Διπλωματική εργασία του φοιτητή

Κωνσταντίνου Τριανταφυλλίδη

Με θέμα:

«Βελτιστοποίηση απόδοσης της αποκωδικοποίησης βίντεο με τον αλγόριθμο MPEG-2 σε υπολογιστικό σύστημα με επεξεργαστή Intel Atom.»

Επιβλέποντες καθηγητές:

Ιωάννης Κατσαβουνίδης

Νικόλαος Μπέλλας

Ευρετήριο

0. Ευχαριστίες ……………………………………………………………………….5

1.Εισαγωγή

1.1. Τι είναι video και γιατί συμπίεση;……………………………….6
1.1.1. Είδη συμπίεσης……………………………………………………..8

1.1.2. Μετατροπή RGB σε YUV…………………………………………..8

1.1.3. Πρότυπα YUV………………………………………………………..9

1.1.4. Διακριτός μετασχηματισμός συνημιτόνου(IDCT)………10

1.1.5. Διαδικασία κβαντοποίησης…………………………………..11

1.1.6. VLC……………………………………………………………….……11

1.1.7. Motion compensation………………………………………....12

1.1.7.1. Είδη frame………………………………………15

1.2. Τεχνολογία SIMD……………………………………………………17

1.2.1. MMX………………………………………………………………18

1.2.2. SSE, SSE2 κτλ……………………………………………..…19

1.3. Επεξεργαστής Atom

1.3.1. Intel Atom……………………………………………….……20

1.3.2. Το παρόν και το μέλλον………………………………..20

1.3.3. Απόδοση……………………………………………………21

1.4. Dell precision Latitude 2100………………………………………22

1.5 Microsoft Visual Studio

1.5.1. Γενικά…………………………………………………………23

1.5.2. Debbuger……………………………………………………..23

1.5.3. MSDN Library…………………………………………….24

1.5.4. Microsoft Visual Studio 2008………………………..24

1.6. Μέτρηση απόδοσης

1.6.1. Vtune………………………………………………………25

1.6.2. Χαρακτηριστικά Vtune………………………………..26

1.7. Intrinsics

1.7.1. Τι είναι Intrinsics………………………………………..27

1.7.2. Intrinsics MMX…………………………………………...27

1.7.3. Intrinsics SSE2…………………………………………….28

2. Αλλαγές στον reference code για διευκόλυνση της εκτέλεσης

2.1. Σώσιμο ή όχι των αποκωδικοποιημένων frame………….30

2.2. Εγγραφή νέου YUV ………………………………………………...31

3.Κόστος συναρτήσεων

3.1. Κόστος συναρτήσεων του reference code…………………37

3.1.1 Μέτρηση των fps…………………………………………...38

3.1.2 Dantes.m2v………………………………………………….39

3.1.3 Input_Inter.m2v…………………………………………..40

3.1.4 Συμπεράσματα αρχικών μετρήσεων……………..41

3.2. Στρατηγική για τη μείωση του κόστους……………………..41

3.3. Επιμέρους συναρτήσεις…………………………………………….41

3.3.1. IDCT………………………………..…………………………42

3.3.1.1 Inverse Transform…………………………..43

3.3.1.2 idctrow-idctcol……………………………….46

3.3.1.3 Κλίση της FAST IDCT ………………………55

3.3.1.4 idct1x1-idct2x2-idct4x4…………………..56

3.3.2. Addblock…………………………………………………66

3.3.3. Motion Compensation………………………………69

3.3.4. Saturate……………………………………………………78

3.3.5. Miss match control……………………………………86

3.5 Απόδοση των αλλαγών 3.1-3.4………………………..90

Dantes.m2v ……………………………………………….91

Input_Inter.m2v…………………………………………94

3.6 Flush_Buffer……………………………………………………98

3.7 Clear_block……………………………………………………100

3.8 Τελική απόδοση – Συμπεράσματα…………………101

3.8.1 dantes - Input_Intep - Vin-4-p(HD)……………...101

4.Συμπεράσματα…………………………………………………104

0.Ευχαριστίες

Τελειώνοντας αυτή εδώ την πτυχιακή, φτάνει πλέον το τέλος της φοιτητικής μου ζωής. Αισθάνομαι την ανάγκη να ευχαριστήσω την οικογένειά μου για την αμέριστη συμπαράσταση όλα αυτά τα χρόνια ηθική αλλά και οικονομική. Επίσης οφείλω να ευχαριστήσω τον αδερφό μου που με ανέχτηκε 5 ολόκληρα χρόνια συγκατοίκησης και όλους τους φίλους που μαζί ξοδέψαμε τα καλύτερά μας χρόνια(δεν αναφέρω ονόματα μήπως ξεχάσω κάποιον).

Όσον αφορά τις σπουδές μου ευχαριστώ θερμά τον υπεύθυνο καθηγητή της διπλωματικής μου εργασίας, τον κ. Ιωάννη Κατσαβουνίδη που με καθοδήγησε όντας παρόν οποτεδήποτε χρειαζόμουν τη βοήθειά του και μου άνοιξε το δρόμο για αυτό το όμορφο ταξίδι τον τελευταίο ενάμιση χρόνο στον κόσμο του video.

Τέλος, για την ολοκλήρωση της διπλωματικής αυτής εργασίας καθοριστικό ρόλο είχε και ο δεύτερος επιβλέπων, κ. Νικόλαος Μπέλλας που με συνεπήρε με το πάθος του για τα embedded και portable υπολογιστικά συστήματα.

1.Εισαγωγή

1.1.Τι είναι βίντεο και γιατί συμπίεση;

Βίντεο είναι μια συνεχόμενη ροή φωτογραφιών με τέτοιο τρόπο ώστε το ανθρώπινο μάτι να αντιλαμβάνεται μία συνεχόμενη φυσική κίνηση. Ας πάρουμε για παράδειγμα μία βιντεοκάμερα που χρησιμοποιούμε για τη μετάδοση ενός ποδοσφαιρικού αγώνα. Μια τυπική βιντεοκάμερα ρυθμισμένη να λειτουργεί στην Ευρώπη κάνει λήψη 25 στιγμιότυπων ανά δευτερόλεπτο(29.996fps είναι το πρότυπο της Αμερικής). Έτσι ουσιαστικά μια βιντεοκάμερα κάνει λήψη 25 φωτογραφιών ανά δευτερόλεπτο (πχ για 60 sec έχει κάνει λήψη 1500 φωτογραφίες), και αν οι φωτογραφίες αυτές παιχτούν με αυτή τη συχνότητα συγκεκριμένα με 25 καρέ ανά δευτερόλεπτο (25 fps=frames per second) τότε έχω μία συνεχόμενη κίνηση ποδοσφαιριστών και μπάλας κατά τη διάρκεια ενός ποδοσφαιρικού αγώνα. Το πρόβλημα πλέον είναι πως θα κάνω αποθήκευση ή και αναμετάδοση ένα τόσο μεγάλο όγκο δεδομένων. Ας αναλογιστούμε ότι έστω μια φωτογραφία ανάλυσης 720x480 σε ασυμπίεστη μορφή έχει μέγεθος 720x480x8 bit τιμές χρωμάτων x 3 χρώματα (YUV) =1036800 Byte ανά καρέ που σημαίνει 25920000 Bytes/sec =25 MB/sec θα πρέπει να είναι το bandwidth. Κάτι τέτοιο είναι ανέφικτο και θα καθιστούσε απαγορευτική την ψηφιοποίηση των βίντεο. Εδώ πλέον βρίσκει εφαρμογή η συμπίεση των δεδομένων του βίντεο ώστε το παραγόμενο ψηφιακό αρχείο να έχει ένα αποδεκτό μέγεθος ώστε και να μπορώ να το αποθηκεύσω στο σκληρό δίσκο ή σε ένα οπτικό δίσκο αλλά και να μπορώ να το αναμεταδώσω ώστε κάποιος να μπορεί να παρακολουθεί το βίντεο σε πραγματικό χρόνο από το σπίτι του. Αντίστοιχα για να δω το συμπιεσμένο βίντεο θα χρειαστεί να κάνω την αντίστροφη διαδικασία από αυτή της συμπίεσης(compression), την αποσυμπίεση (decompression).

Παρακάτω διατίθενται δύο διαγράμματα όπου φαίνονται οι γενικές διαδικασίες συμπίεσης και αποσυμπίεσης video σύμφωνα με τον αλγόριθμο MPEG-2.

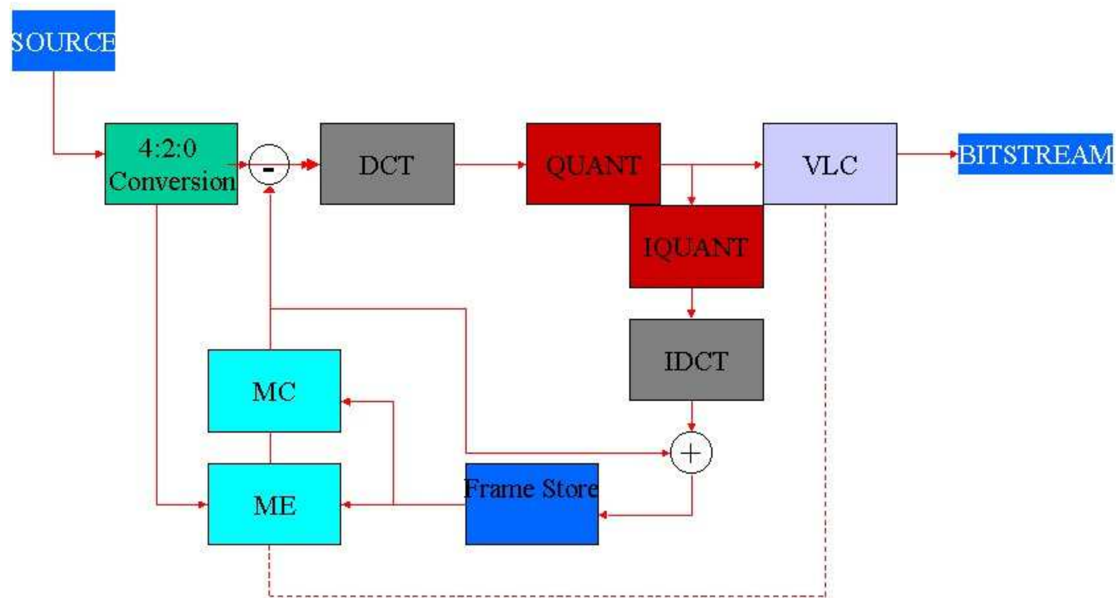
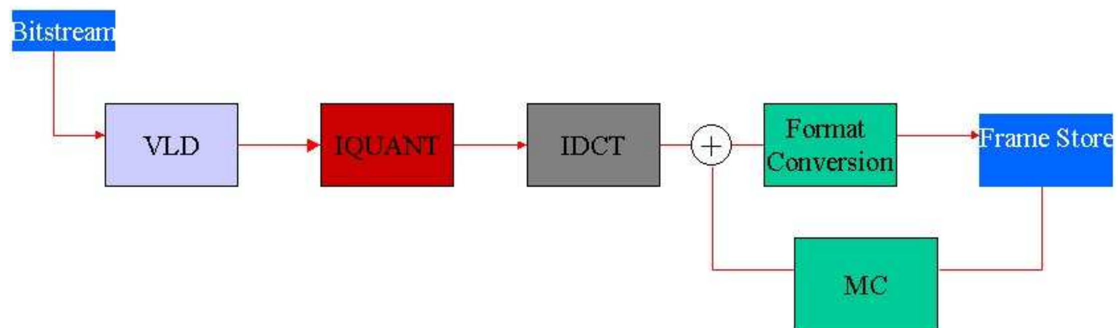


Fig 1.1 video compression video compression



DCT: Discrete Cosine Transform
QUANT: Quantization
IQUANT: Inverse quantization
IDCT: Inverse Discrete Cosine Transform
VLC: Variable length coding
VLD: Variable length decoding

Fig 1.2 video decompression

1.1.1. Είδη συμπίεσης

Υπάρχουν 2 κύρια είδη συμπίεσης ενός αρχείου βίντεο: η συμπίεση κατά την οποία έχω αλλοίωση του αρχικού βίντεο (lossy compression) και η συμπίεση κατά την οποία δεν έχω αλλοίωση του αρχικού βίντεο (lossless compression).

Procedure	Lossy	Lossless
Dct	ΟΧΙ	Ναι
Quantization	Ναι	ΟΧΙ
Vlc	ΟΧΙ	Ναι

Table 1.1 Είδη συμπίεσης

Οι γενικές διαδικασίες συμπίεσης και αποσυμπίεσης ενός βίντεο είναι οι ίδιες. Ακόμα και στη διαδικασία της συμπίεσης, όπως παρατηρούμε και από το παραπάνω σκίτσο, εκτελούμε αποσυμπίεση μετά την αποθήκευση ή την αποστολή του bitstream, ώστε να συμπιέσουμε τα επόμενα δεδομένα σύμφωνα με τα προηγούμενα καθώς είπαμε πως η συμπίεση είναι lossy και όχι lossless, έτσι ώστε ένα σφάλμα που πιθανόν να παράγεται (χάνω σε ποιότητα λόγω της κβαντοποίησης) να μη συνεχίσει και στα παρακάτω frame.

