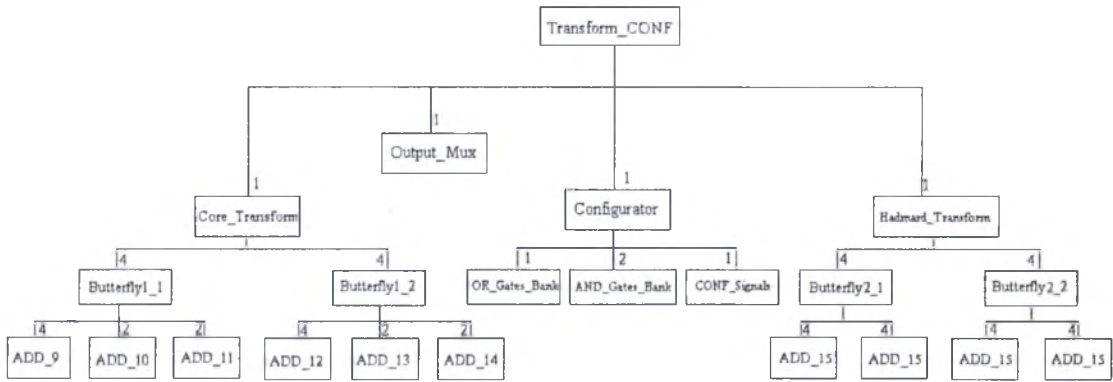


Figure3.8: Transform_CONF design files hierarchy structure.

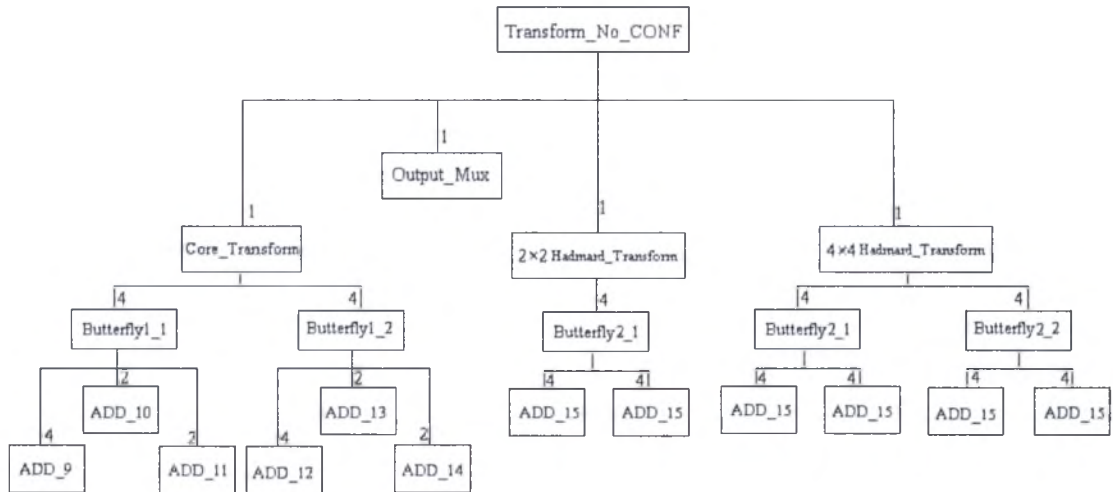


Figure3.9: Transform_No_CONF design files hierarchy structure.

The numbers on the edges represent the number of modules of the child component referenced in its parent. The design files are written in VHDL format.

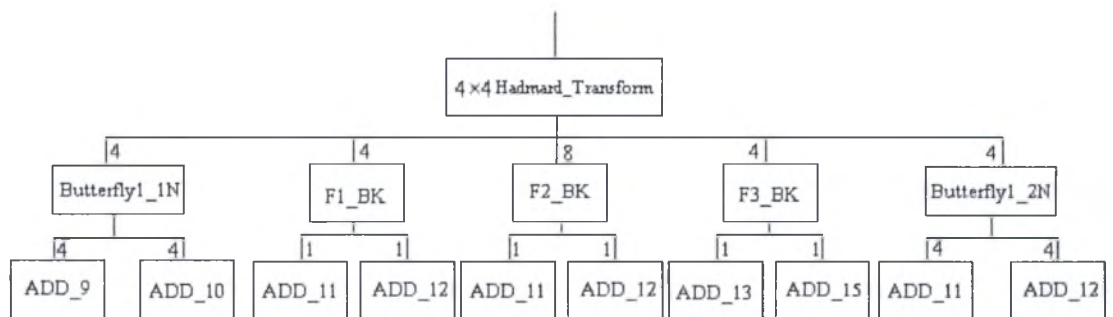


Figure3.10: 4x4 hadamard transform implementation in Transform_N_HAD subdesign.

The other 4 top-level files in table3.1 represent different implementations for the quantization phase. Two of them implement the standard method formulation for two different multiplier architectures (tree multiplier and carry-save multiplier), and the other two implement the suggested f -modification method also for two different multiplier architectures (tree multiplier and carry-save multiplier).

Figures 3.11 and 3.12 show the created design files hierarchy for f -modification and standard methods for the tree multiplier architecture respectively. The numbers on the edges represent the number of modules of the child component referenced in its parent. Take care that the adders used in standard method implementation are unsigned adders, don't let the name similarities disturb you.

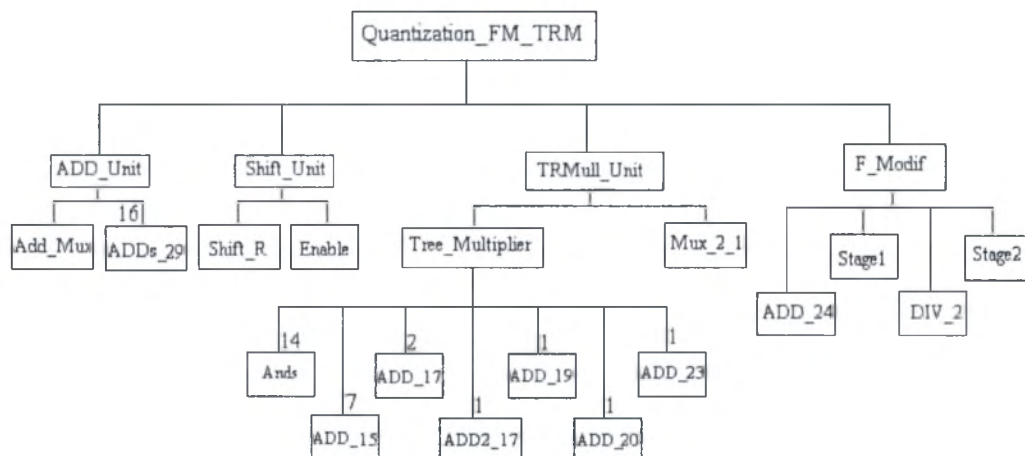


Figure3.11: Quantization_FM_TRM design files hierarchy structure.

For implementations using carry-save multiplier, just we replace the TRMull_Unit subdesign file with the CSMull_Unit subdesign file shown in figure3.13

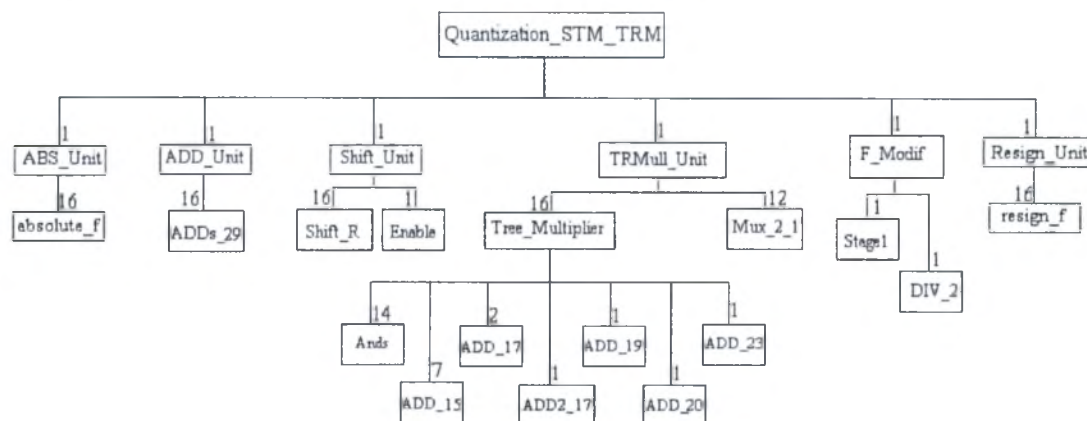


Figure3.12: Quantization_STM_TRM design files hierarchy structure.

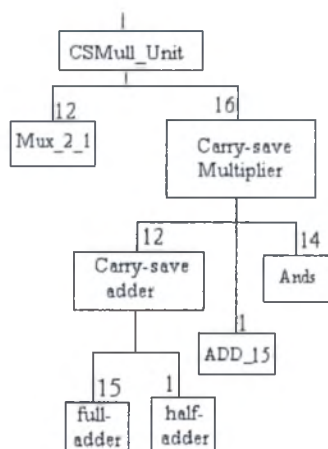


Figure3.13: CSMull_Unit design files hierarchy structure.

Libraries and design environment

We use the 0.13 μm semiconductor technology provided in the technology library “umce13h210t3_wc_108V_125C”. The worst operating conditions set is used; with temperature up to 125C° and at high voltage 1.08v. The used wire load model is described in table3.2:

Suggested wire load model is “10K”.

Table3.2: Wire load model.

Fanout	Length	Capacitance	Resistance	Area
1	7.93	0.00	0.01	0.00
2	17.72	0.00	0.01	0.00
3	29.08	0.01	0.02	0.00
4	35.72	0.01	0.03	0.00
5	60.05	0.02	0.05	0.00
6	63.96	0.02	0.05	0.00
7	94.43	0.02	0.06	0.00
17	610.15	0.15	0.33	0.00
37	1062.47	0.26	0.55	0.00

Design Constraints

- Default design rules provided by the technology library are used.
- The maximum combinational paths delay for each of quantization and transform units is set to 0ns to enforce design compiler get the best implementation.
- For area constraint 0 maximum area constraint is used to push design compiler performs the best for area optimization.
- The consumed power is unconstrained, so the compiler will minimize it as appropriate.

Compile strategy

For the forward path of transform and quantization processes, separate compile is done for each transform and quantization design file to get an overview of the best subdesigns combinations, and how much the costs variations will be. Three compiles of the three different transform unit designs and four compiles for the four different quantization unit designs. A top level file is created later from the best transform and quantization designs in addition to the quantization parameters subdesign. The Top level design is ungrouped and compiled in high effort with flatten effort high and phase apply is true. This strategy will enforce design compiler to compile cross hierarchy levels which will result in faster design.

3.3.1 Transform synthesis results

3.3.1.1 Transform_CONF :(Transform implementation using configurator).

Timing results:

The schematic shown in figure3.14 highlights the critical path. The maximum delay of data arrival is 6.8192 ns. Table3.3 shows the details of the critical path main parts.

Table3.3: critical path.

<i>Functional Unit</i>	<i>Input time (ns)</i>	<i>Output time (ns)</i>
Core Transform	0.00	2.2807
Configurator	2.2807	2.5586
Hadmard transform	2.5586	6.6664
Output Multiplexer	6.6664	6.8192

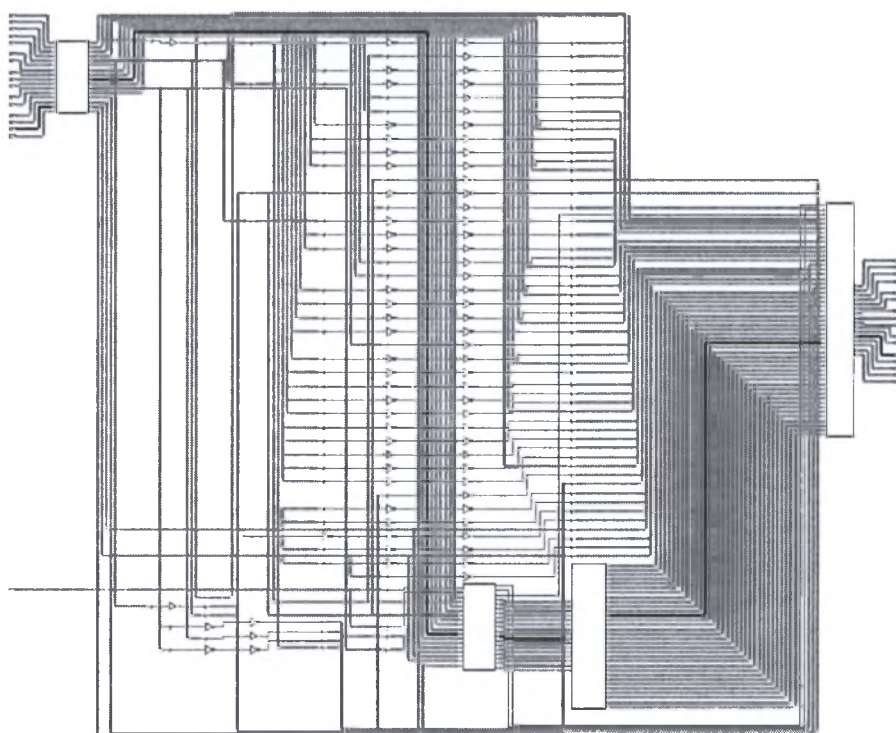


Figure3.14: Transform implementation with configurator schematic.