1.1.2. Μετατροπή RGB σε YUV

Ως γνωστό μια εικόνα απεικονίζεται από τρία χρώματα το κόκκινο, το πράσινο και το μπλε (το γνωστό RGB). Στο βίντεο τα πράγματα είναι λίγο διαφορετικά καθώς χρησιμοποιώ το πρότυπο YUV που δεν είναι κάτι άλλο από μια διαφορετική αναπαράσταση του προτύπου RGB.

Η μετατροπή του RGB σε YUV δίνεται από την παρακάτω γραμμική σχέση:

```
Y = ( (  66 * R + 129 * G +  25 * B + 128) >> 8) +  16
U = ( ( -38 * R -  74 * G + 112 * B + 128) >> 8) + 128
V = ( ( 112 * R -  94 * G -  18 * B + 128) >> 8) + 128
```

1.1.3. Πρότυπα YUV στον αλγόριθμο συμπίεσης MPEG-2

Ο αλγόριθμος συμπίεσης video MPEG-2 υποστηρίζει τρία είδη format για το χρώμα:

- 4:4:4
- 4:2:2
- 4:2:0

Πρότυπο 4:4:4

Σε αυτό το πρότυπο όλα τα στοιχεία του YUV, δηλαδή τόσο το Luma όσο και τα Crhominance στοιχεία Cb, Cr έχουν ίδιο αριθμό για κάθε pixel της εικόνας. Δηλαδή για κάθε ένα στοιχείο Luma (φωτεινότητα) υπάρχει και ένα στοιχείο Cb και ένα Cr.

Πρότυπο 4:2:2

Σε αυτό το πρότυπο τα στοιχεία Cb και Cr είναι τα μισά από ότι τα στοιχεία του Luma. Με αυτό τον τρόπο επιτυγχάνεται μείωση κατά 1/3 της πληροφορίας χωρίς να υπάρχουνε εμφανείς διαφορές στο ανθρώπινο μάτι.

Πρότυπο 4:2:0

Σε αυτό το πρότυπο πάλι τα στοιχεία του Cb και Cr είναι τα μισά από αυτά του Luma αλλά η ειδοποιός διαφορά είναι ότι για να προκύψουν οι τιμές των Chroma γίνεται μια γραμμική παρεμβολή τόσο στον οριζόντιο όσο και στον κατακόρυφο άξονα με αποτέλεσμα η ποιότητα και πιστότητα της εικόνας να παραμένει παρεμφερής με την αρχική, έχοντας πετύχει ωστόσο μείωση του μεγέθους του αρχείου κατά 1/3 όπως και στο πρότυπο 4:2:2.

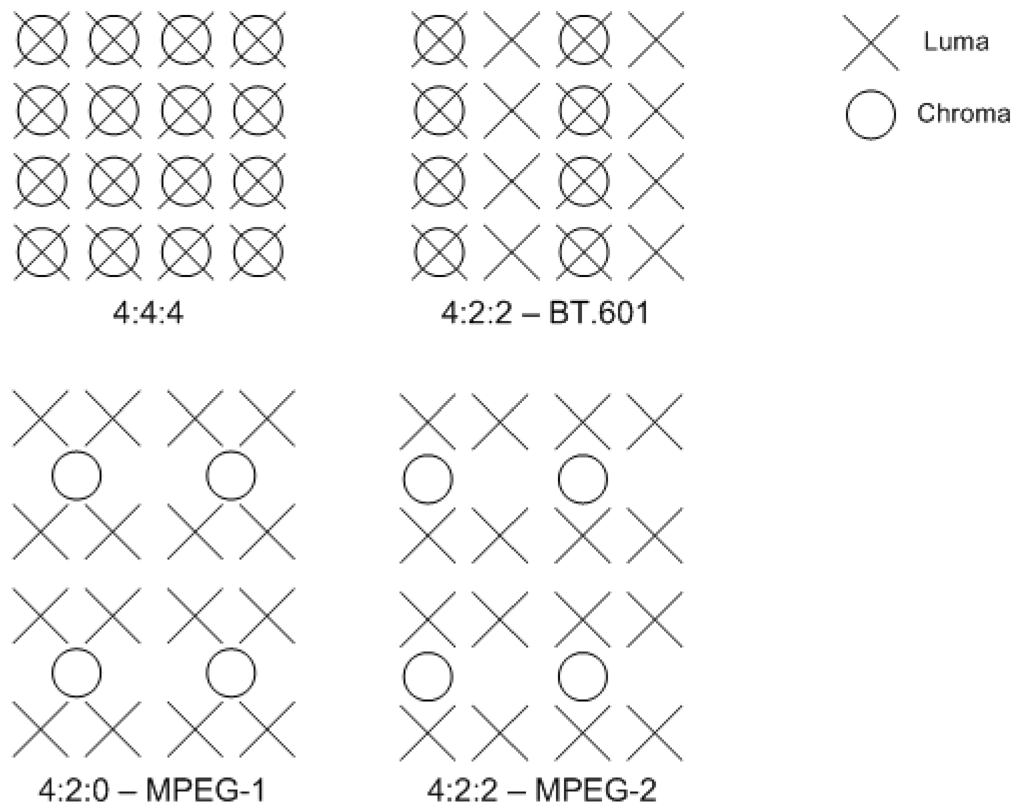


Image 1.1 formats of YUV

1.1.4. Διακριτός μετασχηματισμός συνημιτόνου (DCT)

Ο διακριτός μετασχηματισμός συνημιτόνου είναι ένας μετασχηματισμός που χρησιμοποιείται ιδιαίτερα στις εφαρμογές συμπίεσης εικόνων, βίντεο και ήχου. Η κυριότερη του ιδιότητά που τον κάνει τόσο πολύτιμο εργαλείο είναι η συγκέντρωση της ενέργειας του σήματος σε λίγα διακριτά στιγμιότυπα(συγκέντρωση της ενέργειας στα πάνω αριστερά pixel των block).

Η συνάρτηση που δίνει το DCT ενός σήματος είναι:

$$X_k = \sum_{n=0}^{N-1} x_n \cos\left[\frac{\pi}{N}\left(n+\frac{1}{2}\right)k\right] \qquad k = 0, \dots, N-1.$$

Η διαδικασία του διακριτού μετασχηματισμού συνημιτόνου θεωρείται lossless καθώς η αντίστροφη διαδικασία θεωρητικά προσδίδει το ίδιο αποτέλεσμα με το αρχικό frame. Οι όποιες αποκλίσεις οφείλονται σε τυχόν στρογγυλοποιήσεις.

1.1.5. Διαδικασία κβαντοποίησης (Quantization)

Κβαντοποίηση ή quantization όπως συνηθίζεται να αποκαλείται, είναι η διαδικασία κατά την οποία περιορίζονται τα bit που απεικονίζουνε το χρώμα μιας εικόνας. Έτσι, ενώ αρχικά μια εικόνα-frame έχει εκατομμύρια χρώματα (τιμές χρώματος), μετά τη διαδικασία της κβαντοποίησης οι τιμές που μπορούνε να πάρουνε τα χρώματα είναι 0…255. Εδώ όπως γίνεται αντιληπτό είναι αδύνατο να μην έχω παραποίηση του αρχικού βίντεο καθώς αναγκαζόμαστε να κάνουμε διάφορες στρογγυλοποιήσεις (λόγω διαίρεσης) άρα και χάνουμε και κάποια από την αρχική πληροφορία. Πιο συγκεκριμένα κατά τον MPEG-2, μετά τη διαδικασία του DCT κάνω ουσιαστικά τη διαίρεση της ενέργειας με ένα αριθμό ώστε κατά την αποθήκευση ή μεταφορά των δεδομένων οι αριθμοί που θα προκύψουν να έχουν μικρότερες τιμές άρα και να κοστίζουν λιγότερα bit.

1.1.6. Διαδικασία VLC(variable length coding)-Huffman

Αυτή είναι και η τελευταία διαδικασία πριν την αποστολή ή αποθήκευση του συμπιεσμένου πλέον βίντεο. Μια διαδικασία VLC είναι η διαδικασία κωδικοποίησης Huffman. Ανάλογα με τη συχνότητα εμφάνισης ενός αριθμού αυτός ο αριθμός κωδικοποιείται με όσο το δυνατόν μικρότερο ορμαθό από δυαδικά bit. Έτσι ένας αριθμός που εμφανίζεται πάρα πολλές φορές θα αποτελείται από πολύ λιγότερα bits σε σχέση με ένα άλλο αριθμό που συναντάται πολύ πιο σπάνια μέσα στο ασυμπίεστο video. Πλέον στο βίντεό έχοντας συγκεντρώσει την ενέργεια όλων των pixel του σε συγκεκριμένα pixel μέσω του DCT και κατόπιν εφαρμόζοντας Quantization (δηλαδή διαιρώντας αυτές τις τιμές με κάποιους αριθμούς), θα περιέχει ορισμένες τιμές μη μηδενικές που θα πρέπει να σταλούνε και πάρα πολλές μηδενικές τιμές οι οποίες πιάνουν 8 bit πληροφορίας. Αυτό αντιμετωπίζεται με την τεχνική run length όπου στέλνω ανά ζεύγη τους αριθμούς και ο πρώτος δηλώνει τον αριθμό των μηδενικών ενώ ο δεύτερος τι ακολουθεί το μηδέν . Αυτές οι τιμές θα πρέπει να κωδικοποιηθούνε ώστε να έχω αποστολή όσο το δυνατόν λιγότερο bit(Huffman). Η μέθοδος **Huffman** είναι μία μέθοδος που αντιστοιχίζει σε συχνότερα εμφανιζόμενες τιμές, μία συμβολική τιμή που είναι μικρή (έχει όσο το δυνατόν λιγότερα bit). Ταυτόχρονα δημιουργεί και ένα «λεξικό» (έναν πίνακα δηλαδή) που δείχνει αυτή την αντιστοίχηση και προσθέτει και το λεξικό αυτό στο σήμα για να χρησιμοποιηθεί από τον αποκωδικοποιητή. Έτσι τιμές που εμφανίζονται συχνά και έχουν μεγάλο αριθμό bit περιγράφονται με άλλες που

έχουν μικρότερο, με συνέπεια την οικονομία σε bit. Αυτή η διαδικασία προφανώς και είναι lossless.

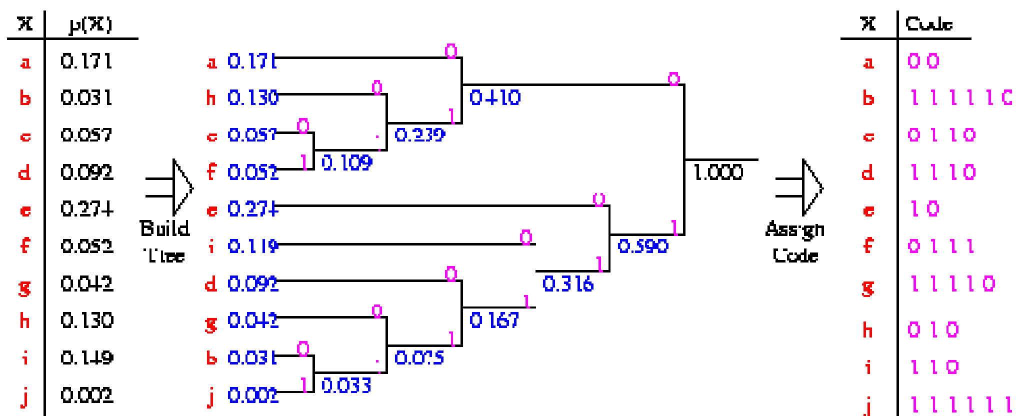


Fig 1.3 Huffmann encoding

Όλες οι παραπάνω διαδικασίες είναι κοινές τόσο για συμπίεση video όσο και για αποσυμπίεση video. Δηλαδή αν ένα βίντεο είναι συμπιεσμένο θα πρέπει να αποσυμπιεστεί με την άκρως αντίστροφη διαδικασία. Η μόνη διαδικασία που πραγματοποιείται στη συμπίεση και όχι στην αποσυμπίεση είναι η διαδικασία του motion estimation.

1.1.7. Motion compensation

Όταν σε ένα βίντεο ο ρυθμός ανανέωσης των καρέ είναι 25 καρέ ανά δευτερόλεπτο τότε η πιθανότητα το ένα καρέ να διαφέρει κατά πολύ από το προηγούμενο είναι πολύ μικρή. Για παράδειγμα όταν το τοπίο σε ένα βίντεο είναι ουρανός δεν μπορεί το επόμενο καρέ να είναι πολύ διαφορετικό από το προηγούμενο. Αυτό το γεγονός θα πρέπει να το εκμεταλλευτούμε κατά τη διαδικασία της συμπίεσης και τα κομμάτια της εικόνας που είναι ίδια να μην τα κωδικοποιήσουμε ξαναστέλνοντας πληροφορία. Αυτό που μπορεί να γίνει είναι αντί να κωδικοποιήσω την εικόνα από την αρχή να στείλω τις διαφορές σε σχέση με το προηγούμενο frame (διανύσματα).