It is clear from the critical path that is the overhead of the configurator and multiplexer together isn't larger than 0.5 ns, while the most overhead (around 4 ns) is in the hadmard transform unit where 4 stages of 15-bit adders are used.

From the slack histogram shown below we can note that most the paths have 6.818 ns delay which is the best obtained implementation.

Area results:

The total area of the design is (178937.859375 area units). Most of the area overhead is in the Hadmard transform unit (42.78%) then in the core transform unit (34.32%). Table3.4 shows the design area details. The total number of cell used in the implementation is 11818 cells.

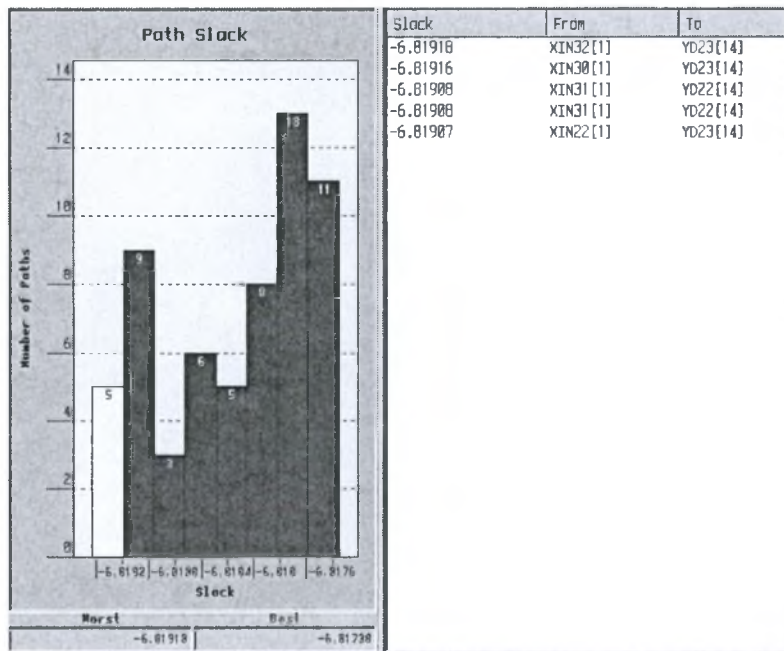


Figure3.15: full path slack histogram.

Table3.4: design area.

Subdesign	Total Area	Percent
Configurator	18473.859375	10.32%
Core Transform	61416.750000	34.32%
Hadnard Transform	76555.451562	42.78%
Output MUX	22491.790938	12.58%
<i>Total</i>	178937.859375	100%

Power results:

Table3.5: Total consumed power

	Cell Power	Net Switching Power	Total
Dynamic power	19.7657 mW	12.0076 mW	31.7733 mW
Leakage power	323.9445 uW	-	323.9445 uW

Table3.5 shows the total dynamic and leakage power consumed in the design. The most power is consumed in the hadnard unit and core unit.

Table3.6: cells' internal consumed power.

Functional unit	Power
Core unit	8.0346 mW
Configurator	2.0127 mW
Hadnard unit	7.0871 mW
Output multiplexer	2.6096 mW
<i>Total</i>	19.7657 mW

3.3.1.2 Transform_No_CONF: (Transform implementation without using configurator)

Timing results:

The schematic shown in figure3.16 highlights the critical path. The maximum delay of data arrival is 6.5677 ns. Table3.7 shows the details of the critical path main parts.

Table3.7: critical path.

<i>Functional Unit</i>	<i>Input time (ns)</i>	<i>Output time (ns)</i>
Core Transform	0.00	2.6301
4×4 Hadmard Transform	2.6301	6.4361
Output Multiplexer	6.4361	6.5677

From the critical path we can notice the effect of removing the configurator, which is around 0.25 ns. However still the overhead is on the 4×4 hadmard path where we have again 4 stages of 15-bit adders which consume around 3.8 ns.

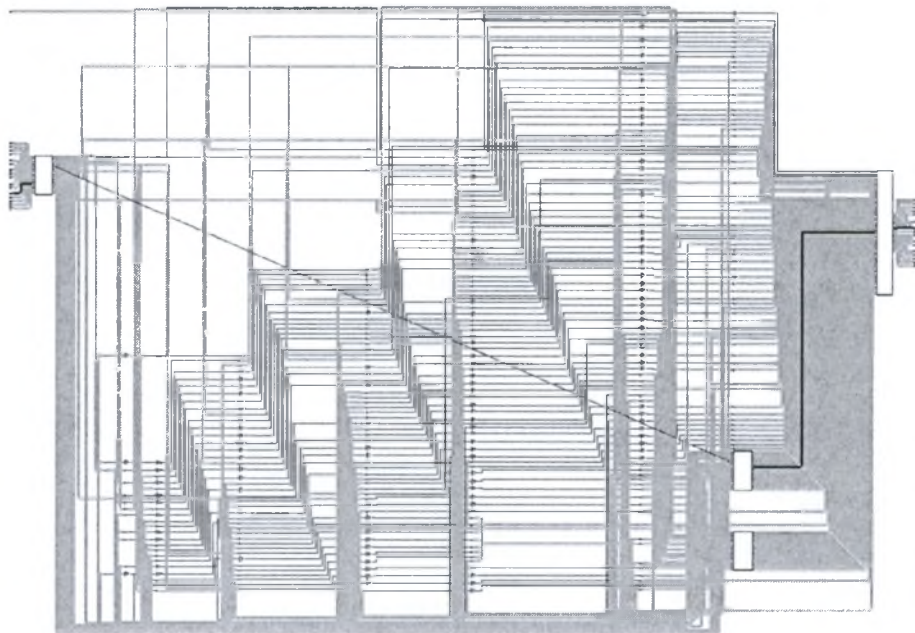


Figure3.16: Transform implementation without configurator schematic.

From the slack histogram shown in figure1.17 we can note that most the paths have 6.566 ns delay which is the best obtained implementation.

Area results:

The total area of the design is (179554.750000 area units). Most of the area overhead is in the 4×4 hadmard transform units (42.00%) then in the core transform unit (32.56%). Table3.8 shows the design area details. The total number of cell used in the implementation is 14137 cells.

Table3.8: design area.

Subdesigns	Total Area	Percent
2×2 Hadmard Transform	25341.140625	14.11%
Core Transform	58454.812500	32.56%
4×4 Hadmard Transform	75408.179688	42.00%
Output MUX	18722.830078	11.33%
<i>Total</i>	179554.750000	100%

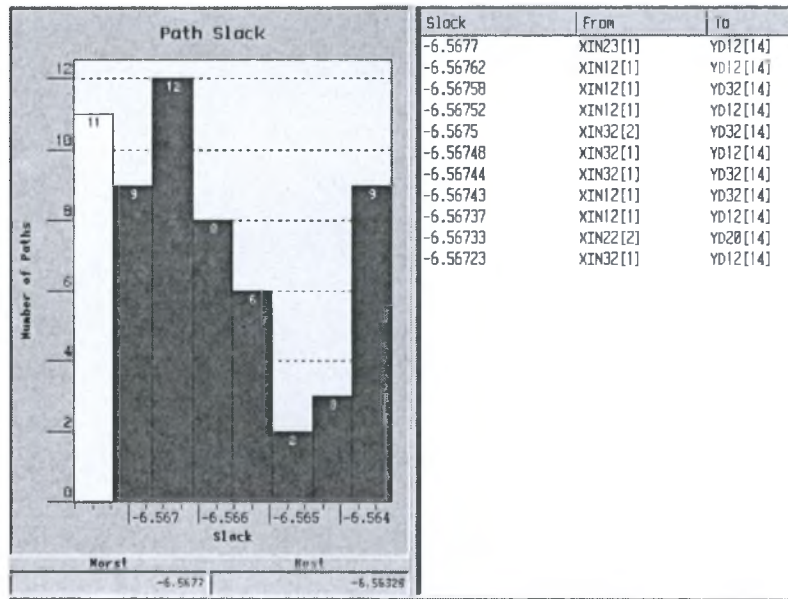


Figure3.17: full path slack histogram.

Power results:

Table3.9: Total consumed power.

	Cell Power	Net Switching Power	Total
Dynamic power	20.7730 mW	17.1418 mW	37.9148 mW
Leakage power	278.0365 uW	-	278.0365 uW

Table3.9 shows the total dynamic and leakage power consumed in the design. Again the most power is consumed in the hadmard units and core unit.

Table3.10: cells' internal consumed power.

Functional unit	Power
Core unit	7.5674 mW
2×2 Hadmard unit	2.1679 mW
4×4 Hadmard unit	8.8490 mW
Output multiplexer	2.1887 mW
<i>Total</i>	20.773 mW

3.3.1.3 Transform_N_HAD: (Transform implementation with 4×4 hadmard transform implemented independently from core transform)

Timing results:

The schematic shown in figure3.18 highlights the critical path. The maximum delay of data arrival is 5.30 ns. Table3.11 shows the details of the critical path main parts.

Table3.11: critical path.

<i>Functional Unit</i>	<i>Input time (ns)</i>	<i>Output time (ns)</i>
Core Transform	0.00	3.06533
2×2 Hadamard Transform	3.06533	4.99024
Output Multiplexer	4.99024	5.30133

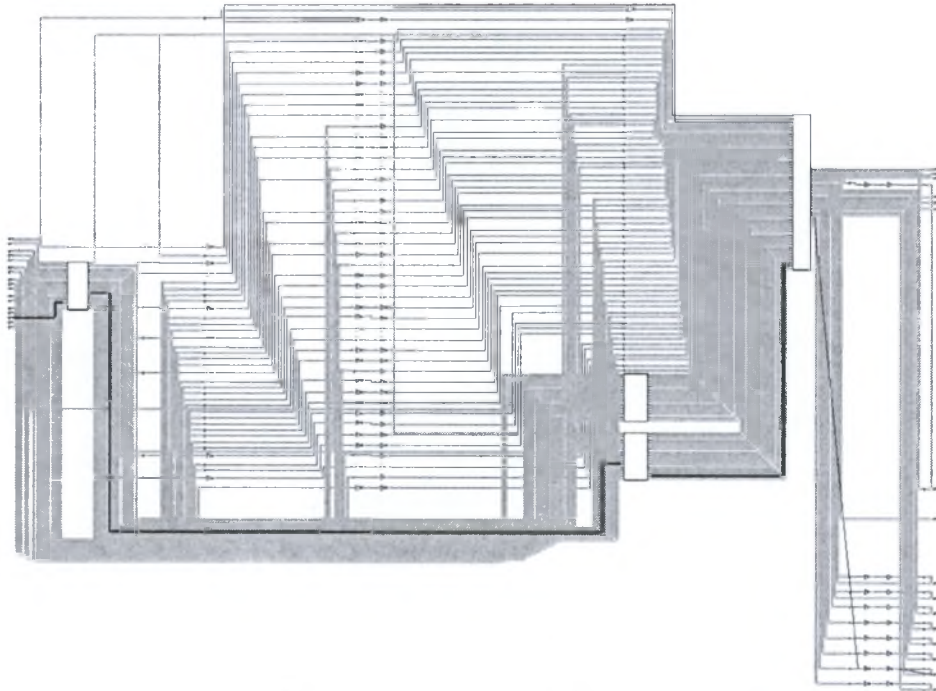


Figure3.18: Transform implementation 4×4 hadamard transform independently of core transform.

From the slack histogram shown in figure1.19 we can note that most the paths have 5.3 ns delay which is the best obtained implementation.

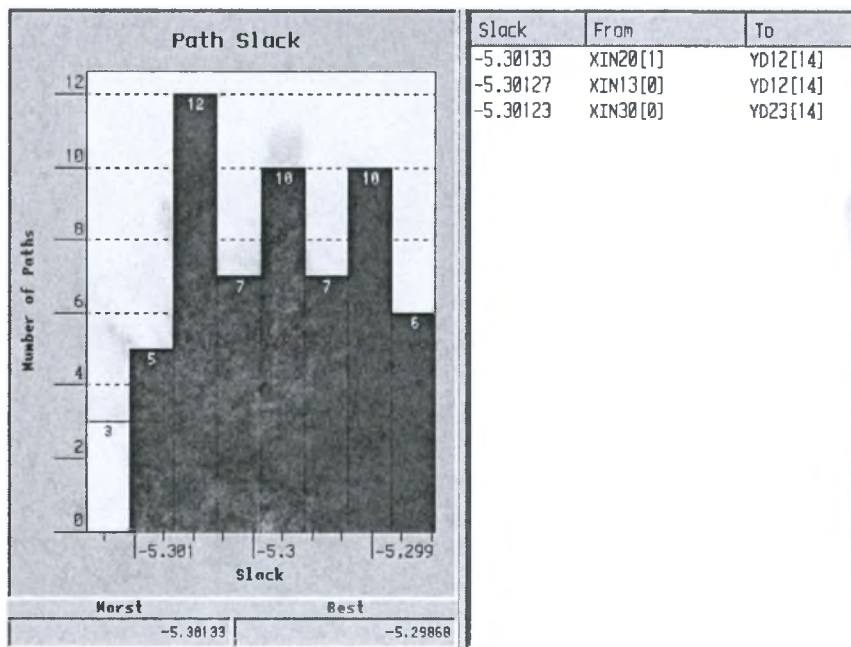


Figure3.19: full path slack histogram.

Area results:

The total area of the design is (186347.515625 area units). Most of the area overhead is in the 4×4 hadmard transform units (42.61%) then in the core transform unit (31.91%). Table3.12 shows the design area details. The total number of used cells is 13369.

Table3.12: design area.

Subdesign	Total Area	Percent
2×2 Hadmard Transform	26133.90625	14.02%
Core Transform	59454.812500	31.91%
4×4 Hadmard Transform	79408.179688	42.61%
Output MUX	19722.830078	11.46%
<i>Total</i>	186347.515625	100%

Power results:

Table3.13: Total consumed power.

	Cell Power	Net Switching Power	Total
Dynamic power	27.7604 mW	18.9453 mW	46.7057 mW
Leakage power	303.4926 uW	-	303.4926 uW

Table3.13 shows the total dynamic and leakage power consumed in the design. Again the most power is consumed in the hadmard units and core unit.

Table3.14: cells' internal consumed power.

Functional unit	Power
Core unit	9.9874 mW
2×2 Hadmard unit	3.1909 mW
4×4 Hadmard unit	11.3894 mW
Output multiplexer	3.1927 mW
<i>Total</i>	27.7604 mW

3.3.1.4 Comparison

As expected, the delay difference between Transform implementation with configurator and transform implementation without it isn't large (0.25 ns). However, the improvement with implementing 4×4 hadmard transform independently from core transform result, we will save a round 1.3 ns.

	Transform CONF	Transform No CONF	Transform N HAD
Delay (ns)	6.82	6.57	5.3
Area	178937.859375	179554.750000	186347.515625
Cells#	11818	14137	13369
Dynamic power	31.7733 mW	37.9148 mW	46.7057 mW
Leakage power	323.9445 uW	278.0365 uW	303.4926 uW

The area of the configurator and the additional nets in the hadmard transform unit is still less than duplication area of the first hadmard butterfly level. And as we

expected, the area of implementation of Transform_N_HAD is larger even the number of used cells is less.

3.3.2 Quantization Synthesis results

3.3.2.1 Quantization_STM_TRM: Quantization implementation in standard method using tree multiplier.

Timing results:

The schematic shown in figure3.20 highlights the critical path. The maximum delay of data arrival is 8.631 ns. Table3.15 shows the details of the critical path main parts.

Table3.15: Critical path.

<i>Functional Unit</i>	<i>Input time (ns)</i>	<i>Output time (ns)</i>
Absolute value unit	0.00	0.66790
Multiplication unit	0.66790	6.08816
Addition unit	6.08816	7.21715
Shift unit	7.21715	7.78249
Resign unit	7.78249	8.63065

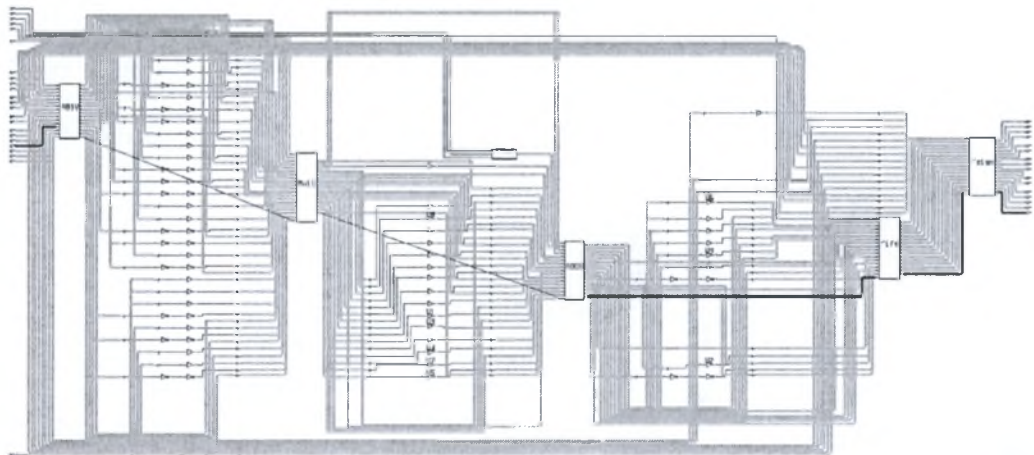


Figure3.20: Quantization implementation in standard method using tree multiplier schematic.

The most delay cost is in the multiplication unit which takes around 5.4 ns. Where the critical path of the multiplication unit consists of multiplexer stage, partial products stage, 15-bit adder, 17-bit adder, 19-bit adder, and 23-bit adder. Also, the use of absolute and resign stages costs around 1.5 ns.

From the slack histogram shown in figure3.21 we can note that most the paths have 8.628 ns delay which is the best obtained implementation.

Area results:

The total area of the design is (412912.500000 area units). Most of the area overhead is in the multiplication unit (78.64%) then in the shift unit (8.51%). Table3.16 shows the design area details. The total number of used cells is 33881.

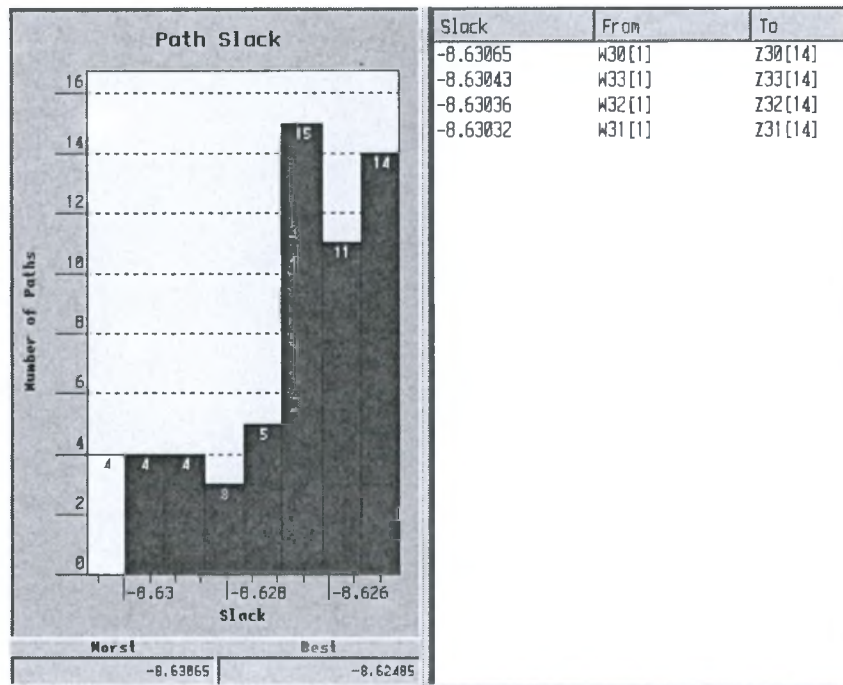


Figure3.21: full path delay slack histogram.

Table3.16: design area.

Subdesign	Total Area	Percent
ABS Unit	16424.640625	4%
ADD_UnitU	18302.197266	4.43%
Fs_Modif	832.895996	0.2%
Resign Unit	17421.695312	4.22%
SShift_Unit	35133.183594	8.51%
TRMull_UnitU	320096.000000	78.64%
<i>Total</i>	412912.500000	100.0%

Power results:

Table3.17 shows the total dynamic and leakage power consumed in the design. Again the most power is consumed in the multiplication unit.

Table3.17: Total consumed power.

	Cell Power	Net Switching Power	Total
Dynamic power	25.6178 mW	15.3214 mW	40.9392 mW
Leakage power	409.9400 uW	-	409.9400 uW

The dynamic power consumed in cell units is distributed over the cells as below:

Table3.18: cells' internal consumed power.

Functional unit	Power
ABS Unit	2.0466 mW
ADD_UnitU	3.8363 mW
Fs_Modif	0.5365 mW
Resign Unit	1.1158 mW
SShift_Unit	1.5548 mW
TRMull_UnitU	16.5278 mW
<i>Total</i>	25.6178 mW

3.3.2.2 Quantization_STM_CSM: Quantization implementation in standard method using carry-save multiplier.

Timing results:

The schematic shown in figure3.20 highlights the critical path. The maximum delay of data arrival is 9.22 ns. Table3.19 shows the details of the critical path main parts.

Table3.19: Critical path.

Functional Unit	Input time (ns)	Output time (ns)
Absolute value unit	0.00	1.26252
Multiplication unit	1.26252	6.98347
Addition unit	6.98347	8.15038
Shift unit	8.15038	8.63427
Resign unit	8.63427	9.21604

Again, the most critical path cost is in the multiplication unit with 5.7 ns, and absolute and resign units cost around 1.6 ns.

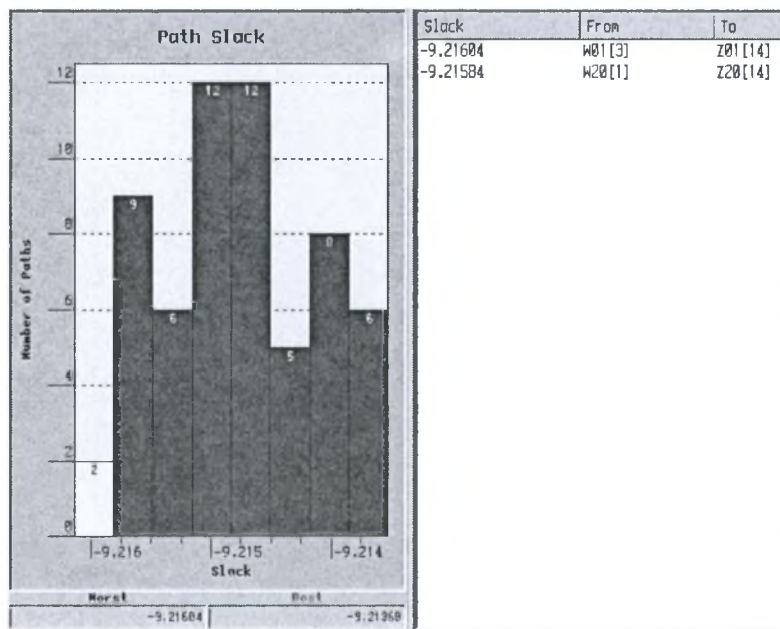


Figure3.22: full path delay slack histogram.

From the slack histogram shown in figure3.22 we can note that most the paths have 9.215 ns delay which is the best obtained implementation.

Area results:

The total area of the design is (336792.375000 area units). Most of the area overhead is in the multiplication unit (54.00%) then in the Shift unit (24.955%). Table3.20 shows the design area details. The total number of cells used is 23058.

Table3.20: design area.

Subdesign	Total Area	Percent
ABS Unit	16280.5434075	4.834%
ADD_UnitU	32012.11524375	9.505%
Fs_Modif	1956.76369875	0.581%
Resign Unit	20628.53296875	6.125%
SShift_Unit	84046.53718125	24.955%
CSMull_UnitU	18186788.25	54.00%
<i>Total</i>	<i>336792.375000</i>	<i>100%</i>

Power results:

Table3.21 shows the total dynamic and leakage power consumed in the design. Again the most power is consumed in the multiplication unit.

Table3.21: Total consumed power.

	Cell Power	Net Switching Power	Total
Dynamic power	28.4720 mW	23.0667 mW	51.5387 mW
Leakage power	584.6456 uW	-	584.6456 uW

The dynamic power consumed in cell units is distributed over the cells as below:

Table3.22: cells' internal consumed power.

Functional unit	Power
ABS Unit	2.27 mW
ADD_UnitU	3.12 mW
Fs_Modif	0.422 mW
Resign Unit	1.30 mW
SShift_Unit	2.61 mW
CSMull_UnitU	18.75 mW
<i>Total</i>	<i>28.4720 mW</i>

3.3.2.3 Quantization_FM_TRM: Quantization implementation in f -modification method using tree multiplier.

Timing results:

The schematic shown in figure3.23 highlights the critical path. The maximum delay of data arrival is 6.382 ns. Table3.23 shows the details of the critical path main parts.

Table3.23: Critical path.