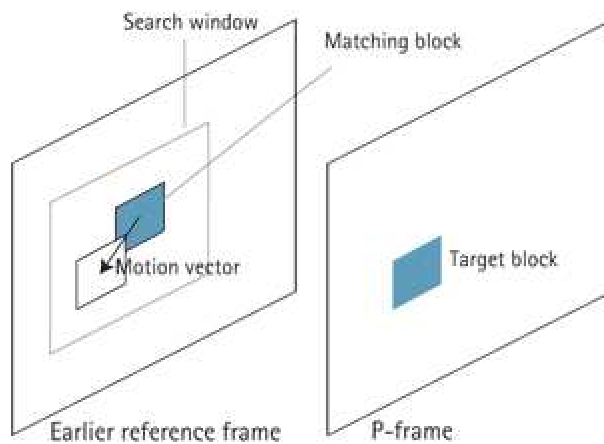


Image 1.2 motion vector

Με τη βοήθεια αυτής της μεθόδου μπορούμε να εκτελέσουμε τη δια-πλαισιακή (inter-frame) κωδικοποίηση και να περιγράψουμε την αλληλουχία εικόνων ως σειρά ομοιοτήτων και διαφορών. Παρακάτω φαίνονται 3 συνεχόμενα frame από ένα video όπου κάποια αερόστατα απογειώνονται από το έδαφος. Από τις εικόνες γίνεται αντιληπτό ότι οι διαφορές στα frame δεν είναι πολύ μεγάλες, παρά μόνο έχω αλλαγή της θέσης των αερόστατων, ενώ το background παραμένει ίδιο.

a.

b.

c.

Image 1.3 screenshots of a video a. frame 1, b. frame 2 and c. frame 3

Σύμφωνα με τα παραπάνω δίνεται η δυνατότητα αντί να κωδικοποιήσουμε ανεξάρτητα τα δύο frame, να τα χωρίσουμε σε ίσα τμήματα και να φτιάξουμε ένα πίνακα που να περιέχει τμήματα που έχουν μείνει ίδια και διανύσματα που να δείχνουν τη νέα θέση των τμημάτων που άλλαξαν θέση. Έτσι αν έχουμε ήδη αποστείλει το πρώτο frame, μπορούμε να στείλουμε το δεύτερο σαν ένα πίνακα 20 διανυσμάτων και ορισμένων σταθερών τμημάτων, που προφανώς έχει πολύ μικρότερο μέγεθος.

Ωστόσο δεν είναι εφικτό να αναπαραστήσουμε ένα frame χρησιμοποιώντας μόνο το προηγούμενό του, καθώς οι διαφορές δύο συνεχόμενων frame δεν θα είναι μόνο στη θέση κάποιων αντικειμένων. Τα δύο frame θα έχουν σίγουρα κάποια κοινά τμήματα που αλλάζουν θέση αλλά θα υπάρχουν και τμήματα που αλλάζουν θέση διατηρώντας το σχήμα τους αλλά μεταβάλλεται το χρώμα τους, καθώς και άλλα που δεν υπάρχουν σε προηγούμενο πλαίσιο αλλά εμφανίζονται σε κάποιο για πρώτη φορά. Για την κάλυψη αυτών των περιπτώσεων χρησιμοποιείται μία πιο βελτιωμένη εκδοχή της παραπάνω ιδέας (ή για την ακρίβεια διάφορες εκδοχές της παραπάνω ιδέας).

Η σύνταξη του MPEG καθορίζει πως θα αναπαρίσταται η πληροφορία για την κίνηση του κάθε macroblock, ότι θα γίνεται δηλαδή αυτή η αναπαράσταση με τη χρήση διανυσμάτων κίνησης, αλλά δεν καθορίζει πως τα διανύσματα αυτά θα υπολογίζονται και για το λόγο αυτό εμφανίζονται διάφορες υλοποιήσεις της μεθόδου εύρεσης των διανυσμάτων κίνησης οι οποίες στηρίζονται όλες στην ελαχιστοποίηση μίας συνάρτησης που υπολογίζει την ταύτιση του τρέχοντος με το macroblock αναφοράς.

Αν και μπορεί να χρησιμοποιηθεί κάθε συνάρτηση σφάλματος που υπάρχει, η πιο συχνά χρησιμοποιούμενη συνάρτηση είναι η Απόλυτη Διαφορά (AE - Absolute Error) η οποία δίνεται από τον παρακάτω τύπο :

$$\mathrm{AE}(d_x, d_y) = \sum_{i=0}^{15} \sum_{j=0}^{15} |f(i,j) - g(i - d_x, j - d_y)|$$

Στην παραπάνω εξίσωση το f(i,j) και g(i,j) αντιπροσωπεύουν τις συντεταγμένες των pixel στο τρέχον και το macroblock αναφοράς αντίστοιχα. Το macroblock αναφοράς που καθορίζεται από το διάνυσμα (d_x,d_y) αντιπροσωπεύει την περιοχή αναζήτησης. Το macroblock που παράγει το μικρότερο σφάλμα αντιστοιχεί στην τιμή του διανύσματος που ψάχνουμε.

Η πιο απλή διορατικά αλλά και η πιο πολύπλοκη από πλευράς υπολογιστικής πολυπλοκότητας είναι η πλήρης αναζήτηση (full search) η οποία καλύπτει κάθε pixel στην περιοχή αναζήτησης.

Σύμφωνα με τα παραπάνω υπάρχουνε frame που πρέπει να κωδικοποιούνται από την αρχή(reference frame), και frame που εξαρτώνται από τα reference frame. Έτσι τα frame στο mpeg2 είναι 3: I frame, B frame και P frame.

1.1.7.1. Είδη frame

Όπως παρατηρώ και από την παραπάνω εικόνα τα P frames εξαρτώνται μόνο από τα I frames ενώ τα B frames που είναι ενδιάμεσα των I και P, εξαρτώνται τόσο από τα I frames όσο και από τα P frames.

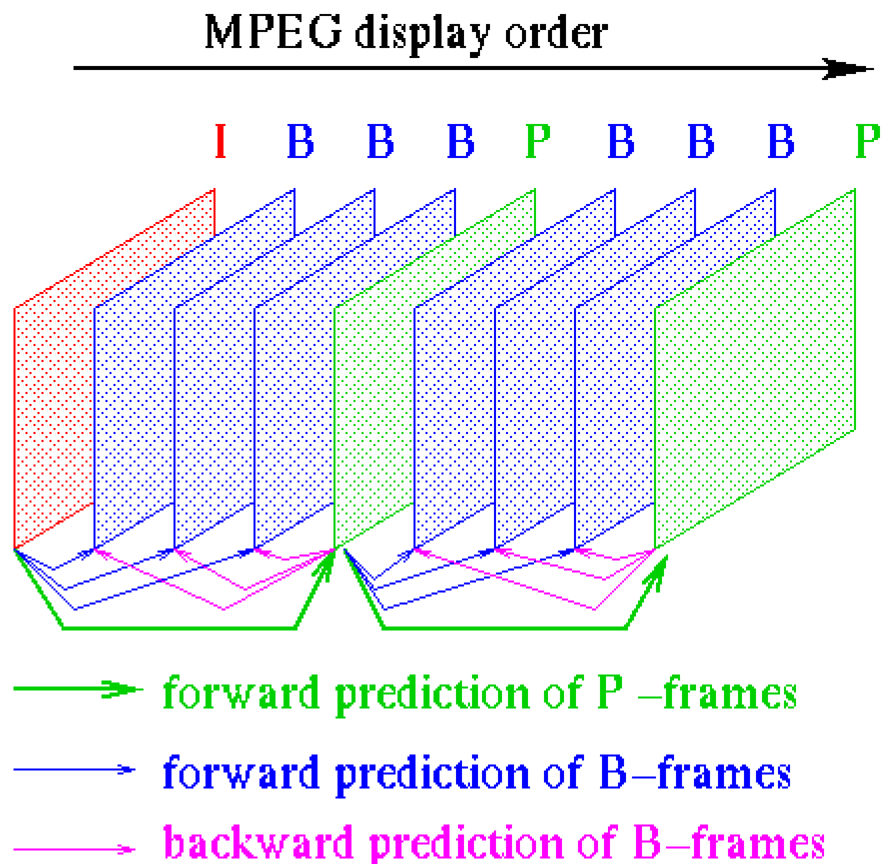


Image 1.4 frame order

I (Intra frame) : Το είδος αυτό των πλαισίων κάνει χρήση του intra frame-coding. Τα πλαίσια **τύπου I** είναι τα μόνα που είναι κωδικοποιημένα στο σύνολό τους και η αποκωδικοποίηση μπορεί να γίνει χωρίς αναφορά σε κάποιο άλλο. Είναι κατά συνέπεια τα μεγαλύτερα σε μήκος και αποτελούν σημεία αναφοράς κατά την τυχαία προσπέλαση ενός σήματος. Επειδή η παρουσία τους είναι απαραίτητη σα σημείο χρονικής αναφοράς και για να αποφευχθεί η διάδοση των σφαλμάτων που δημιουργούν τα P πλαίσια επιβάλλεται να μεταδίδονται ανά τακτά χρονικά πλαίσια. Έτσι υπάρχει ένα I πλαίσιο τουλάχιστον κάθε 15 πλαίσια (δηλαδή δύο φόρες το δευτερόλεπτο αν η συχνότητα είναι 30 Hz).

Η διαδικασία της κωδικοποίησης ενός I πλαισίου φαίνεται στο παρακάτω σχήμα. Η εικόνα χωρίζεται σε macroblocks και για κάθε block ξεχωριστά εφαρμόζεται DCT, Κβαντοποίηση, Zig-Zag Scanning,Run-Length-Encoding και Huffman Encoding (οι μέθοδοι αυτοί θα εξεταστούν ξεχωριστά παρακάτω).

P (Predicted frame) : Τα πλαίσια **τύπου P** είναι βασισμένα σε ένα προηγούμενο I ή P πλαίσιο. Με τη βοήθεια του motion compensation προβλέπουν τη νέα θέση όποιων macroblocks έχουν απλά μετακινηθεί και κωδικοποιούν τον αριθμό του macroblock και ένα διάνυσμα κίνησης. Με τη σειρά τους μπορούν να αποτελέσουν και αυτά σημείο αναφοράς για επόμενα πλαίσια και αυτός είναι και ο λόγος που συμβάλλουν στην εισαγωγή και διάδοση σφαλμάτων, αφού η διαδικασία της πρόβλεψης κίνησης δεν μπορεί να είναι 100% ακριβής. Δεν έχουν το μέγεθος των I πλαισίων γιατί δεν έχουν περιγραφεί με την ίδια ακρίβεια, δηλαδή παρουσιάζουν μεγαλύτερο ποσοστό συμπίεσης. Η διαδικασία λοιπόν της κωδικοποίησης τους, όπως φαίνεται και από το παρακάτω σχήμα είναι: σύγκριση macroblocks και δημιουργία ενός γραμμικού συνδυασμού αυτών που παρουσιάζουν σημαντική ομοιότητα, δημιουργία motion vector, μετασχηματισμός DCT σε κάθε block του νέου macroblock, Κβαντοποίηση, Run-Lenght-Encoding και το αποτέλεσμα κωδικοποιείται με κωδικοποίηση Huffman, όπως και στα I πλαίσια.

B (Bi-directional frame): Τα πλαίσια **τύπου B** είναι πλαίσια που δημιουργούνται λαμβάνοντας το μέσο όρο σε επίπεδο macroblock ενός προηγούμενου και ενός επόμενου πλαισίου I και P (ένα από το κάθε είδος). Δε συντελούν τόσο πολύ στη διάδοση των σφαλμάτων γιατί δεν χρησιμοποιούνται ως σημεία αναφοράς και επιπλέον μειώνουν σημαντικά το σφάλμα παίρνοντας το μέσο όρο από δύο πλαίσια. Μπορούμε να πούμε ότι ο 'κύκλος της ζωής' τους περιορίζεται μόνο σε αυτά και δεν επεκτείνεται με το να κληροδοτούν πληροφορίες σε άλλα πλαίσια, κάτι που πολλές φορές σε συνδυασμό και με την υπολογιστική πολυπλοκότητα που απαιτούν για την κωδικοποίηση και αποκωδικοποίηση τα κάνει μη επιθυμητά από τους κατασκευαστές. Η διαδικασία της κωδικοποίησης περιλαμβάνει συνδυασμό των αντίστοιχων macroblocks που παρουσιάζουν μικρές διαφορές με τα αντίστοιχα των πλαισίων αναφοράς (προηγούμενο και επόμενο) δηλαδή αφαίρεση του μέσου όρου των άλλων δύο από το τρέχον πλαίσιο, συνδυασμό των διανυσμάτων κίνησης των πλαισίων αναφοράς (που συνδυάζονται όπως και τα αντίστοιχα macroblocks, δηλαδή λαμβάνεται ο μέσος όρος τους) και στη συνέχεια την ίδια διαδικασία με τα I και P πλαίσια για την κωδικοποίηση του macroblock που προκύπτει.

Ο κύριος λόγος ύπαρξης των B-πλαισίων είναι η κάλυψη της περίπτωσης κάποιες πληροφορίες της εικόνας να υπάρχουν σε επόμενα πλαίσια και να μην υπάρχουν στα προηγούμενα. Συνεπώς η πρόβλεψή τους με τα P πλαίσια θα ήταν αδύνατη. Σαν παράδειγμα μπορούμε να αναφέρουμε μία πόρτα που ανοίγει ξαφνικά. Η πληροφορία για το τι βρίσκεται πίσω από την πόρτα υπάρχει στα επόμενα πλαίσια

και όχι στα προηγούμενα και για να εμφανιστεί και στο τρέχον πλαίσιο πρέπει να ληφθούν σαν σημεία αναφοράς και το προηγούμενο και το επόμενο.

Αφού τα πλαίσια P κατασκευάζονται με βάση τα I και τα B με βάση τα I και P είναι προφανές ότι τα I πρέπει να έχουν σταλεί πριν τα αντίστοιχα P. Επίσης και τα P αλλά και τα I πρέπει να έχουν σταλεί πριν από τα αντίστοιχα B, παρόλο που στη μετάδοση αυτά παρεμβάλλονται ανάμεσά τους. Η σειρά με την οποία απεικονίζονται και η σειρά με την οποία αποστέλλονται τα πλαίσια (που προφανώς δεν είναι η ίδια) φαίνεται στο παρακάτω σχήμα :

Η συνηθέστερη διάταξη των πλαισίων σε ένα σήμα MPEG είναι η παρακάτω. Πολλές φορές παρεμβάλλονται περισσότερα B πλαίσια και τα I πλαίσια απέχουν περισσότερο μεταξύ τους (αλλά αυτό υποβαθμίζει την ποιότητα της εικόνας γιατί τα σφάλματα διαδίδονται περισσότερο).

Η μικρότερη μονάδα που μπορεί να αποκωδικοποιηθεί ανεξάρτητα ονομάζεται **GOP** (Group of Pictures) και περιέχει όλα τα I,P,B πλαίσια που χρειάζονται για την αποκωδικοποίηση, χωρίς να γίνονται αναφορές σε άλλο GOP.

Στον παρακάτω πίνακα φαίνεται η αναλογία πλαισίων I,P,B σε ένα σήμα MPEG :

Είδος εικόνας	Bit-rate	I	P	B	Μέσος όρος
MPEG-1	(1.15 Mbit/sec)	150,000	50,000	20,000	38,000
MPEG-2	(4.00Mbit/sec)	400,000	200,000	80,000	130,000

Table 1.2 Bitrate of MPEG-1 and MPEG-2

1.2. SIMD

Πράξεις

Η όλη διαδικασία της συμπίεσης και της αποσυμπίεσης βίντεο είναι μια διαδικασία που κοστίζει πολύ σε επεξεργαστή και απαιτεί αρκετούς κύκλους μηχανής καθώς όταν μιλάμε για αναλύσεις της τάξης των 720x480 ,μια τυπική ανάλυση DVD, θα πρέπει κάθε πράξη να γίνεται για κάθε pixel του κάθε frame!!!

Η ιδέα!!!

Κάθε pixel αναπαριστάται από τρεις τιμές: Red, Green, Blue (RGB). Όπως είναι γνωστό η τιμή του κάθε χρώματος είναι ένας ακέραιος αριθμός μεταξύ του 0 και του 255 που σημαίνει ότι αρκούν 8 bit για την αναπαράστασή του. Και αν είχα

καταχωρητές μεγέθους 64 bit και μπορούσα να φορτώσω 8 αριθμούς σε ένα register; Δεν θα ήτανε περίπου 8 φορές γρηγορότερο;

Η απάντηση

Την απάντηση σε αυτό το ερώτημα έδωσε η Intel το 1996 με την κατασκευή των επαναστατικών για την εποχή επεξεργαστών Pentium MMX, που αλλάξανε όλη την τεχνολογία γύρω από τις multimedia εφαρμογές και πλέον δόθηκε μία άλλη διάσταση στον τομέα αυτό.

1.2.1. **Τι είναι MMX**

Οι επεξεργαστές της Intel από εποχής Pemtium 1 και μετά περιέχουν 8 καταχωρητές με ονομασία mm0,mm1,…,mm7 οι οποίοι έχουνε μέγεθος 64 bit. Οι καταχωρητές αυτοί δίνουνε τη δυνατότητα αντί να χρησιμοποιηθούνε για να φορτώσουνε ένα αριθμό 64bit να φορτώσουνε 2 αριθμούς των 32 bit , 4 των 16, είτε 8 των bit.

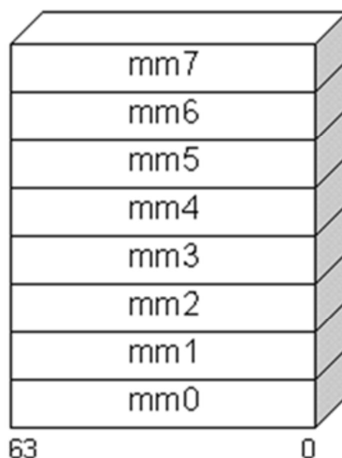


Image 1.5 Registers of MMX

Μέσω των πράξεων SIMD (Single Instruction, is a technique employed to achieve data level parallelism) επιτεύχθηκε παραλληλισμός των πράξεων.

Οι επεξεργαστές υποστηρίζουνε μόνο πράξεις ακεραίων αριθμών ενώ αργότερα μέσω νέων επεξεργαστών υποστηρίχθηκαν και πράξεις αριθμών κινητής υποδιαστολής.

1.2.2. Η συνέχεια(SSE,SSE2,…)

Το 1999 και ενώ το αντίπαλο δέος (AMD) έχει δημιουργήσει ένα αντίστοιχο σετ εντολών με τα SIMD, το 3DNOW , η Intel βγάζει στην αγορά νέους επεξεργαστές (Pentium 3) και πλέον το πρότυπο από MMX μετονομάζεται σε SSE (Streaming SIMD Extensions). Πλέον οι καταχωρητές είναι 128 bit και υποστηρίζουνε πράξεις τόσο ακεραίων όσο και κινητής υποδιαστολής. Οι καταχωρητές πάλι είναι 8 στον αριθμό αλλά αλλάζει η ονομασία τους από MMX σε XMMX.

Οι επιτρεπτές πράξεις είναι:

- 4 32-bit single-precision floating point numbers
- 2 64-bit double-precision floating point numbers
- 2 64-bit integers
- 4 32-bit integers
- 8 16-bit short integers
- 16 8-bit bytes or characters.

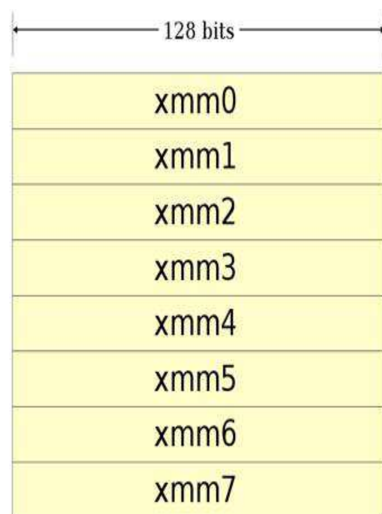


Image 1.6 Registers of SSE

Στη συνέχεια η Intel συνέχισε στον ίδιο δρόμο προχωρώντας στο SSE2,SSE3,SSSE3(supplemental SSE3) και SSE4 σε καθένα από τα οποία και προσθέτει κάποιες νέες εντολές στο instruction set ωστόσο η φιλοσοφία παραμένει η ίδια και οι διαφορές είναι πολύ μικρές από γενιά σε γενιά.

Σημείωση!!!

Πρέπει να σημειωθεί ότι κάθε νεότερη γενιά επεξεργαστών που υποστηρίζει επόμενης γενιάς instruction set για SIMD, υποστηρίζει και το instruction set των επεξεργαστών προηγούμενης γενιάς.

1.3.1 Intel atom

Η Intel στην προσπάθειά της να ακολουθήσει την τάση της αγοράς για συσκευές multimedia δημιούργησε τον δικό της επεξεργαστή με χαμηλή κατανάλωση. Αυτός ο επεξεργαστής έλαβε το όνομα Atom και η πρώτη του παρουσίαση έγινε στις 2 Μαρτίου του 2008. Ο επεξεργαστής Atom είναι ουσιαστικά ένας μονοπύρηνος (single core) επεξεργαστής x86 με την ειδοποιό διαφορά της εξαιρετικά χαμηλής κατανάλωσης σε σχέση με τους συμβατικούς x86 της Intel. Η υλοποίησή του είναι στα 45nm κάτι που του δίνει την ευελιξία να μη θερμαίνεται σχετικά εύκολα με αποτέλεσμα να είναι μία εξαιρετική επιλογή για χρήση σε netbooks.

1.3.2. Το παρόν και το μέλλον

Η εισαγωγή των netbook στην αγορά στέφθηκε με απόλυτη επιτυχία καθώς πλέον οι πωλήσεις netbook για το 2010 υπολογίζεται να είναι περίπου στο ¼ των συνολικών πωλήσεων laptop. Αυτό το γεγονός άνοιξε την όρεξη και άλλων εταιρειών με παράδοση στο χώρο των επεξεργαστών για φορητές συσκευές. Η εταιρεία Freescale κατασκεύασε ένα επεξεργαστή με πυρήνα arm cortex A8 τον οποίο και προορίζει για την αγορά των netbook και όχι μόνο.

Η Intel φοβούμενη τις εξελίξεις αναγκάστηκε να παρουσιάσει την επόμενη γενιά, που υπόσχεται καλύτερες επιδόσεις και ακόμα μικρότερη κατανάλωση ρεύματος.

Η τωρινή γενιά Atom όπου και βασίστηκε και αυτή η εργασία είναι η σειρά N200 των Atom ενώ η σειρά που τώρα κάνει το ντεμπούτο της είναι η σειρά N450(11 Ιανουαρίου 2010).

Model Number	sSpec Number	Frequency	L2 Cache	FSB	Voltage	TDP	Socket	Release Date	Release Price (USD)
Atom N270	SLB73 (C0)	1600 MHz	512 KB	533 MT/s	0.9 - 1.1625 V	2.5 W	BGA 437	June 3, 2008	$44
Atom N280	SLGL9 (C0)	1667 MHz	512 KB	667 MT/s	0.9 - 1.1625 V	2.5 W	BGA 437	February 6, 2009	

Table 1.3 Atom N270 and Atom N280 specifications

Model Number	sSpec Number	Frequency	L2 Cache	Memory	Voltage	TDP	Socket	Release Date	Release Price (USD)
Atom N450	SLBMG (A0)	1667 MHz	512 KB	1 x DDR2-667	0.8 - 1.175 V	5.5 W	micro-FCBGA8 559	December 21, 2009	$63
Atom N470		1833 MHz	512 KB	1 x DDR2-667	0.8 - 1.175 V	5.5 W	micro-FCBGA8 559	Q1, 2010	

Table 1.4 Atom N450 and Atom N470 specification

1.3.3. Απόδοση

Ένας επεξεργαστής Atom N230 χρονισμένος στα 1.6 GHz εκτελεί 3300 mips σε ένα τυπικό benchmark. Αντίστοιχα ο Pentium M 740 που είναι χρονισμένος στα 1,73 GHz αποδίδει 7400 mips. Η παραπάνω σύγκριση δείχνει ότι ο Atom είναι περίπου μισής ισχύος από έναν Pentium M χρονισμένο στην ίδια συχνότητα.