Functional Unit	Input time (ns)	Output time (ns)
Multiplication unit	0.00	5.3725
Addition unit	5.3725	6.1550
Shift unit	6.1550	6.3815

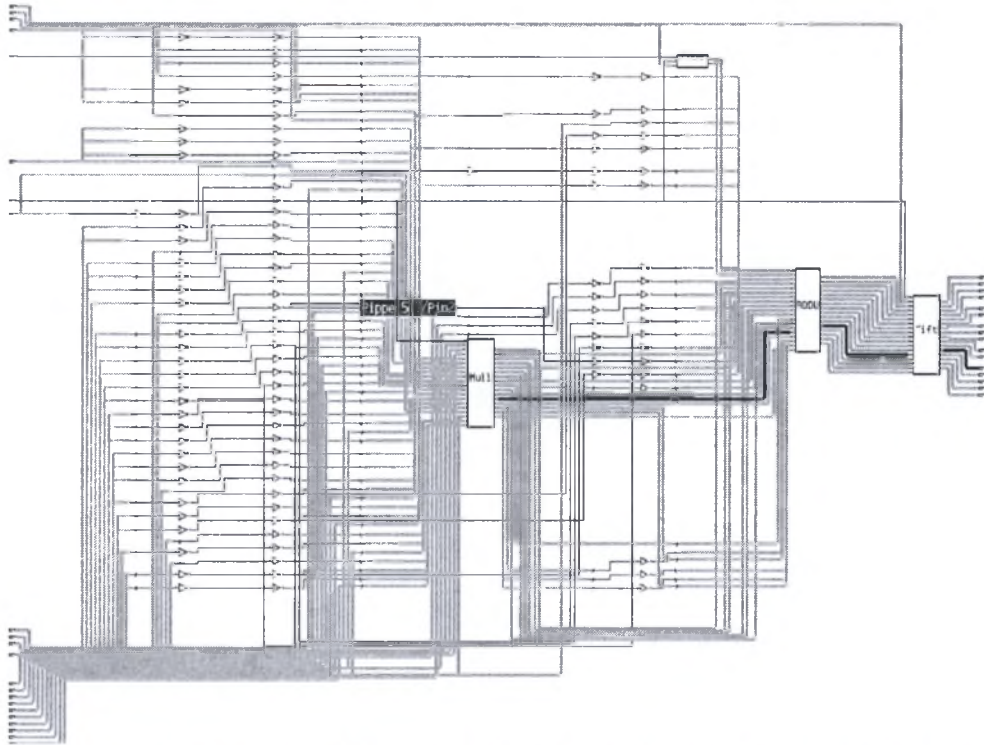


Figure3.23: Quantization implementation in f -modification method using tree multiplier schematic.

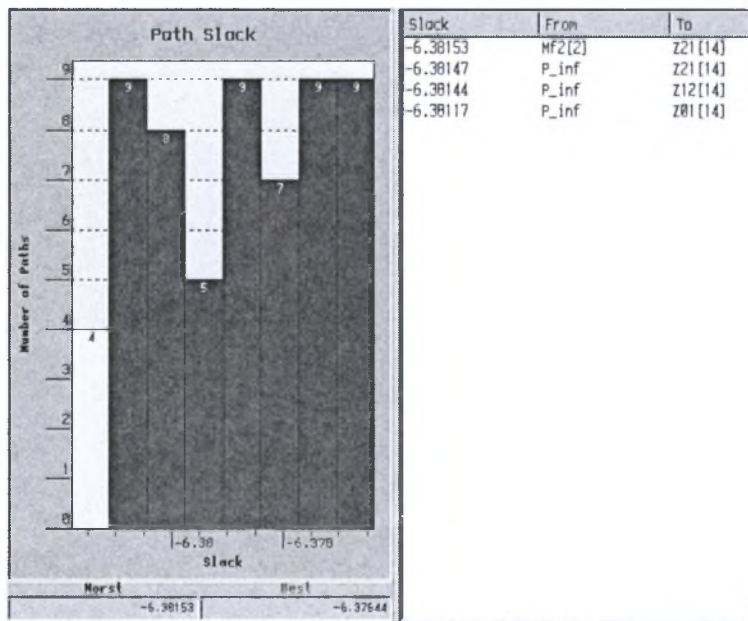


Figure3.24: full path delay slack histogram.

Around 84% of the critical path cost is in the multiplication unit which cost around 5.37 ns, while the addition and shift unit don't cost larger than 1.1 ns.

From the slack histogram shown in figure3.24 we can note that most the paths have 6.38 ns delay which is the best obtained implementation

Area results:

The total area of the design is (418205.375000 area units). Most of the area overhead is in the multiplication unit (71.03%) then in the Shift Unit (18.16%). Table3.24 shows the design area details. The total number of used cells is 32247.

Table3.24: design area.

Reference	Total Area	Percent
ADD_Unit	40766.972656	9.75%
F_Modif	4437.504883	1.06%
Shift_Unit	75929.546875	18.16%
TRMull_Unit	296978.03125	71.03%
<i>Total</i>	418205.375000	100.0%

Power results:

Table3.25 shows the total dynamic and leakage power consumed in the design. Again the most power is consumed in the multiplication unit.

Table3.25: Total consumed power.

	Cell Power	Net Switching Power	Total
Dynamic power	20.9847 mW	16.4189 mW	37.4036 mW
Leakage power	349.1045 uW	-	349.1045 uW

The dynamic power consumed in cell units is distributed over the cells as below:

Table3.26: cells' internal consumed power.

Functional unit	Power
ADD_Unit	3.2959 mW
F_Modif	0.5025 mW
Shift_Unit	0.8031 mW
TRMull_Unit	16.2832 mW
<i>Total</i>	20.9847 mW

3.3.2.4 Quantization_FM_CSM: Quantization implementation in *f*-modification method using carry-save multiplier.

Timing results:

The schematic shown in figure3.25 highlights the critical path. The maximum delay of data arrival is 6.98 ns. Table shows the details of the critical path main parts.

Table3.27: Critical path.

Functional Unit	Input time (ns)	Output time (ns)
Multiplication unit	0.00	6.34
Addition unit	6.34	6.97
Shift unit	6.97	6.98

From the slack histogram shown figure3.26 we can note that most the paths have 6.975 ns delay which is the best obtained implementation.

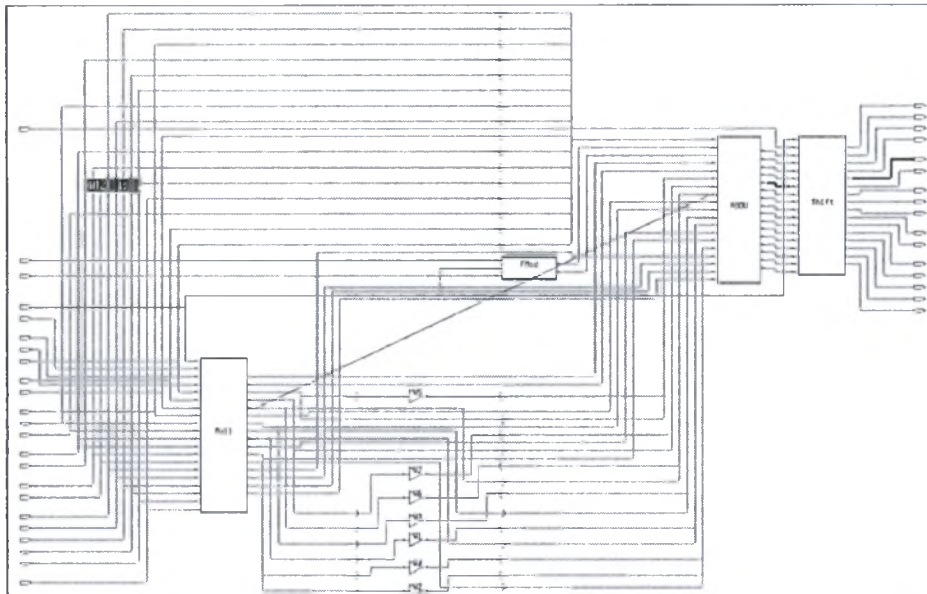


Figure3.25: Quantization implementation in *f*-modification method using carry-save multiplier schematic.

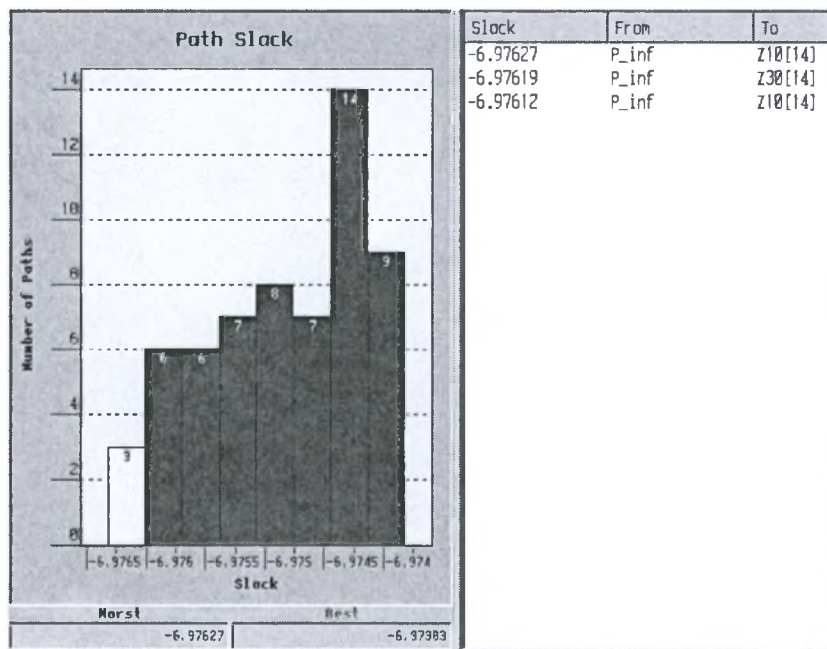


Figure3.26: full path delay slack histogram.

Area results:

The total area of the design is (326448.562500 area units). Most of the area overhead is in the multiplication unit (62.48%) then in the shift unit (22.9%). Table3.28 shows the design area details. The total number of used cells is 22178.

Table3.28: design area.

Subdesign	Total Area	Percent
ADD Unit	43645.796875	13.37%
F Modif	4072.894287	1.25%
Shift Unit	74742.921875	22.9%
CSMull Unit	203933.312500	62.48%
<i>Total</i>	326448.562500	100.0%

Power results:

Table3.29 shows the total dynamic and leakage power consumed in the design. Again the most power is consumed in the multiplication unit.

Table3.29: Total consumed power

	Cell Power	Net Switching Power	Total
Dynamic power	25.9138 mW	21.8868 mW	47.8006 mW
Leakage power	555.1185 uW	-	555.1185 uW

The dynamic power consumed in cell units is distributed over the cells as below:

Table3.30: cells' internal consumed power.

Functional unit	Power
ADD Unit	3.3090 mW
F Modif	0.3856 mW
Shift Unit	1.9020 mW
CSMull Unit	20.3072 mW
<i>Total</i>	25.9138 mW

3.3.2.5 Quantization phase synthesis results comparison

	Standard Method		<i>f</i> -Modification Method	
	Tree Multiplier	Carry-save Multiplier	Tree Multiplier	Carry-save Multiplier
Delay (ns)	8.63065	9.21604	6.3815	6.98
Area	412912.500	336792.375	418205.375	326448.562
Cells#	33881	23058	32247	22178
Dynamic Power.	40.9392 mW	51.5387 mW	37.4036 mW	47.8006 mW
Leakage Power.	409.9400 uW	584.6456 uW	349.1045 uW	555.1185 uW

From the above table there are three conclusions can be taken:

- Cost of *f*-modification method implementation is less than cost of standard method implementation.
- Cost of implementation using tree multiplier architecture is less than cost of implementation except for area overhead of tree multiplier that can be considered high compared with carry-save multiplier implementation.
- Quantization implementation in *f*-modification method and using tree multiplier is the best solution.

It is clear how much removing absolute and resign stages from the implementation of quantization process will save time. There is around 2.2 ns save in time from Standard method to *f*-modification method.

Also the area overhead of absolute and resign stages is eliminated in the *f*-modification, and can be used to recover the increase in the multiplication unit when using tree multiplier.

The last benefit of removing absolute and resign units is the noticeable decrease in consumed power for the case of dynamic and leakage power.

Even the carry-save multiplier saves the carry calculations, but the number of necessary stages to perform partial products addition creates overhead on the critical path. On the contrary, even in the tree multiplier the adders' size increase as the progress in partial products addition, the smaller number of needed stages to perform partial products addition reduces the overhead on the critical path. This can be noted clearly from the table.

However the increase in the area of the tree multiplier can be as critical as clear from the table where it is around 90000 area units, and an increase in the number of cells used around 10000 cells.

3.3.3 Transform and Quantization Integration

In this section we will compile the transform unit and quantization unit as one top-level design. Since tree multiplier that it is the best. Two top level designs will be tested; the first one consists of transform unit built in subdesign file Transform_N_HAD and quantization unit built in f -modification method using tree multiplier (Tr_QuFTRM_Phases). While the second one consists of transform unit built in subdesign file Transform_N_HAD and quantization unit built in standard method using tree multiplier (Tr_QuSTRM_Phases).

3.3.3.1 Tr_QuFTRM_Phases

The schematic shown in figure3.27 highlights the critical path. The maximum delay of data arrival is 8.78 ns.