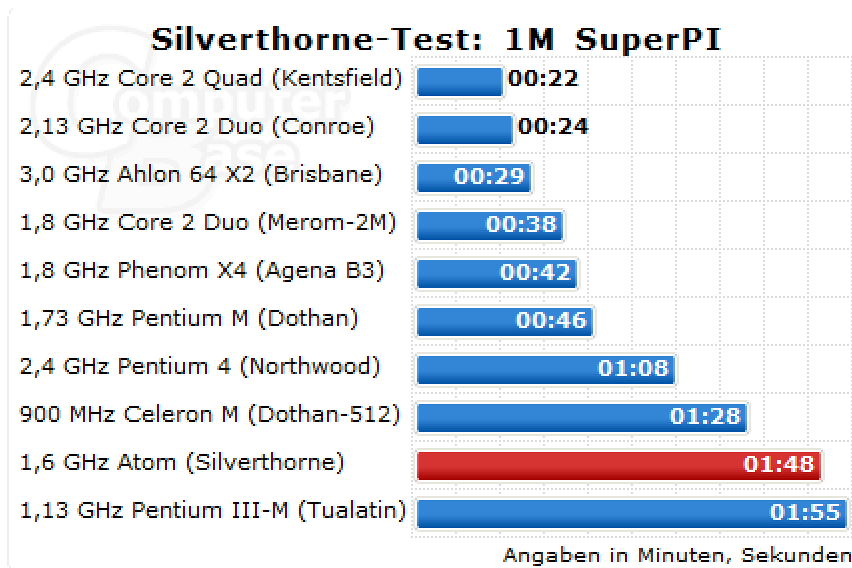


Image 1.7 Atom vs other intel's processors

1.4 Dell precision latitude 2100

Το netbook πάνω στο οποίο βασίστηκε η εργασία αυτή είναι ένα netbook της Dell, βασιζόμενο στον επεξεργαστή Intel Atom N270. Όπως προαναφέρθηκε ο επεξεργαστής αυτός έχει συχνότητα λειτουργίας τα 1.6 Ghz. Η μνήμη ram με την οποία είναι εξοπλισμένο το netbook είναι 1GB DDR2, ενώ είναι εξοπλισμένο με σκληρό δίσκο χωρητικότητας 160GB και ταχύτητα περιστροφής 5400 rpm. Τέλος, το λειτουργικό σύστημα που χρησιμοποιήθηκε είναι Windows XP sp2 και το compile του κώδικα έγινε σε Visual studio 2008.

Image 1.8 Dell latitude 2100

1.5.1. Microsoft Visual Studio

Το Microsoft visual studio είναι ένα περιβάλλον ανάπτυξης εφαρμογών κατασκευασμένο από τη Microsoft. Ο προγραμματιστής μπορεί να αναπτύξει τόσο προγράμματα κονσόλας όπως αποκαλούνται(δηλαδή χωρίς γραφικό περιβάλλον), αλλά και προγράμματα με γραφικό περιβάλλον ή αλλιώς GUI (graphical user interface) . Η υποστήριξη όσον αφορά τις γλώσσες προγραμματισμού που παρέχει είναι πολύ καλή καθώς υποστηρίζει τις πλέον διαδεδομένες γλώσσες προγραμματισμού όπως C/C++, VisualBasic.NET, C#, F#, M, Python, XML/XSLT, HTML/XHTML, Javascript και CSS.

1.5.2. Debugger

Σε μία εφαρμογή όπως το βελτιστοποίηση ενός κώδικα μέσω SIMD καταχωρητών είναι αναγκαία η χρήση ενός debugger ευέλικτου και εύκολα προσαρμόσιμου στις ιδιαιτερότητες του κώδικα. Ο debugger του Visual Studio είναι ο πλέον σύγχρονος και ενδεδειγμένος για τέτοιες εφαρμογές. Μερικά από τα χαρακτηριστικά του είναι:

- Disassembly: δίνει τη δυνατότητα στο χρήστη να κάνει disassembly, δηλαδή να μετατρέψει τον κώδικα μιας γλώσσας πχ της C σε

assembly code ώστε να μπορεί ο προγραμματιστής να βελτιώσει τον κώδικα.

- Breakpoints: ο προγραμματιστής μπορεί να βάζει breakpoints μέσα στον κώδικα και να κάνει μια παύση του προγράμματος σε εκείνο το σημείο για να ελέγξει τις τιμές των μεταβλητών.
- Edit and continue: εντυπωσιακό χαρακτηριστικό καθώς δίνει τη δυνατότητα να αλλάξουμε κάτι στον κώδικα ενώ κάνουμε debugging και κατόπιν να συνεχίσουμε (μόνο για 32bit επεξεργαστές).
- Hover over by mouse: κατά τη διάρκεια του debugging μπορεί ο χρήστης να μάθει την τιμή μιας μεταβλητής τη δεδομένη χρονική στιγμή βάζοντας το ποντίκι πάνω στη μεταβλητή.

1.5.3. Χρήσιμο εργαλείο(MSDN Library)

Το εργαλείο που με έσωσε ουκ ολίγες φορές είναι to MSDN library. Το MSDN library είναι ένα online εργαλείο όπου κάνοντας search βρίσκεις πληθώρα συναρτήσεων και εντολών. Αρκεί να αναφέρω πως περιέχει επεξήγηση για όλα τα intrinsic των SSE κάτι που μου φάνηκε αρκετά χρήσιμο καθώς δε χρειαζότανε να ανατρέξω σε manual της Intel παρά μόνο ελαχίστων περιπτώσεων.

1.5.4. Visual studio 2008 professional

Το visual studio 2008 είναι ο αντικαταστάτης του visual studio 2005. Οι διαφορές στο οπτικό κομμάτι δεν είναι μεγάλες αν και το 2008 είναι σαφώς πιο εργονομικό. Ωστόσο από την άλλη ο compiler είναι εντελώς καινούριος και με πολλές βελτιστοποιήσεις όπως υποστηρίζει η Microsoft αλλά και όπως διαπίστωσα και ο ίδιος τρέχοντας τα .exe που προέκυπταν από κάθε ένα compiler. Αξίζει να αναφέρω ότι δεν κατάφερα να υποστηρίξω τη βιβλιοθήκη SSSE3 στο visual studio 2005 και για αυτό στράφηκα στη χρήση του 2008.

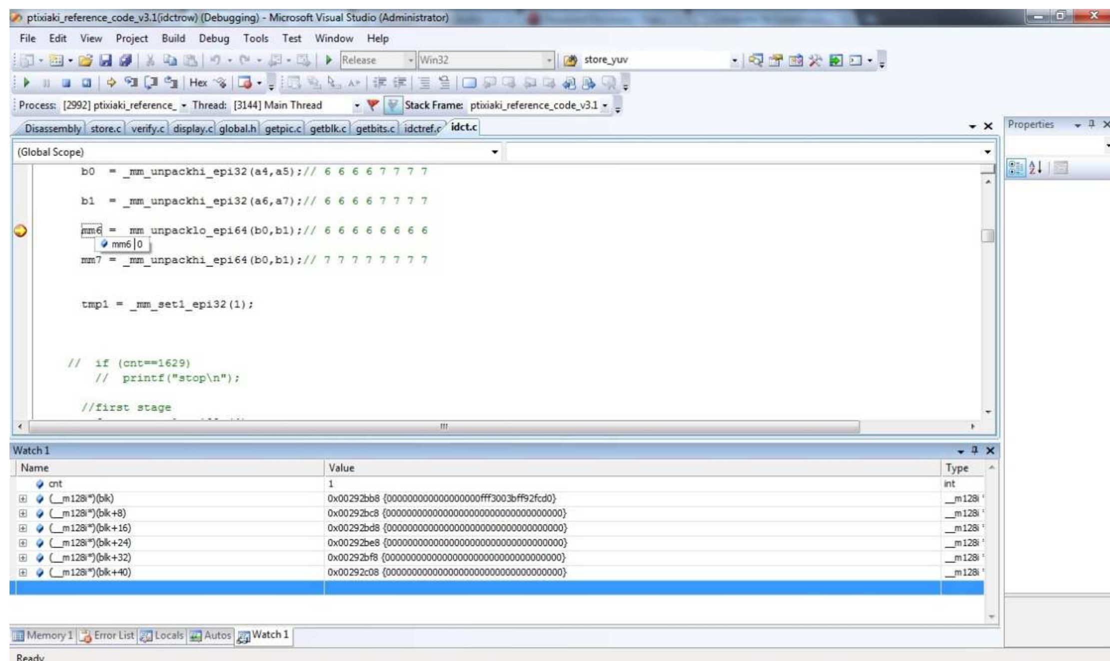


Image 1.9 Interface of Visual Studio 2008

1.6.1. Πως μετράμε απόδοση;

Ένα πρόβλημα που αντιμετωπίζουν οι προγραμματιστές όταν προσπαθούν να βελτιώσουν ως προς την ταχύτητα εκτέλεσης ένα πρόγραμμα είναι η βελτίωση που μόλις κάνανε πόσο ουσιαστική διαφορά έχει από τον προηγούμενο κώδικα. Η λύση σε αυτό το πρόβλημα όσον αφορά τουλάχιστον τους επεξεργαστές της Intel δόθηκε από την ίδια την Intel με την εφαρμογή Vtune.

Το Vtune είναι μία performance analyzer εφαρμογή για επεξεργαστές x86. Είναι φτιαγμένο από την Intel και υποστηρίζει όλους τους επεξεργαστές της. Το Vtune περιέχει πολλά είδη profiling με κυριότερα το time-based και το event-based. Ο profiler παρουσιάζει το χρόνο που ξοδεύτηκε σε κάθε υπορουτίνα του προγράμματος που κάνουμε profiling, κάτι που δίνει τη δυνατότητα στο χρήστη να γνωρίζει ανά πάσα στιγμή που «πάσχει» η εφαρμογή του και να διορθώσει-βελτιώσει άμεσα τον κώδικά του.

1.6.2. Vtune

Χαρακτηριστικά Vtune:

- Call graph: είναι η γραφική άποψη της ροής των συναρτήσεων μιας εφαρμογής. Έτσι ο χρήστης μπορεί να δει τη ροή του προγράμματος το οποίο θέλει να βελτιώσει και να ανακαλύψει τυχόν λάθη παραλήψεις στο σχεδιασμό του.
- Time based sampling: βρίσκει μετρώντας το χρόνο πόσο χρόνο καταναλώνει κάθε συνάρτηση και έτσι ο χρήστης ξέρει που χωλαίνει η εφαρμογή του.
- Event based sampling: βρίσκει τα σημεία όπου στο πρόγραμμα έχω cache failure, μη πρόβλεψη διακλάδωσης (branch mis-prediction)
- Source view: τα αποτελέσματα του sampling παρουσιάζονται γραμμή-γραμμή τόσο πηγαίου κώδικα όσο και κώδικα assembly για περαιτέρω βελτίωση και ώστε ο χρήστης να αντιληφθεί επακριβώς ποια γραμμή κώδικα μπορεί να βελτιώσει.
- Counter monitor: παρουσιάζει στο χρήστη πληροφορίες του συστήματος, πχ πόση ενέργεια δαπανήθηκε κατά την εκτέλεση της εφαρμογής του.
- Intel thread profiler: παρουσιάζει μια γραμμή εκτέλεσης ανάλογα με το χρόνο εκτέλεσης (time line execution), όπου φαίνεται πότε και τι κάνει κάθε νήμα όταν εκτελείται καθώς και την αλληλεπίδραση των νημάτων μεταξύ τους.

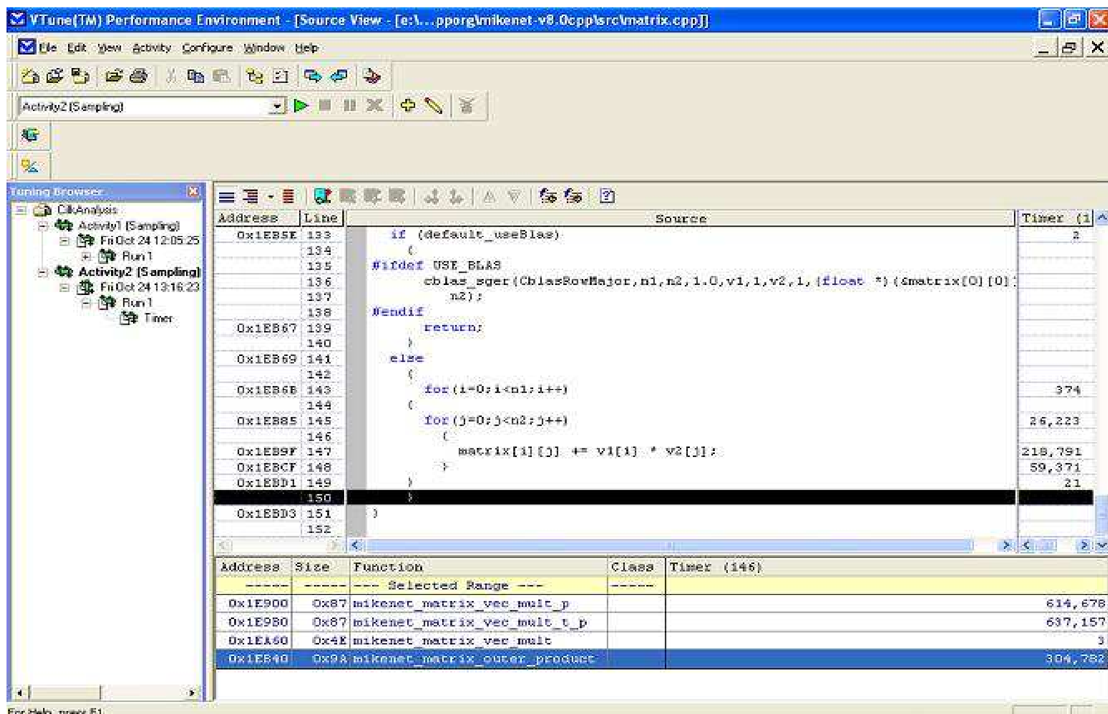


Image 1.10 Vtune’s environment

1.7.1. Τι είναι Intrinsics

Κατά τη βελτιστοποίηση multimedia εφαρμογών και στην περίπτωσή μας του MPEG-2, όπως είπαμε και παραπάνω είναι αναγκαία η χρήση των SIMD από τα πακέτα της οικογένειας MMX, SSE κτλ. Ωστόσο μέχρι πρότινος για να γράψει ένας προγραμματιστής κώδικα έπρεπε να γράφει τον κώδικα σε assembly. Αυτό είναι κάτι το εξαιρετικά χρονοβόρο καθώς θα πρέπει ο προγραμματιστής να κάνει σχεδόν για κάθε πράξη store, load κτλ. Επίσης λόγω της ιδιομορφίας του προγραμματισμού SSE (υπενθύμιση: έχω μόνο 8 καταχωρητές) θα πρέπει ο προγραμματιστής να κάνει ο ίδιος load και store σε συγκεκριμένους registers και να θυμάται-ελέγχει την εμβέλεια κάθε καταχωρητή ώστε να μπορεί να χρησιμοποιήσει καταχωρητή που είχε χρησιμοποιήσει προηγουμένως και πλέον η εμβέλειά του έχει λήξει.