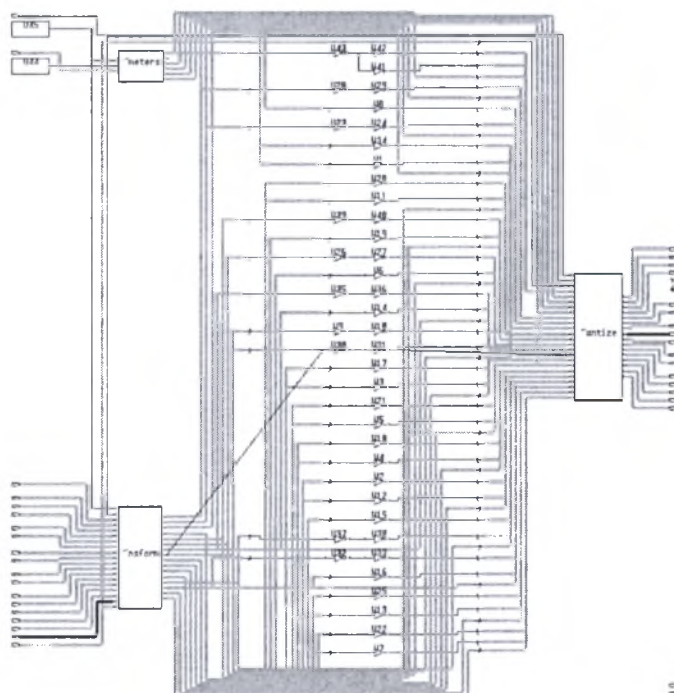


Figure3.27: Tr_QuFTRM_Phases schematic.

Table3.31 shows the details of the critical path main parts. It is clear from the critical path table that compiling the transform and quantization subdesigns together (top-down compile strategy) will give better results.

Table3.31: Critical path.

<i>Functional Unit</i>	<i>Input time (ns)</i>	<i>Output time (ns)</i>
Transform Unit	0.00	4.62
Quantization Unit	4.62	8.78

From the slack histogram shown in figure3.28 we can note that most the paths have 8.78 ns delay which is the best obtained implementation. In this implementation of the transform and quantization subdesigns the system could work on clock rate 113.9 MHz.

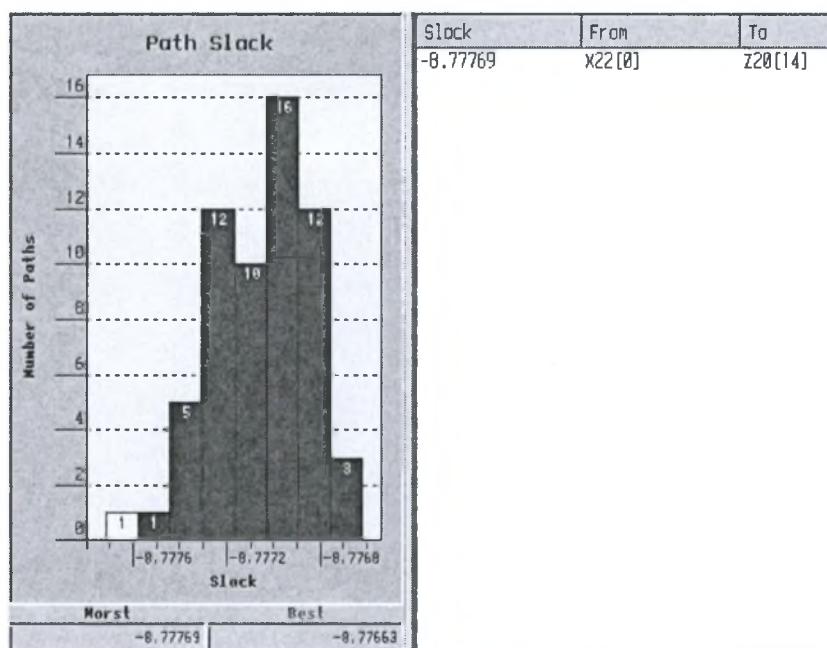


Figure3.28: Full path delay histogram.

Area results:

The total area of the design is (788451.812500 area units). Most of the area overhead is in the quantization unit (68.68%) then in the transform unit (30.93%). Table3.32 shows the design area details. The total number of used cells is 56524.

Table3.32: design area.

Subdesign	Total Area	Percent
Transform_N_HAD	243868.14561	30.93%
Quant_Param	3074.9621	0.39%
Quantization_FM_TRM	541508.71	68.68%
<i>Total</i>	788451.812500	100.00%

Power results:

Table3.33 shows the total dynamic and leakage power consumed in the design.

Table3.33: Total consumed power

	Cell Power	Net Switching Power	Total
Dynamic power	30.7053 mW	25.9731 mW	56.6784 mW
Leakage power	1.0302 mW	-	1.0302 mW

The dynamic power consumed in cell units is distributed over the cells as below:

Table3.34: cells' internal consumed power.

Functional unit	Power
Transform_N_HAD	12.3564 mW
Quant Param	0.1354 mW
Quantization FM TRM	18.0675 mW
Other cells	0.146 mW
<i>Total</i>	20.1372 mW

3.3.3.2 Tr_QuSTRM_Phases

Timing results:

The schematic shown in figure3.27 highlights the critical path. The maximum delay of data arrival is 11.312 ns. Table3.35 shows the details of the critical path main parts.

Table3.35: Critical path.

Functional Unit	Input time (ns)	Output time (ns)
Transform Unit	0.00	5.183
Quantization Unit	5.183	11.312

From the slack histogram shown below we can note that most the paths have 11.31 ns delay which is the best obtained implementation. In this implementation of the transform and quantization subdesigns the system could work on clock rate 88.4MHz.

Area results:

The total area of the design is (845750.562500 area units). Most of the area overhead is in the quantization unit (72.2653%) then in the transform unit (27.2125%). Table3.36 shows the design area details. The total number of used cells is 61998.

Table3.36: design area.

Reference	Total Area	Percent
Transform_CONF	230149.87182	27,2125%
Quant_Param	4416.509437375	0.5222%
Quantization_STM_TRM	611184.181242	72,2653%
<i>Total</i>	845750.562500	100.0%

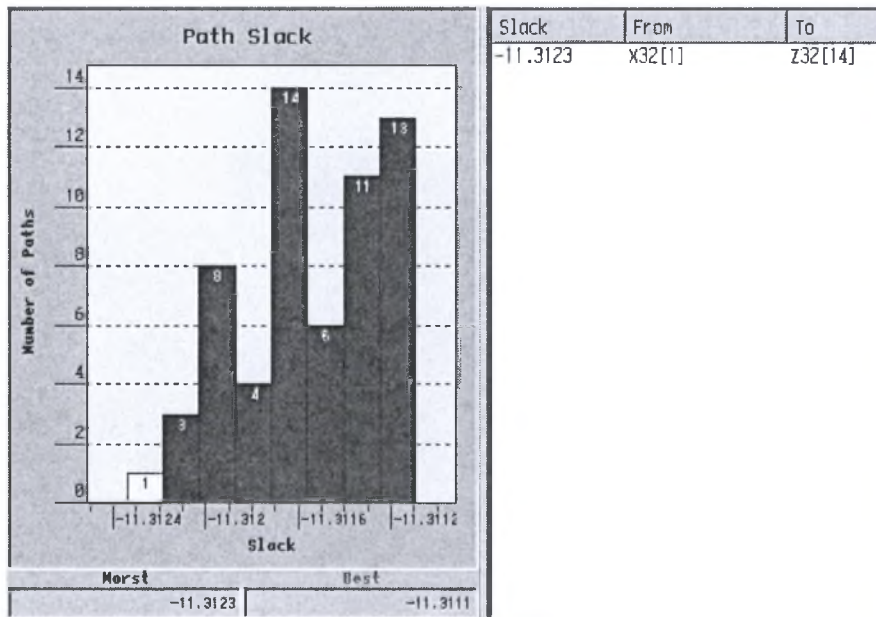


Figure3.29: full path delay slack histogram.

Power results:

Table3.37 shows the total dynamic and leakage power consumed in the design.

Table3.37: Total consumed power

	Cell Power	Net Switching Power	Total
Dynamic power	37.6985 mW	22.7645 mW	60.4630 mW
Leakage power	632.5847 uW	-	632.5847 uW

The dynamic power consumed in cell units is distributed over the cells as below:

Table3.38: cells' internal consumed power.

Functional unit	Power
Transform_CONF	11.2026 mW
Quant_Param	4.0446 mW
Quantization_STM_TRM	22.2495 mW
<i>Total</i>	37.6985 mW

3.3.3.3 Comparison

	Tr QuSTRM Phases	Tr QuFTRM Phases
Delay (ns)	11.312	8.78
Area	845750.562500	788451.8125
Cells#	61999	56524
Dynamic power	60.4630 mW	56.6784 mW
Leakage power	632.5847 uW	1.0302 mW

As clear from the table above and as we expected from the results of quantization and transform phases separate compile, the implementation of transform and quantization process using f-modification method and tree multiplier, will meet our targets, and gives the best expected results. The critical path delay is 8.78 which will

enable use of 113.9MHz clock sufficiently, with acceptable results for area and power consumption.

3.3.4 Inverse Transform and Quantization Synthesis

Timing results:

The schematic shown in figure3.30 highlights the critical path. The maximum delay of data arrival is 8.8 ns. Table3.39 shows the details of the critical path main parts.

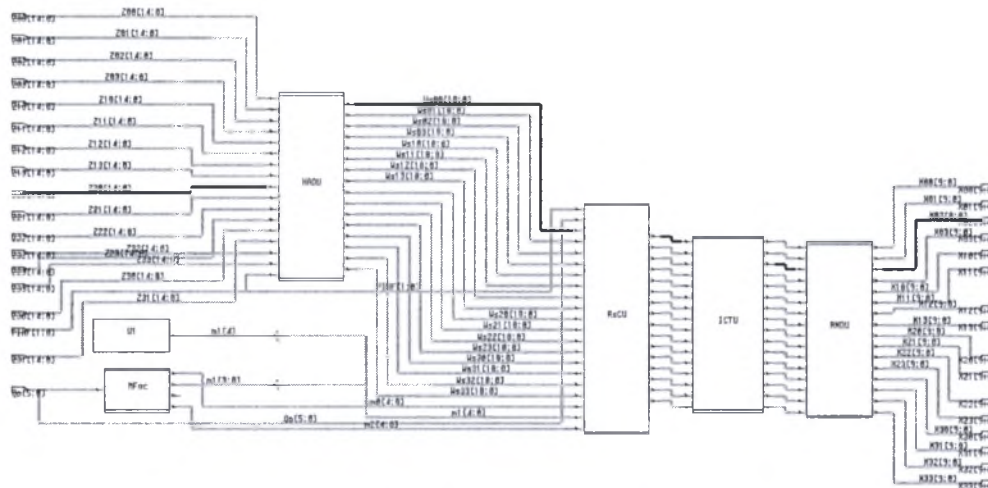


Figure3.30: Inverse_TR_Qu schematic.

Table3.39: Critical path.

<i>Functional Unit</i>	<i>Input time (ns)</i>	<i>Output time (ns)</i>
Hadnard Unit	0.00	3.6963
Rescale Unit	3.6963	5.8050
Inverse Core Unit	5.8050	8.5077
Round Unit	8.5077	8.7989

From the slack histogram shown in figure3.31 we can note that most the paths have 8.798 ns delay which is the best obtained implementation.

Area results:

The total area of the design is (560350.625000 area units).

Table3.40: design area.

Subdesign	Total Area	Percent
Hadnard Unit	172139.712	30.72%
Rescale Unit	287740.0459375	51.35%
Mull Factors Unit	1008.631125	0.18%
Inverse Core Unit	95259.60625	17.00%
Round Unit	4202.6296875	0.75%
<i>Total</i>	560350.625000	100.00%

Most of the area overhead is in the Rescale unit (51.35%) then in the Hadnard unit (30.72%). Table3.40 shows the design area details. The total number of used cells is 39985.

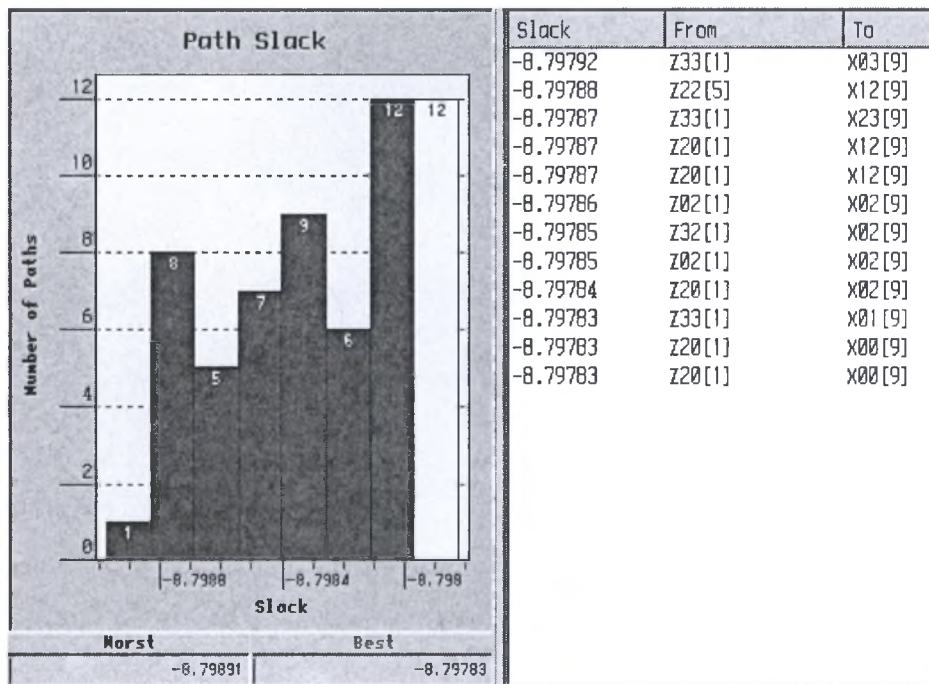


Figure3.31: full path delay slack histogram.