Τη λύση ήρθε να δώσει η Intel με τη χρήση των Intrinsics. Τα Intrinsics είναι ουσιαστικά κάποιες συναρτήσεις που υλοποιούν πράξεις σε επίπεδο assembly χωρίς να είναι πιο αργές(μόλις 1-2 %) από τον αυθεντικό κώδικα assembly. Επίσης αναλαμβάνει ο compiler να κάνει δέσμευση και spillover τους καταχωρητές. Έτσι ο προγραμματιστής δε χρειάζεται να ανησυχεί για τους καταχωρητές που δεσμεύει και την εμβέλειά τους καθώς ο compiler κάνει μόνος του spillover και έτσι λύνονται τα χέρια του προγραμματιστή.

1.7.2. Intrinsics MMX

Η υποστήριξη των Intrinsic αφορά και το πακέτο εντολών MMX. Όπου δεν είναι αναγκαία η χρήση 128 bit καταχωρητών μπορεί ο προγραμματιστής να χρησιμοποιήσει τους 64 bit καταχωρητές που του προσφέρει το MMX και είναι πιο γρήγοροι στις πράξεις. Για να χρησιμοποιήσει ο προγραμματιστής τα SIMD που του παρέχει το πακέτο MMX αρκεί μα κάνει include στο Visual Studio τη βιβλιοθήκη mmintrin.h(include mmintrin.h).

Παρακάτω είναι ένας πίνακας όπου φαίνονται κάποιες από τις πράξεις

Intrinsic Name	Operation	Corresponding MMX Instruction
_mm_empty	Empty MM state	EMMS
_mm_cvtsi32_si64	Convert from int	MOVD
_mm_cvtsi64_si32	Convert to int	MOVD
_mm_cvtsi64_m64	Convert from __int64	MOVQ
_mm_cvtm64_si64	Convert to __int64	MOVQ
_mm_packs_pi16	Pack	PACKSSWB
_mm_packs_pi32	Pack	PACKSSDW
_mm_packs_pu16	Pack	PACKUSWB
_mm_unpackhi_pi8	Interleave	PUNPCKHBW
_mm_unpackhi_pi16	Interleave	PUNPCKHWD
_mm_unpackhi_pi32	Interleave	PUNPCKHDQ
_mm_unpacklo_pi8	Interleave	PUNPCKLBW
_mm_unpacklo_pi16	Interleave	PUNPCKLWD
_mm_unpacklo_pi32	Interleave	PUNPCKLDQ
_mm_add_pi8	Addition	PADDB
_mm_add_pi16	Addition	PADDW
_mm_add_pi32	Addition	PADDD
_mm_adds_pi8	Addition	PADDSB
_mm_adds_pi16	Addition	PADDSW
_mm_adds_pu8	Addition	PADDUSB
_mm_adds_pu16	Addition	PADDUSW
_mm_sub_pi8	Subtraction	PSUBB
_mm_sub_pi16	Subtraction	PSUBW
_mm_sub_pi32	Subtraction	PSUBD
_mm_subs_pi8	Subtraction	PSUBSB
_mm_subs_pi16	Subtraction	PSUBSW
_mm_subs_pu8	Subtraction	PSUBUSB
_mm_subs_pu16	Subtraction	PSUBUSW
_mm_madd_pi16	Multiply and add	PMADDWD
_mm_mulhi_pi16	Multiplication	PMULHW
_mm_mullo_pi16	Multiplication	PMULLW

Image 1.11 MMX intrinsics

1.7.3. Intrinsics SSE2

Για να δηλώσω στον compiler του Visual Studio ότι θα χρησιμοποιήσω τις βιβλιοθήκες του πακέτου εντολών SSE2, κάνω include τη βιβλιοθήκη emmintrin.h(#include “emmintrin.h”).

Εδώ παρουσιάζονται κάποια intrinscics της Intel και η λειτουργία που καθένα παρέχει στο πακέτο εντολών SSE2 που χρησιμοποιήθηκε στην εργασία.

Intrinsic Name	Operation	Corresponding SSE2 Instruction
_mm_cmpeq_pd	Equality	CMPEQPD
_mm_cmplt_pd	Less Than	CMPLTPD
_mm_cmple_pd	Less Than or Equal	CMPLEPD
_mm_cmpgt_pd	Greater Than	CMPLTPDr
_mm_cmpge_pd	Greater Than or Equal	CMPLEPDr
_mm_cmpord_pd	Ordered	CMPORDPD
_mm_cmpunord_pd	Unordered	CMPUNORDPD
_mm_cmpneq_pd	Inequality	CMPNEQPD
_mm_cmpnlt_pd	Not Less Than	CMPNLTPD
_mm_cmpnle_pd	Not Less Than or Equal	CMPNLEPD
_mm_cmpngt_pd	Not Greater Than	CMPNLTPDr
_mm_cmpnge_pd	Not Greater Than or Equal	CMPNLEPDr
_mm_cmpeq_sd	Equality	CMPEQSD
_mm_cmplt_sd	Less Than	CMPLTSD
_mm_cmple_sd	Less Than or Equal	CMPLESD
_mm_cmpgt_sd	Greater Than	CMPLTSDr
_mm_cmpge_sd	Greater Than or Equal	CMPLESDr
_mm_cmpord_sd	Ordered	CMPORDSD
_mm_cmpunord_sd	Unordered	CMPUNORDSD
_mm_cmpneq_sd	Inequality	CMPNEQSD
_mm_cmpnlt_sd	Not Less Than	CMPNLTSD
_mm_cmpnle_sd	Not Less Than or Equal	CMPNLESD
_mm_cmpngt_sd	Not Greater Than	CMPNLTSDr
_mm_cmpnge_sd	Not Greater Than or Equal	CMPNLESDr
_mm_comieq_sd	Equality	COMISD
_mm_comilt_sd	Less Than	COMISD
_mm_comile_sd	Less Than or Equal	COMISD
_mm_comigt_sd	Greater Than	COMISD
_mm_comige_sd	Greater Than or Equal	COMISD
_mm_comineq_sd	Not Equal	COMISD
_mm_ucomieq_sd	Equality	UCOMISD
_mm_ucomilt_sd	Less Than	UCOMISD
_mm_ucomile_sd	Less Than or Equal	UCOMISD
_mm_ucomigt_sd	Greater Than	UCOMISD
_mm_ucomige_sd	Greater Than or Equal	UCOMISD
_mm_ucomineq_sd	Not Equal	UCOMISD

Image 1.12 SSE2 intrinsics

2. Αλλαγές στον reference code για διευκόλυνση της εκτέλεσης

2.1. Σώσιμο ή όχι των αποκωδικοποιημένων frame

Στον reference code η συνάρτηση “store_yuv” αποθηκεύει στο σκληρό δίσκο σε τρία χωριστά αρχεία (.Y, .U, .V) τα αποτελέσματα των αποκωδικοποιημένων frame. Αρχικά, θα πρέπει να βρούμε ένα τρόπο για να αποθηκεύουμε και τα τρία αρχεία σε ένα ώστε να μπορούμε να δούμε το οπτικό αποτέλεσμα των αποκωδικοποιημένων frame. Αυτό είναι εφικτό με τη χρήση του flag “ **O_APPEND** ” στη συνάρτηση “open” την οποία χρησιμοποιούμε για να αποθηκεύσουμε το αρχείο. Το flag αυτό σημαίνει ότι όταν μια συνάρτηση κάνει open το ίδιο αρχείο τότε δε σβήνονται τα προηγούμενα δεδομένα, παρά συνεχίζει από το σημείο στο οποίο σταμάτησε όταν κλήθηκε προηγουμένως (η open) με αποτέλεσμα με τρεις διαδοχικές κλήσεις της store_yuv1 να έχω αποθήκευση διαδοχικά των Y, U και V.

Μπορεί από τη μία πλευρά να μου είναι χρήσιμο το σώσιμο των frame για να μπορώ να διαπιστώσω το αποτέλεσμα της αποκωδικοποίησης από την άλλη πλευρά ωστόσο κατά τη μέτρηση της απόδοσης δεν θα πρέπει να αποθηκεύονται τα frame καθώς θα έχω καθυστέρηση κατά την αποθήκευση των frame στο σκληρό δίσκο. Από τα παραπάνω συνεπάγεται ότι θα πρέπει να εισαχθεί στον κώδικα μια δομή κατά την οποία θα αποθηκεύονται ή όχι τα αποκωδικοποιημένα frame.

Για την αποθήκευση ή όχι του frame χρησιμοποιείται μια καθολική μεταβλητή (**write_flag**) η οποία αν έχει τιμή 1 τότε τα frame θα αποθηκεύονται στο σκληρό δίσκο. Η τιμή της μεταβλητής καθορίζεται από την είσοδο που εισάγει ο χρήστης στη γραμμή εντολών για την εκτέλεση του προγράμματος αποκωδικοποίησης. Έτσι αν ο χρήστης εισάγει το σύμβολο –S τότε ο decoder αποθηκεύει τα αποκωδικοποιημένα frame, σε αντίθετη περίπτωση τα frame δεν αποθηκεύονται και ο χρήστης μπορεί να κάνει αξιόπιστες μετρήσεις.

2.2. Αποθήκευση νέου YUV χωρίς να χρειάζεται σβήσιμο του παλιού YUV

Στην προηγούμενη παράγραφο εξηγήθηκε πως γίνεται να δημιουργηθεί ένα YUV αρχείο ώστε να μπορεί ο χρήστης να βλέπει εύκολα τα αποτελέσματα της διαδικασίας της αποκωδικοποίησης με τη χρήση του flag “**O_APPEND**” . Ωστόσο το μειονέκτημα που προκύπτει από τη χρήση του παραπάνω flag είναι ότι κάθε φορά που ο χρήστης τρέχει τον decoder θα πρέπει να σβήνει τυχόν προηγούμενα αρχεία εξόδου καθώς κατά τη νέα εκτέλεση γίνεται συγκόλληση (merge) των προηγούμενων frame, με τα frame που προκύπτουν από τη νέα εκτέλεση. Αυτό μπορεί να επιτευχθεί με τη χρήση του flag “**O_TRUNC**” στη συνάρτηση open.

Συνοψίζοντας τα παραπάνω για να έχω την προσδοκώμενη λειτουργία του decoder αρκεί την πρώτη φορά που θα κληθεί η open να χρησιμοποιώ το flag “ **O_TRUNC**” και στη συνέχεια να χρησιμοποιώ το flag “**O_APPEND**”. Αυτό είναι εφικτό αν χρησιμοποιώ μια καθολική μεταβλητή (append) η οποία αρχικοποιείται με 0 και στην περίπτωση που η μεταβλητή αυτή είναι 0 (πρώτη εκτέλεση) να εκτελείται η open με flag “**O_TRUNC**”. Κατόπιν η μεταβλητή αρχικοποιείται με 1 και έτσι από εκεί και έπειτα έχω εκτέλεση της open με flag “**O_APPEND**” όπου και γίνεται το “merge” των YUV.

Παρακάτω παρουσιάζεται ο πηγαίος κώδικας του αρχείου store.c όπου και γίνεται η υλοποίηση των παραπάνω(με υπογράμμιση κίτρινη τονίζονται οι αλλαγές του κώδικα).