Power results:

Table3.41 shows the total dynamic and leakage power consumed in the design.

Table3.41: Total consumed power

	Cell Power	Net Switching Power	Total
Dynamic power	20.8175 mW	16.6068 mW	37.4244 mW
Leakage power	799.2933 uW	-	799.2933 uW

The dynamic power consumed in cell units is distributed over the cells as below:

Table3.42: cells' internal consumed power.

Functional unit	Power
Hadnard Unit	7.771 mW
Mull Factors Unit	0.066 mW
Rescale Unit	8.875 mW
Inverse Core Unit	4.004 mW
Round Unit	0.1015 mW
<i>Total</i>	20.8175 mW

3.4 Conclusion and Final Results

Implemented forward path hardware deals with 4×4 input data block with each data element is 9-bit signed integer, with the most significant bit is the sign bit. The output of the forward path is 4×4 quantized data block with each element is 15-bit signed integer with the most significant bit is the sign bit. QP, P_INF and PRED inputs are as described in section 2.1.

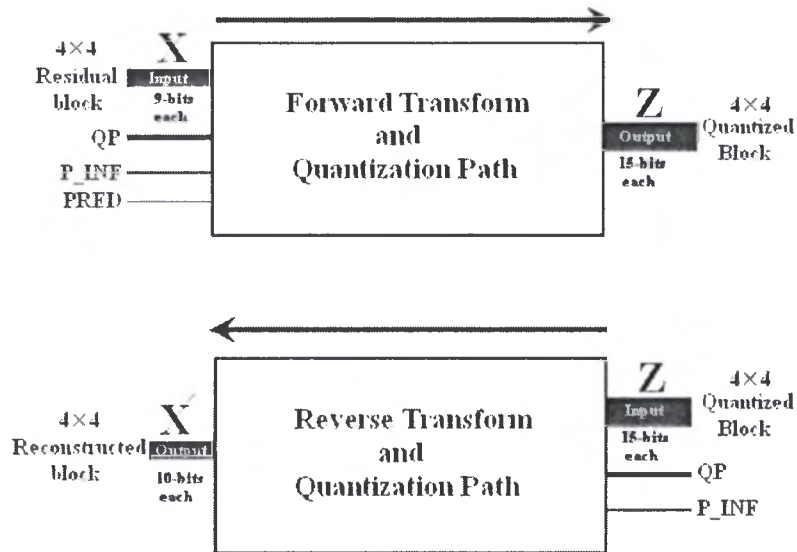


Figure3.32: Forward and reverse paths inputs and outputs.

Reverse path takes the resulting 4x4 quantized data block from the forward path output and QP, and P_INF inputs. The output of the reverse path is 4x4 reconstructed data block, with each element are 10-bits signed integer, with the most significant bit is sign bit.

A total of 393 ports are needed in the forward path, and 408 ports are necessary for reverse path.

At the end we can say that forward transform and quantization path could work with worst delay of 8.78ns, with a clock rate around 113.9 MHz. 56524 cells are used to realize the implementation, with estimated occupied area around 788451 area units of technology 130 nm area units. Estimated consumed dynamic power is around 60 mW, and around 1mW static power is consumed.

	Forward Path	Inverse Path
Delay (ns)	8.78	8.8
Clock Rate (MHz)	113.9	113.6
Dynamic power (mW)	56.6784	37.4244
Leakage power (uW)	1030.2	799.2933
Area	788451	560350
Cells#	56524	39985
Nets#	56677	40233
Ports#	393	408

The reverse path (inverse transform and inverse quantization) could work with worst delay of 8.8 ns, which allow use of clock rate 113.6 MHz. just 39985 cells are used to realize the implementation, with estimated occupied area around 560350 area units. Estimated consumed dynamic power is around 40mW, and a static power around 0.8mW.

From the above result we could say that the forward and reverse path can implemented the two paths together with a worst delay around 17.6 ns, and so could run with clock rate 56.8MHz, or a pipelining of the two stages can be done, with clock rate is 113MHz.

Appendix B

Synopsys Design Compiler

Synopsys provides an integrated RTL synthesis solution. Using Design Compiler tools, you can

- Produce fast, area-efficient ASIC designs by employing user-specified gate-array, FPGA, or standard-cell libraries.
- Translate designs from one technology to another.
- Explore design tradeoffs involving design constraints such as timing, area, and power under various loading, temperature, and voltage conditions.
- Synthesize and optimize finite state machines, including automatic state assignment and state minimization.
- Integrate netlist inputs and netlist or schematic outputs into third-party environments while still supporting delay information and place and route constraints.
- Create and partition hierarchical schematics automatically.

Design compiler family mainly includes:

- **DC Expert:** Applied to high-performance ASIC and IC designs.
- **DC Ultra:** Applied to high-performance deep submicron ASIC and IC designs, where maximum control over the optimization process is required.
- **HDL Compiler Tools:** include HDL Compiler (Presto Verilog) and (V)HDL Compiler. The Verilog or (V)HDL compiler reads the HDL files and performs translation and architectural optimization of the designs.
- **Power Compiler:** Offers a complete methodology for power, including analyzing and optimizing designs for static and dynamic power consumption.

Before the start of synthesis process, the designer should write the design files and test design functionality. The design compiler take the design files and run synthesis process, it doesn't support writing design files. However, Synopsys design compiler supports several design files formats including VHDL, and Verilog as appear in the table below:

Format	Description	Keyword	Extension
PLA	Berkeley (Espresso) PLA format	Pa	.pla
State Table	Synopsys state table format	St	.st
TDL	Tool Design Language netlist format	Tl	.tdl
Verilog	Verilog Hardware Description Language	Verilog	.v
VHDL	VHSIC Hardware Description Language	Vhdl	.vhd
XNF	Xilinx netlist format	Xnf	.xnf

B.1 Design Compiler Interfaces

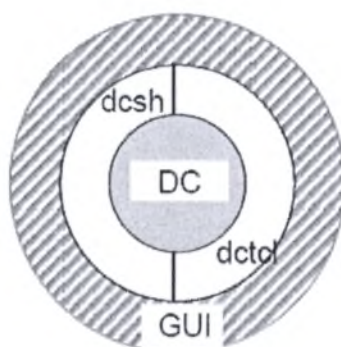
The Design Compiler has the following interfaces:

- **dc_shell** command-line interface: the dc_shell command-line interface has the following modes:

- **dcs** mode: This is the original Design Compiler command language and is specific to Synopsys

- **dctcl** mode: This Design Compiler command language is based on the tool command Language (Tcl) and includes certain command extensions needed to implement specific Design Compiler functionality.

- Graphical user interface: The Design Vision graphical user interface (GUI) provides menus, dialog boxes, and so forth for implementing Design Compiler commands. It also provides graphical displays, such as design schematics.



FigureA.1: Design compiler interfaces.

To start the design compiler enter the appropriate command to invoke it.

To invoke dc_shell in dcs mode, enter the dc_shell command at the system prompt:

```
% dc_shell
```

The system prompt changes to

```
% dc_shell>
```

To invoke dc_shell in the dctcl mode, enter

```
% dc_shell -tcl_mode
```

To run the design vision interface insert:

```
design_vision -xg
```

You can also include numerous options in these command lines, such as

- -checkout to access additional licenses
- -wait to set a wait time limit for checking out any additional licenses
- -f to execute a script file
- -x to include a dc_shell command that is executed at startup

Other options are available.

At startup, dc_shell does the following tasks:

1. Creates a command log file
2. Reads and executes the .synopsys_dc.setup files

3. Executes any script files or commands specified by the -x and -f options, respectively, on the command line.
4. Displays the program header and dc_shell prompt in the window from which you invoked dc_shell.

In this small tutorial we will discuss using of Design vision graphical user interface. When you run the command `design_vision -xg`

The program header will list for you the licensed members of the design compiler family as in figureA.2 and A.3

```

DC Professional (TM)
DC Expert (TM)
DC Ultra (TM)
FloorPlan Manager (TM)
HDL Compiler (TM)
VHDL Compiler (TM)
Library Compiler (TM)
DesignWare Developer (TM)
DFT Compiler (TM)
BSD Compiler
Power Compiler (TM)

Version Y-2006.06 for sparc64 -- May 25, 2006
Copyright (c) 1988-2006 by Synopsys, Inc.
ALL RIGHTS RESERVED

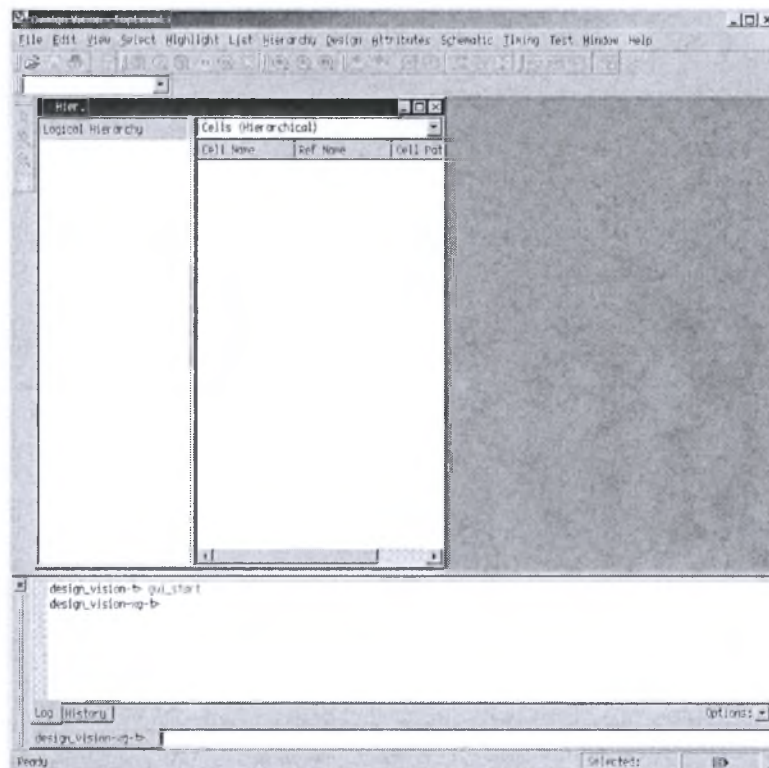
This software and the associated documentation are confidential and
proprietary to Synopsys, Inc. Your use or disclosure of this software
is subject to the terms and conditions of a written license agreement
between you, or your company, and Synopsys, Inc.

The above trademark notice does not imply that you are licensed to use
all of the listed products. You are licensed to use only those products
for which you have lawfully obtained a valid license key.

Initializing...
design_vision-xg-t> design_vision-xg-t> █

```

FigureA.2: Design vision startup display.



FigureA.3: Design vision startup display.

Maybe what displayed for you differs in the listed members according what licenses you have.

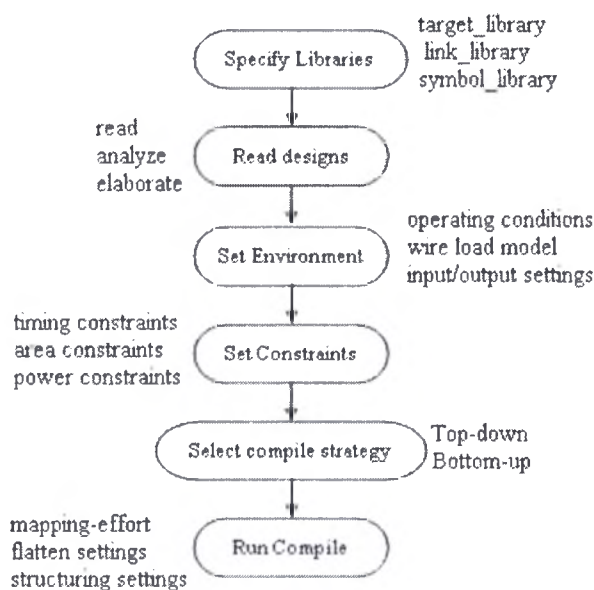
The displayed window of design vision you will see directly is shown in figure A.3.

The “Hier.1” window will display the hierarchy of the current design when you read it. In the log window at the bottom, the design compiler translate all the menus actions into commands as you press, them, also any information during processing is displayed in the log window.