Αρχικοποίηση της μεταβλητής append

```
int append = 0;

/*
 * store a picture as ei                          lds
 */
void Write_Frame(src,frame)
unsigned char *src[];
int frame;
{
  char outname[FILENAME_LENGTH];

  if (progressive_sequence || progressive_frame || Frame_Store_Flag)
  {
    /* progressive */
    sprintf(outname,Output_Picture_Filename,frame,'f');
    store_one(outname,src,0,Coded_Picture_Width,vertical_size);
  }
  else
  {
    /* interlaced */
    sprintf(outname,Output_Picture_Filename,frame,'a');
    store_one(outname,src,0,Coded_Picture_Width<<1,vertical_size>>1);

    sprintf(outname,Output_Picture_Filename,frame,'b');
    store_one(outname,src,
      Coded_Picture_Width,Coded_Picture_Width<<1,vertical_size>>1);
  }
}
```

```
/*
 * store one frame or one field
 */
static void store_one(outname,src,offset,incr,height)
char *outname;
unsigned char *src[];
int offset, incr, height;
{
        //save or no
        if(write_flag == 1)
        {
                  switch (Output_Type)
                  {
                  case T_YUV:
                        store_yuv(outname,src,offset,incr,height,0);
                        break;
                  case T_SIF:
                        store_sif(outname,src,offset,incr,height);
                        break;
                  case T_TGA:
                        store_ppm_tga(outname,src,offset,incr,height,1);
                        break;
                  case T_PPM:
                        store_ppm_tga(outname,src,offset,incr,height,0);
                        break;
                #ifdef DISPLAY
                  case T_X11:
                        dither(src);
                        break;
                #endif
                  default:
                        break;
                  }
        }

}

/* separate headerless files for y, u and v */
static void store_yuv(outname,src,offset,incr,height,Flag_Write)
char *outname;
unsigned char *src[];
int offset,incr,height;
{
  int hsize;
  char tmpname[FILENAME_LENGTH];

  hsize = horizontal_size;

 sprintf(tmpname,"%s.YUV",outname);
 store_yuv1(tmpname,src[0],offset,incr,hsize,height);

 append = 1;

  if (chroma_format!=CHROMA444)
  {
    offset>>=1; incr>>=1; hsize>>=1;
  }

  if (chroma_format==CHROMA420)
  {
    height>>=1;
  }

 // sprintf(tmpname,"%s.U",outname);
  store_yuv1(tmpname,src[1],offset,incr,hsize,height);

 // sprintf(tmpname,"%s.V",outname);
  store_yuv1(tmpname,src[2],offset,incr,hsize,height);


}
```

Έλεγχος για το αν πρέπει να έχω σώσιμο ή όχι των αποκωδικοποιημένων frame

Εκτέλεση της store_yuv1 για 1η φορά (append = 0)

Append = 1, μετά την 1η εκτέλεση ώστε να παραχθεί ένα αρχείο YUV

```
/* auxiliary routine */
static void store_yuv1(name,src,offset,incr,width,height)
char *name;
unsigned char *src;
int offset,incr,width,height;
{
  int i, j;
  unsigned char *p;

  //if (!Quiet_Flag)
   // fprintf(stderr,"saving %s\n",name);


  //change it so as to store in one file YUV and when running again
  //overwrite existing file
  if (append == 1)
  {

        if ((outfile = open(name,O_CREAT|O_APPEND|O_WRONLY|O_BINARY,0666))==-1)
        {
              sprintf(Error_Text,"Couldn't create %s\n",name);
              Error(Error_Text);
        }
  }

  else
  {

        if ((outfile = open(name,O_CREAT|O_TRUNC|O_WRONLY|O_BINARY,0666))==-1)
        {
              sprintf(Error_Text,"Couldn't create %s\n",name);
              Error(Error_Text);
        }

  }
```

Έλεγχος για το αν η store_yuv1 εκτελείται για για 1η φορά και έτσι αποφασίζεται το είδος της open που θα κληθεί

Χρήση των flag O_APPEND ή O_TRUNC ανάλογα με την περίσταση

Παρακάτω παρουσιάζεται η αλλαγή στη συνάρτηση Process_Options του αρχείου mpeg2dec.c, ώστε από τη γραμμή εντολών να μπορεί ο χρήστης χρησιμοποιώντας την επιλογή –S αν αποθηκεύει ή όχι τα παραγόμενα frame.

```
/* option processing */
static void Process_Options(argc,argv)
int argc;                   /* argument count  */
char *argv[];               /* argument vector */
{
  int i, LastArg, NextArg;

  /* at least one argument should be present */
  if (argc<2)
  {
    printf("\n%s, %s\n",Version,Author);
    printf("Usage:  mpeg2decode {options}\n\
Options: -b  file  main bitstream (base or spatial enhancement layer)\n\
         -cn file  conformance report (n: level)\n\
         -e  file  enhancement layer bitstream (SNR or Data Partitioning)\n\
         -f        store/display interlaced video in frame format\n\
                -n               no store of the video\n\
         -g        concatenated file format for substitution method (-x)\n\
         -in file  information & statistics report  (n: level)\n\
         -l  file  file name pattern for lower layer sequence\n\
                   (for spatial scalability)\n\
         -on file  output format (0:YUV 1:SIF 2:TGA 3:PPM 4:X11 5:X11HiQ)\n\
         -q        disable warnings to stderr\n\
         -r        use double precision reference IDCT\n\
         -t        enable low level tracing to stdout\n\
         -u  file  print user_data to stdio or file\n\
         -vn       verbose output (n: level)\n\
```

```
         -x  file  filename pattern of picture substitution sequence\n\n\
File patterns:  for sequential filenames, \"printf\" style, e.g. rec%%d\n\
                 or rec%%d%%c for fieldwise storage\n\
Levels:        0:none 1:sequence 2:picture 3:slice 4:macroblock 5:block\n\n\
Example:       mpeg2decode -b bitstream.mpg -f -r -o0 rec%%d\n\
         \n");
    exit(0);
  }


  Output_Type = -1;
  i = 1;

  /* command-line options are proceeded by '-' */

  while(i < argc)
  {
    /* check if this is the last argument */
    LastArg = ((argc-i)==1);

    /* parse ahead to see if another flag immediately follows current
       argument (this is used to tell if a filename is missing) */
    if(!LastArg)
      NextArg = (argv[i+1][0]=='-');
    else
      NextArg = 0;

    /* second character, [1], after '-' is the switch */
    if(argv[i][0]=='-')
    {
      switch(toupper(argv[i][1]))
      {
        /* third character. [2], is the value */
      case 'B':
        Main_Bitstream_Flag = 1;

        if(NextArg || LastArg)
        {
          printf("ERROR: -b must be followed the main bitstream filename\n");
        }
        else
          Main_Bitstream_Filename = argv[++i];

        break;
//no record of the file


      case 'C':

#ifdef VERIFY
        Verify_Flag = atoi(&argv[i][2]);

        if((Verify_Flag < NO_LAYER) || (Verify_Flag > ALL_LAYERS))
        {
          printf("ERROR: -c level (%d) out of range [%d,%d]\n",
            Verify_Flag, NO_LAYER, ALL_LAYERS);
          exit(ERROR);
        }
#else  /* VERIFY */
       // printf("This program not compiled for Verify_Flag option\n");
#endif /* VERIFY */
        break;

          case 'S':
                  write_flag = 1;
                  break;


      case 'E':
        Two_Streams  =  1;  /*  either  Data  Partitioning  (DP)  or  SNR  Scalability
enhancment */

        if(NextArg || LastArg)
        {
          printf("ERROR: -e must be followed by filename\n");
          exit(ERROR);
        }
```

Έλεγχος στη γραμμή εντολών για το αν πρέπει να αποθηκεύσω ή όχι στο σκληρο δίσκο τα αποκωδικοποιημένα frame (η μεταβλητή wrtite_flag είναι αρχικοποιημένη αρχικά με 0 και γίνεται 1 στην περίπτωση που υπάρχει option –S στη γραμμή εντολών οπότε έχω και αποθήκευση των παραγόμενων frame)

```
        else
          Enhancement_Layer_Bitstream_Filename = argv[++i];

        break;


      case 'F':
        Frame_Store_Flag = 1;
        break;

      case 'G':
        Big_Picture_Flag = 1;
        break;


      case 'I':
#ifdef VERIFY
        Stats_Flag = atoi(&argv[i][2]);
#else /* VERIFY */
        printf("WARNING: This program not compiled for -i option\n");
#endif /* VERIFY */
        break;

      case 'L':  /* spatial scalability flag */
        Spatial_Flag = 1;

       if(NextArg || LastArg)
       {
         printf("ERROR: -l must be followed by filename\n");
         exit(ERROR);
       }
       else
         Lower_Layer_Picture_Filename = argv[++i];

        break;

      case 'O':

        Output_Type = atoi(&argv[i][2]);

        if((Output_Type==4) || (Output_Type==5))
          Output_Picture_Filename = "";  /* no need of filename */
        else if(NextArg || LastArg)
        {
          printf("ERROR: -o must be followed by filename\n");
          exit(ERROR);
        }
        else
        /* filename is separated by space, so it becomes the next argument */
          Output_Picture_Filename = argv[++i];

        break;

      case 'Q':
        Quiet_Flag = 1;
        break;

      case 'R':
        Reference_IDCT_Flag = 1;
        break;

      case 'T':
#ifdef TRACE
        Trace_Flag = 1;
#else /* TRACE */
        printf("WARNING: This program not compiled for -t option\n");
#endif /* TRACE */
        break;

      case 'U':
        User_Data_Flag = 1;

      case 'V':
#ifdef VERBOSE
        Verbose_Flag = atoi(&argv[i][2]);
#else /* VERBOSE */
        printf("This program not compiled for -v option\n");
```

```
#endif /* VERBOSE */
        break;

      case 'X':
        Ersatz_Flag = 1;

       if(NextArg || LastArg)
       {
         printf("ERROR: -x must be followed by filename\n");
         exit(ERROR);
       }
       else
        Substitute_Picture_Filename = argv[++i];

        break;



      default:
        fprintf(stderr,"undefined option -%c ignored. Exiting program\n",
          argv[i][1]);

        exit(ERROR);

      } /* switch() */
    } /* if argv[i][0] == '-' */

    i++;

       /* check for bitstream filename argument (there must always be one, at the very
end
          of the command line arguments */

  } /* while() */


  /* options sense checking */

  if(Main_Bitstream_Flag!=1)
  {
    printf("There must be a main bitstream specified (-b filename)\n");
  }

  /* force display process to show frame pictures */
  if((Output_Type==4 || Output_Type==5) && Frame_Store_Flag)
    Display_Progressive_Flag = 1;
  else
    Display_Progressive_Flag = 0;


  /* no output type specified */
  if(Output_Type==-1)
  {
    Output_Type = 9;
    Output_Picture_Filename = "";
  }


}
```

3.Κόστος συναρτήσεων

3.1. Κόστος συναρτήσεων του reference code

Για να αποφασίσουμε ποιες θα είναι η συναρτήσεις τις οποίες και θα πρέπει να βελτιστοποιήσουμε και να επικεντρώσουμε τις προσπάθειές μας θα πρέπει να ξέρουμε το αρχικό κόστος των συναρτήσεων κατά την αποκωδικοποίηση ενός video. Αυτή η διαδικασία μπορεί να γίνει με τη χρήση του εργαλείου profiling Vtune η χρήση και οι ιδιότητες του οποίου έχουνε αναλυθεί στην παράγραφο 1.6.2.

Ωστόσο, το κόστος μιας συνάρτησης είναι άμεσα εξαρτώμενο και από το video με το οποίο γίνεται η δοκιμή. Για παράδειγμα ένα video με Bitrate 2Mbps θα είναι αρκετά πιο εύκολο στην αποκωδικοποίηση από ένα άλλο με 4 Mbps, αρκεί να σκεφτούμε ότι οι τιμές αρκετών pixel στο video των 2Mbps θα είναι μηδενικές λόγω της υψηλότερης συμπίεσης και άρα και αρκετές συναρτήσεις μπορούνε να παρακαμφθούν. Επίσης, αν ένα video είναι HD απαιτεί περίπου 8 φορές περισσότερη επεξεργαστική ισχύ με αποτέλεσμα οι μετρήσεις του να χρήζουν ιδιαίτερου ενδιαφέροντος.

Σε αυτή τη διπλωματική εργασία χρησιμοποιήθηκαν 3 video με διαφορετικά χαρακτηριστικά ώστε τα συμπεράσματα που θα εξαχθούν να είναι ασφαλέστερα και να καλύπτουν όσο το δυνατόν περισσότερες περιπτώσεις. Επίσης λόγω του ότι η διπλωματική αναπτύχθηκε σε επεξεργαστή Intel core2duo κυρίως λόγω μεγαλύτερης οθόνης, έχουνε γίνει και κάποιες μετρήσεις σε αυτόν τον επεξεργαστή τα οποία αναφέρονται κυρίως για εγκυκλοπαιδικές γνώσεις αλλά και λόγω περιέργειας του συγγραφέα της διπλωματικής αυτής εργασίας.

Τα χαρακτηριστικά των 3 video φαίνονται στον παρακάτω πίνακα.