At the tab “design_vision-xg-t>” you can perform processing using Tcl commands instead using menus.

B.2 Synthesis Flow

The basic synthesis flow consists of the steps shown in figureA.4, the figure shows beside each step the parameters to be set and actions.



FigureA.4: Synthesis flow.

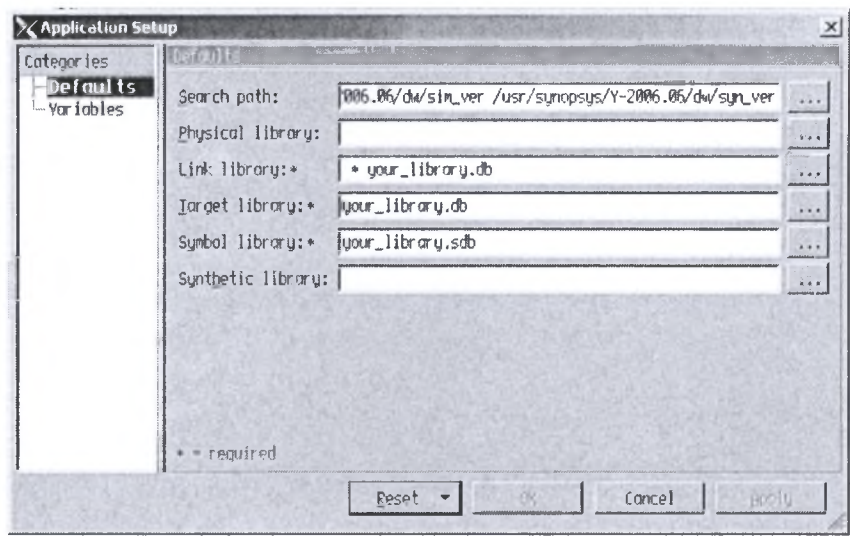
1. Specify Libraries

The first step after you start the design vision is to define the target library, link library, and symbol library to the design compiler. Those libraries should be in the search path of the design compile, or you can provide it with the full path.

To define the libraries, from file menu->setup, you get the setup window shown in figureA.5. In the search path you can define the path of your libraries (where you save them) and put it on the top of search paths list. In fields of link library, target library, symbol library, and synthetic library you can insert the names of your libraries.

As you note the target and link libraries should be in the db format, while the symbol library in the sdb format. You can use library compiler to translate libraries

into db format if they were in another format, normally semiconductor vendor provide designers with technology libraries in db format.



FigureA.5: setup window.

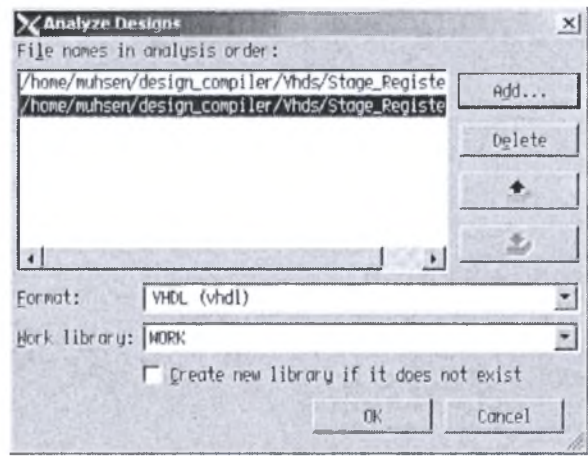
2. Read Designs

After definition of libraries, you can start design files read phase. The design compiler provide designer with two ways to read designs:

- Read command which can be used to read designs in any format type.
- Analyze and elaborate commands which specific for verilog and vhdl formats.

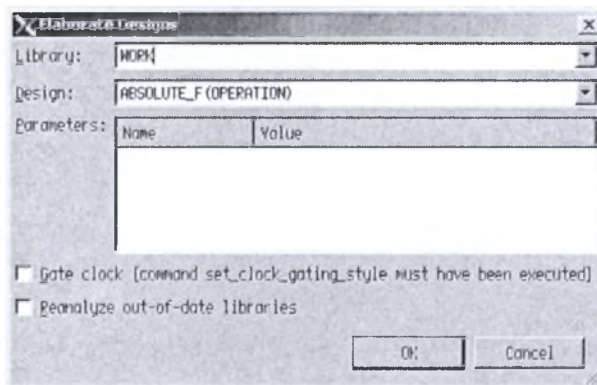
If your design files are written in vhdl or verilog format it is better to use analyze and elaborate commands since they give designer more abilities to change some generic parameters in the design, and they make check while reading.

The design files process must respect hierarchy; which means you should read in bottom-up direction, child file then its parents until reach the top file. To read design files run from *file menu* the *analyze* command then the window in figureA.6 will appear for you, by clicking add button you can select the files you want to read, take care to the order of reading, subdesign files at the top in the files list and (note you can reorder the design after addition by the arrow button), also take care to select design files format.



FigureA.6: read designs (analyze).

By *analyze* command you read the files as we said in bottom-up flow, and *elaborate* command is used only on the top level design and an subdesign you want to change some of its generic list parameters, also respect hierarchy in elaborate command.



FigureA.7: read designs (elaborate).

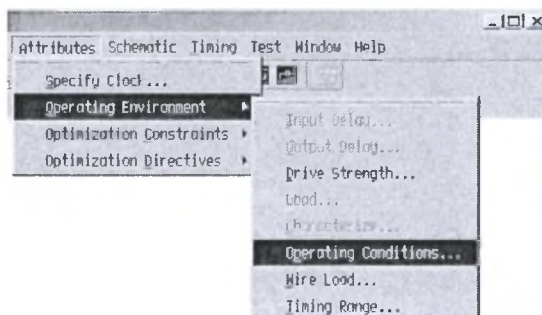
Also from file menu you can run elaborate command, and then the window shown in figureA.7 will appear. Note the parameters field; if the design has any generic list it will appear in parameters field, and so you can change them if you want.

After you run elaborate command on a design file, that design will be the current design, each design you run elaborate command for it will be available in the design compiler library, and you can switch between designs using **current_design** parameter.

3. Define Design Environment

Design Compiler requires that you model the environment of the design to be synthesized. This model comprises the external operating conditions (manufacturing process, temperature, and voltage), loads, drives, fanouts, and wire load models. It directly influences design synthesis and optimization results.

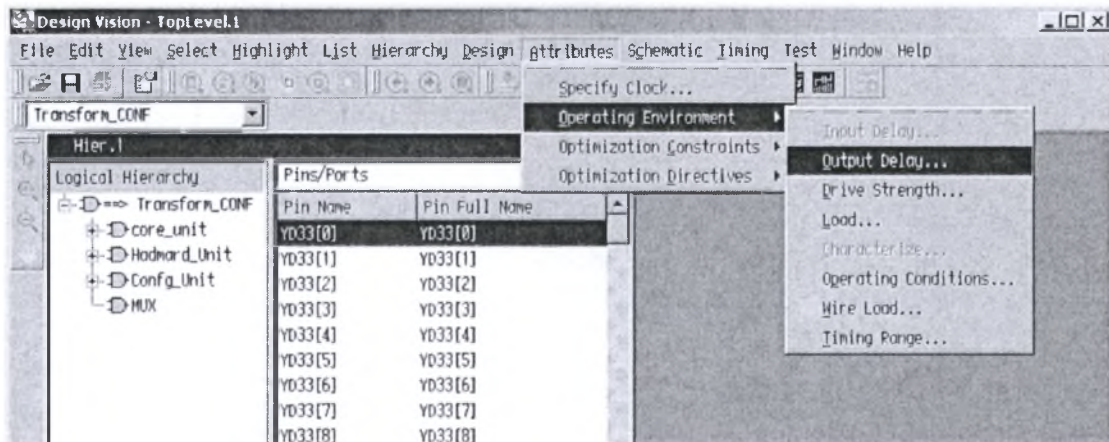
From *attributes* menu you can set the environment parameters as appear in figureA.8



FigureA.8: setting environment.

By selecting *operating environment* submenu you can select to set all possible environment parameters. The inactivated submenus aren't applied for the selected design object. For example, by selecting output port from the "Hier.1" window, as

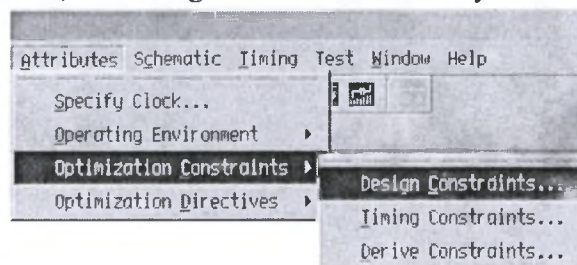
shown in figure below, the “output delay” and “load” submenus are activated, since we can apply these parameters for output ports.



FigureA.9: setting environment.

4. Set Design Constraints

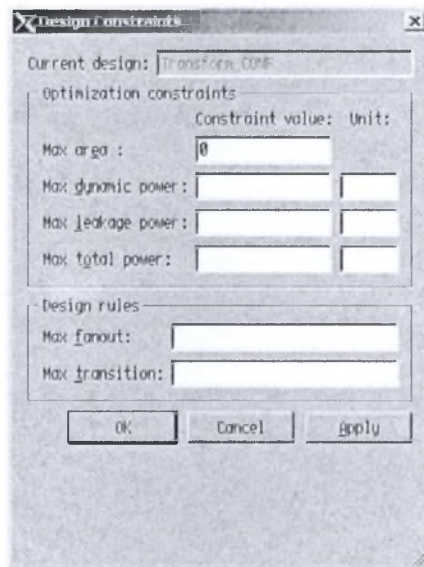
Design Compiler uses design rules and optimization constraints to control the synthesis of the design. Design rules are provided in the vendor technology library to ensure that the product meets specifications and works as intended. Typical design rules constraints are transition times, fanout loads, and capacitances. These rules specify technology requirements that you cannot violate. (You can, however, specify stricter constraints.) Optimization constraints define the design goals for timing (clocks, clock skews, input delays, and output delays) and area (maximum area). In the optimization process, Design Compiler attempts to meet these goals, but no design rules are violated by the process. To define design constraints, also from the *attributes* menu -> *optimization constraints* submenu you can select the *design constraints* to set area and power constraints, also design rules constraints if you want.



FigureA.10: setting design constraints.

The window shown in figureA.11 will appear; it has two parts optimization constraints part where area and power constraints are applied. The second part is the design rules part constraints.

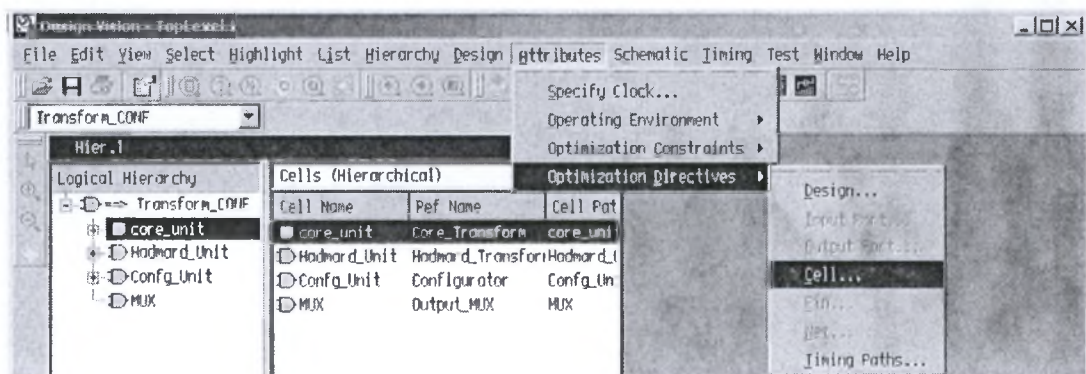
Again from *attributes* -> *optimization constraints* you can run *Timing constraints* window also you can specify clock signal.



FigureA.11: design constraints.

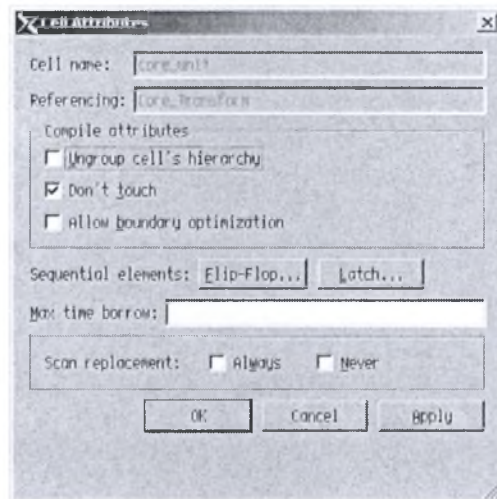
5. Select Compile Strategy

The two basic compile strategies that you can use to optimize hierarchical designs are referred to as top-down and bottom-up. In the top-down strategy, the top-level design and all its subdesigns are compiled together. All environment and constraint settings are defined with respect to the top-level design. Although this strategy automatically takes care of interblock dependencies, the method is not practical for large designs because all designs must reside in memory at the same time.



FigureA.12: compile strategy.

In the bottom-up strategy, individual subdesigns are constrained and compiled separately. After successful compilation, the designs are assigned the *dont_touch* attribute to prevent further changes to them during subsequent compile phases. Select the cell in the hierarch that you did compilation of it previously and you don't want to recompile it from the start when compile the top level, then from *attributes->optimization directives* select *cell*, the window in figureA.13 will appear.



FigureA.13: cell's attributes.

The compiled subdesigns are assembled to compose the designs of the next higher level of the hierarchy (any higher-level design can also incorporate unmapped logic), and these designs are compiled. This compilation process is continued up through the hierarchy until the top-level design is synthesized. This method lets you compile large

Designs faster because Design Compiler does not need to load all the uncompiled subdesigns into memory at the same time. At each stage, however, you must estimate the interblock constraints, and typically you must iterate the compilations, improving these estimates, until all subdesign interfaces are stable.

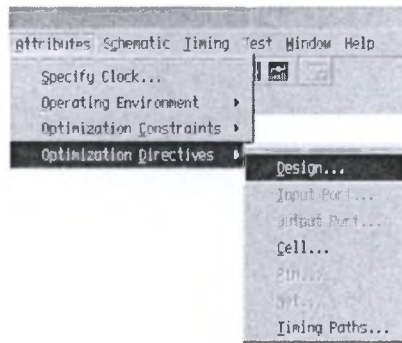
Each strategy has its advantages and disadvantages, depending on your particular designs and design goals. You can use either strategy to process the entire design, or you can mix strategies, using the most appropriate strategy for each subdesign.

6. Optimize the Design

To optimize the design you can direct the design compiler process by set of flatten and structuring parameters. Structuring process adds intermediate variables and logic structure to a design, which can result in reduced design area.

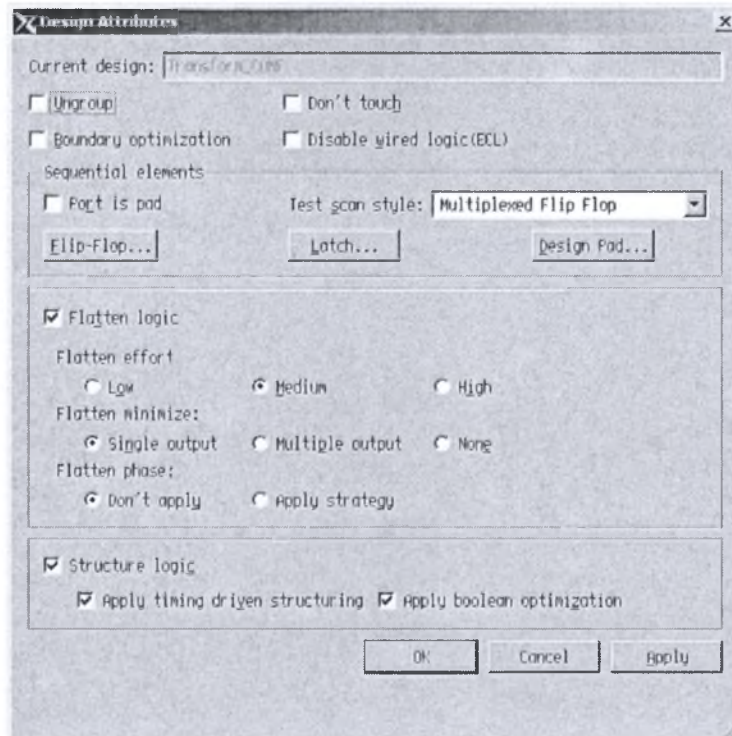
While flatten process converts combinational logic paths of the design to a two-level, sum-of-products representation. During flattening, Design Compiler removes all intermediate variables, and therefore all its associated logic structure, from a design. Flattening is not always practical, however, because it requires a large amount of CPU time and can increase area.

From the *attributes->optimization directives* select *design* as shown in figureA.14, the window of design attributes will appear.



FigureA.14: set optimization directives.

In the design attributes window you can enable flatten or structuring processes. You can select low, medium, or high effort for flatten.

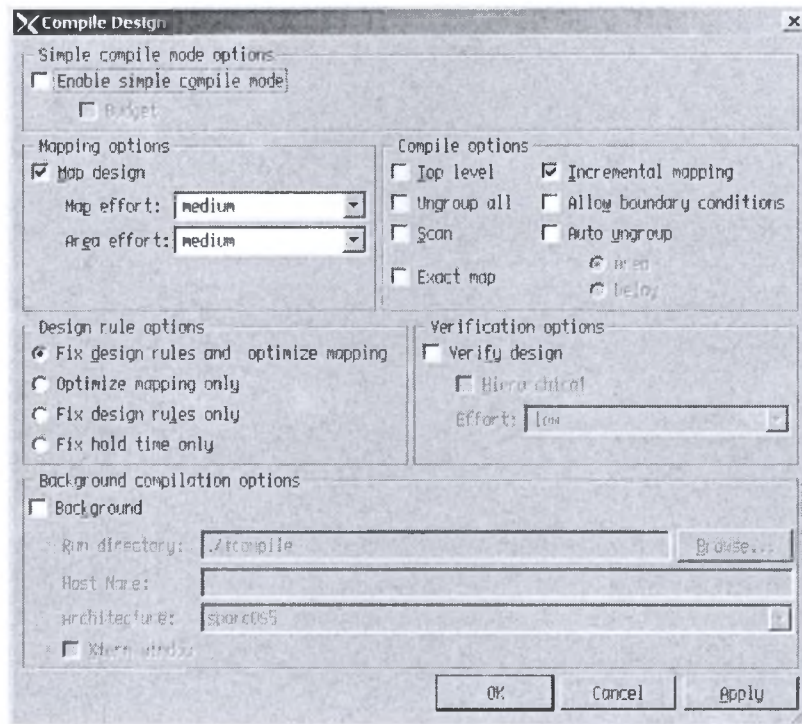


FigureA.15: design attributes.

7. Run Compile process

Now, after you finish all compile process setting you can run the design compiler. From *design* menu select *compile design* submenu the compile design window will appear.

Several compile options are available. In particular, the *map_effort* option can be set to low, medium, or high. In a preliminary compile, when you want to get a quick idea of design area and performance, you set *map_effort* to low. In a default compile, when you are performing design exploration, you use the medium *map_effort* option; you might want to set *map_effort* to high. You should use this option judiciously, however, because the resulting compile process is CPU intensive. Often setting *map_effort* to medium is sufficient.



FigureA.16: compile design.

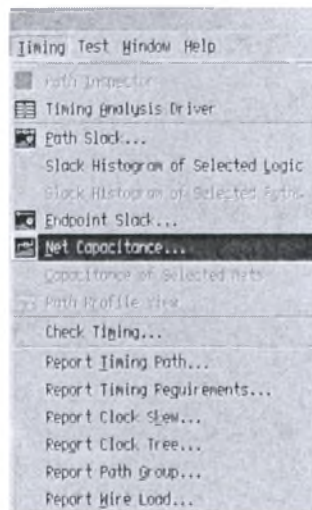
8. Analyze and Resolve Design Problems

Design Compiler can generate numerous reports on the results of a design synthesis and optimization, for example, area, constraint, and timing reports. You use reports to analyze and resolve any design problems or to improve synthesis results. From *design* menu you can run a lot of reports that enable you analyze the synthesis result.




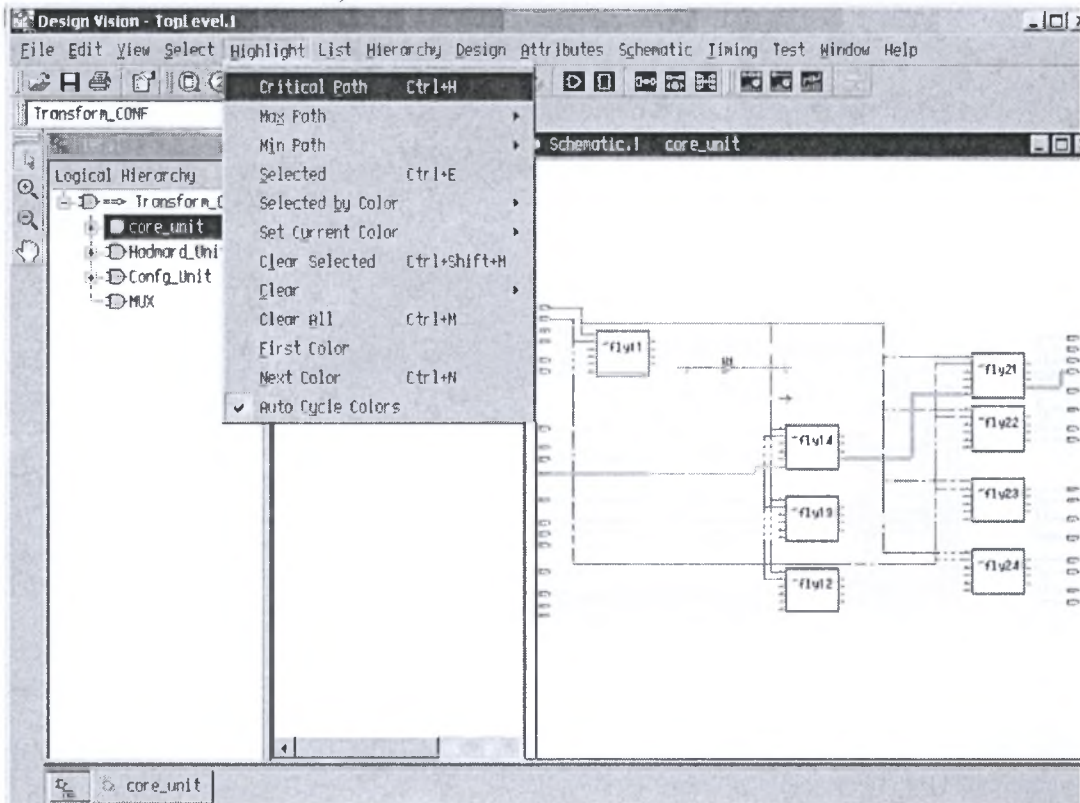
FigureA.17: get analyze reports.

Also from *Timing* menu you can run several kinds of timing reports and histograms of *net capacitance and timing slacks*. Also design vision interface enable you to view schematics of the design and mark critical path and other path types to take good overview of the design.



FigureA.17: get timing reports.

From *schematic->new design schematic view* submenu you can get the design schematic, or by clicking the button  you will get the same thing. From the *Highlight* menu you can select the path you want to highlight (take care that the schematic window is activated).



FigureA.18: Design schematic.

9. Save the Design Database

Remember that Design Compiler does not automatically save designs before exiting. You can also save in a script file the design attributes and constraints used during synthesis. Script files are ideal for managing your design attributes and constraints. From *file menu select save as* and then save the design where you want.

References

- [1] I. Amer, W. Badawy, and G. Jullien, "Hardware Prototyping for The H.264 4×4 Transformation," accepted in IEEE International Conference on Acoustics, Speech, and Signal Processing, Montreal, Canada, May 2004.
- [2] I. Amer, W. Badawy, and G. Jullien, "A VLSI Prototype for Hadamard Transform with Application to MPEW", accepted in IEEE International Conference on Acoustics, Speech, and Signal Processing, Montreal, Canada, May 2004.
- [3] Iain E. G. Richardson, "H.264 and MPEG-4 Video Compression", John Wiley & Sons Ltd, England, 2003.
- [4] S. Malvar, A. Hallapuro, M. Karczewicz, and L. Kerofsky, "Low-Complexity Transform and Quantization in H.264/AVC", IEEE transactions on circuits and systems for video technology, Vol. 13, NO. 7, July 2003, pp. 598-603.
- [5] I. E. G. Richardson, "H.264/MPEG-4 Part 10: Transform & Quantization", A white paper. [Online]. Available: <http://www.vcodex.com>, March 2003.
- [6] R. Schafer, T. Wiegand, and H. Schwarz, "The Emerging H.264/AVC Standard", *EBU Technical Review*, January 2003.
- [7] H. Malvar, A. Hallapuro, M. Karczewicz, and L. Kerfosky, "Low-Complexity Transform and Quantization with 16-bit Arithmetic for H.26L". *IEEE International Conference on Image Processing*, Rochester, New York, September 2002.
- [8] A. Hallapuro, M. Karczewicz, and H. Malvar, "Low Complexity Transform and Quantization - Part 11: Extensions", *Joint Video Team (J V of ISO/IEC MPEG and ITU-T VCEG, doc. JvT-BO39r2*, February 2002.
- [9] T. Stockhammer, M. M. Hannuksela, T. Wiegand, "H.264/AVC in wireless environments", *IEEE Transactions on Circuits and Systems For Video Technology*, Vol. 13, No. 7, July 2003, pp. 657-673.
- [10] K. Denolf, C. Blanch, G. Lafruit, and A. Bormans, "Initial memory complexity analysis of the AVC codec", *IEEE Workshop on Signal Processing Systems*, October 2002, pp. 222-227.

ΠΑΝΕΠΙΣΤΗΜΙΟ
ΘΕΣΣΑΛΙΑΣ



004000085926