	Ανάλυση	Bitrate	Frames
Dantes.m2v	SD (720x480)	2Mbps	1638
Input_inter.m2v	SD (720x480)	4 Mbps	2600
Vin.m2v	HD (1440x1080)	24 Mbps	451

Table 3.1 Video’s characteristics

3.1.1 Μέτρηση των fps που παράγονται από τον reference code στα 3 video

	Fps atom	Fps core2duo(2,4GHz)
Dantes.m2v	40	127
Input_inter.m2v	36,4	111
Vin.m2v	9,47	29,44

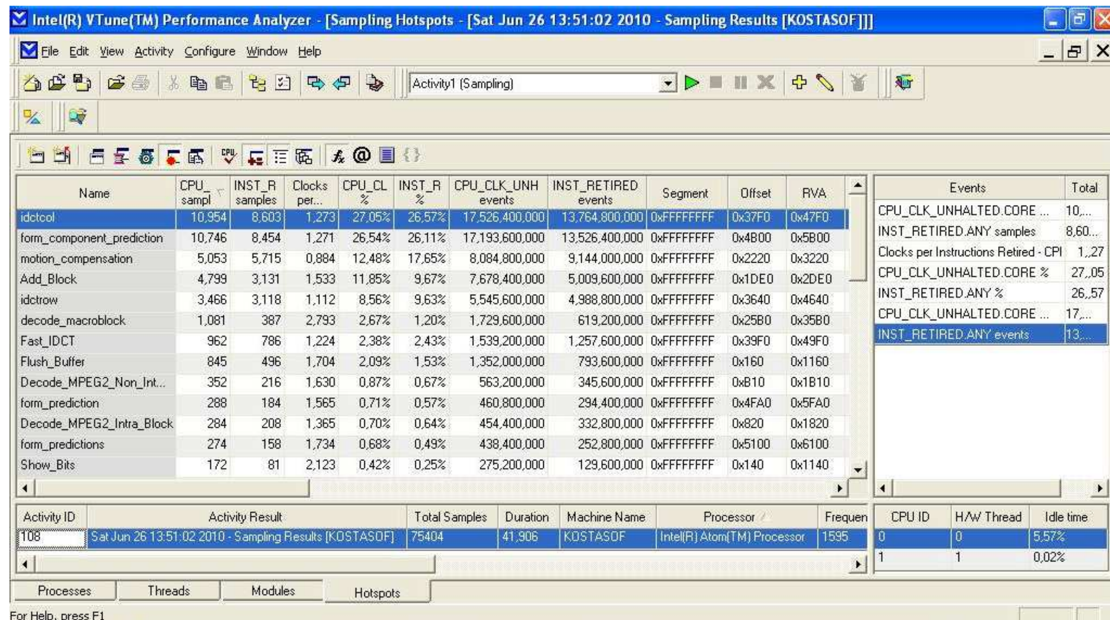
Table 3.2 Fps by using reference code

Από τον παραπάνω πίνακα παρατηρούμε ότι όσο μεγαλύτερο είναι το Bitrate πέφτει η απόδοση κάτι που είναι λογικό καθώς εκτελούνται περισσότερες πράξεις αποκωδικοποίησης(πτώση απόδοσης κατά 9%). Ωστόσο και στις δύο περιπτώσεις έχουμε real time αποκωδικοποίηση του video με ανάλυση SD (standard definition), αν και στην ανάλυση HD παρατηρούμε ότι για τον επεξεργαστή atom τα πράγματα είναι πραγματικά δύσκολα καθώς για να τρέξει σε real time θα πρέπει να πετύχουμε τουλάχιστον 3x, αλλά και πάλι ενδέχεται να τρέχει με κολλήματα. Στην πλευρά του core2duo τα πράγματα είναι καλύτερα καθώς οριακά ο reference code καταφέρνει να πετύχει real time αποκωδικοποίηση με αποτέλεσμα να είμαστε σχετικά αισιόδοξοι ότι στο τελικό αποτέλεσμα θα έχουμε ένα decoder ο οποίος θα τρέχει με σχετική άνεση video ανάλυσης HD(1440x1080).

Με βάση τα παραπάνω αποτελέσματα αρχικά ξεχνάμε την αποκωδικοποίηση HD και στρεφόμαστε στα δύο άλλα video με ανάλυση SD πάνω στα οποία θα μετράμε την απόδοση του atom με βάση τις αλλαγές στον κώδικα. Στο τέλος και έχοντας κάνει ότι καλύτερο μπορούμε θα προσπαθήσουμε να κάνουμε real time decoding του HD και εάν είναι εφικτό τότε θα έχουμε πετύχει ένα άκρως εντυπωσιακό αποτέλεσμα.

3.1.2 Dantes.m2v

Παρακάτω παρουσιάζεται ένα screenshot από το Vtune κατά τη διάρκεια εκτέλεσης του reference code MPEG-2 με σημείο αναφοράς το video Dantes.m2v.



Name	CPU_ sampl	INST_R samples	Clocks per...	CPU_CL %	INST_R %	CPU_CLK_UNH events	INST_RETIRED events	Segment	Offset	RVA
idctcol	10,954	8,603	1,273	27,05%	26,57%	17,526,400,000	13,764,800,000	0xFFFFFFFF	0x37F0	0x47F0
form_component_prediction	10,746	8,454	1,271	26,54%	26,11%	17,193,600,000	13,526,400,000	0xFFFFFFFF	0x4B00	0x5B00
motion_compensation	5,053	5,715	0,884	12,48%	17,65%	8,084,800,000	9,144,000,000	0xFFFFFFFF	0x2220	0x3220
Add_Block	4,799	3,131	1,533	11,85%	9,67%	7,678,400,000	5,009,600,000	0xFFFFFFFF	0x1DE0	0x2DE0
idctrow	3,466	3,118	1,112	8,56%	9,63%	5,545,600,000	4,988,800,000	0xFFFFFFFF	0x3640	0x4640
decode_macroblock	1,081	387	2,793	2,67%	1,20%	1,729,600,000	619,200,000	0xFFFFFFFF	0x25B0	0x35B0
Fast_IDCT	962	786	1,224	2,38%	2,43%	1,539,200,000	1,257,600,000	0xFFFFFFFF	0x39F0	0x49F0
Flush_Buffer	845	496	1,704	2,09%	1,53%	1,352,000,000	793,600,000	0xFFFFFFFF	0x160	0x1160
Decode_MPEG2_Non_Int...	352	216	1,630	0,87%	0,67%	563,200,000	345,600,000	0xFFFFFFFF	0xB10	0x1B10
form_prediction	288	184	1,565	0,71%	0,57%	460,800,000	294,400,000	0xFFFFFFFF	0x4FA0	0x5FA0
Decode_MPEG2_Intra_Block	284	208	1,365	0,70%	0,64%	454,400,000	332,800,000	0xFFFFFFFF	0x820	0x1820
form_predictions	274	158	1,734	0,68%	0,49%	438,400,000	252,800,000	0xFFFFFFFF	0x5100	0x6100
Show_Bits	172	81	2,123	0,42%	0,25%	275,200,000	129,600,000	0xFFFFFFFF	0x140	0x1140

Events	Total
CPU_CLK_UNHALTED.CORE ...	10,...
INST_RETIRED.ANY samples	8,60...
Clocks per Instructions Retired - CPI	1,,27
CPU_CLK_UNHALTED.CORE %	27,,05
INST_RETIRED.ANY %	26,,57
CPU_CLK_UNHALTED.CORE ...	17,...
INST_RETIRED.ANY events	13,...

Activity ID	Activity Result	Total Samples	Duration	Machine Name	Processor	Frequen
108	Sat Jun 26 13:51:02 2010 - Sampling Results [KOSTASOF]	75404	41,906	KOSTASOF	Intel(R) Atom(TM) Processor	1595

CPU ID	H/W Thread	Idle time
0	0	5,57%
1	1	0,02%

Image 3.1 performance of reference code in Dantes.m2v

Παρακάτω παρουσιάζεται το διάγραμμα των συναρτήσεων που χρησιμοποιούνται κατά τη διάρκεια αποκωδικοποίησης του video Dantes.m2v.

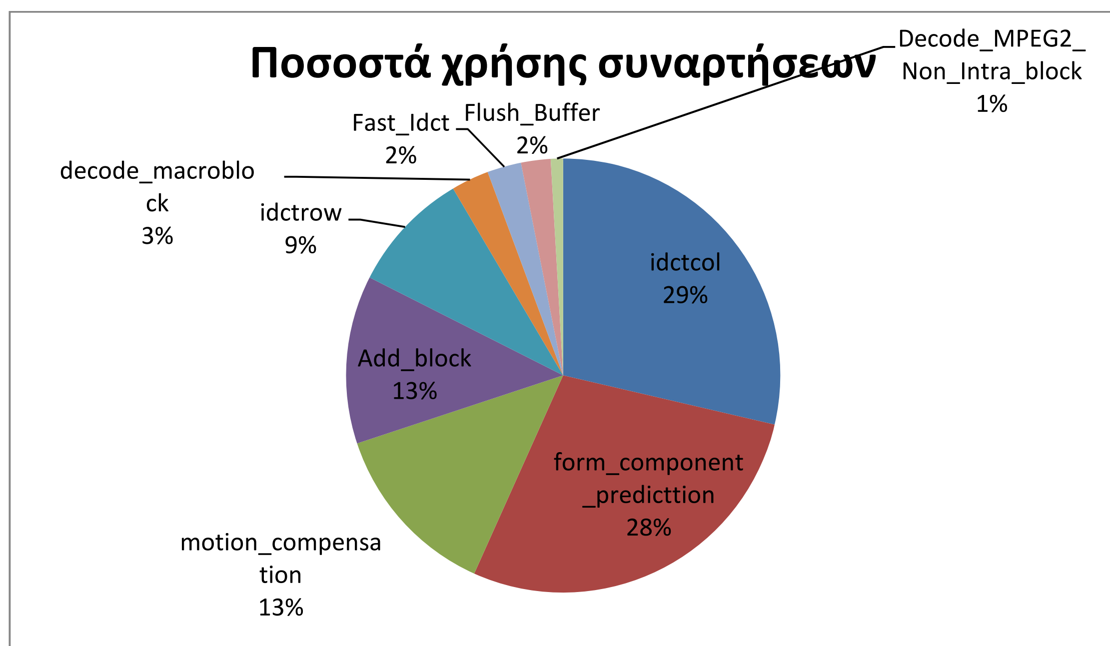
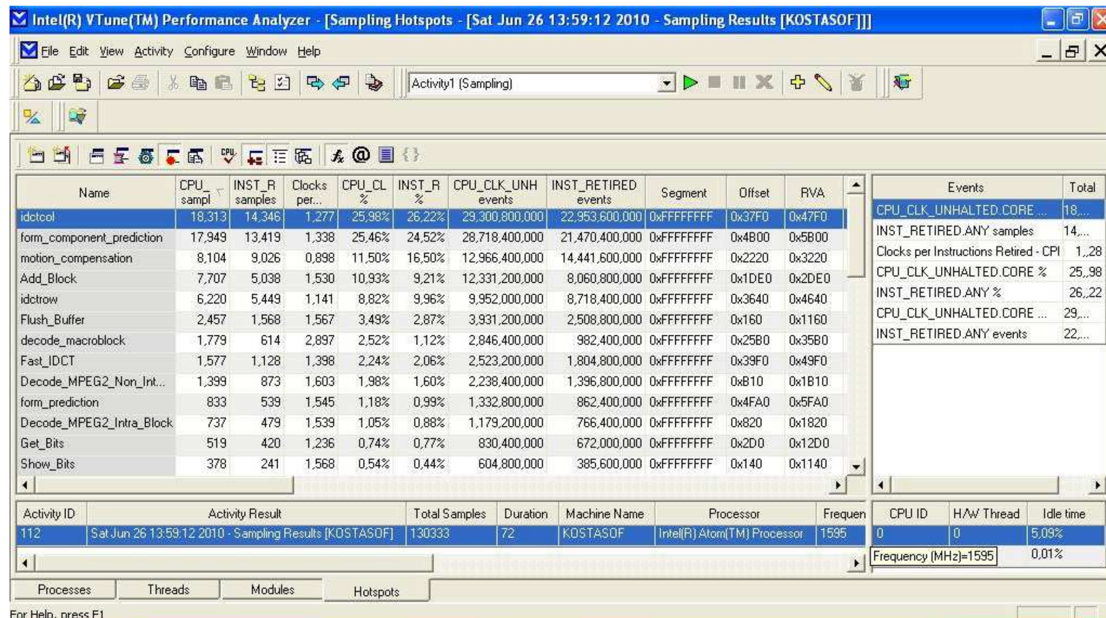


Fig 3.1 Function percentage

3.1.3 Input_Inter.m2v

Παρακάτω παρουσιάζεται ένα screenshot από το Vtune κατά τη διάρκεια εκτέλεσης του reference code MPEG-2 με σημείο αναφοράς το video Input_Inter.m2v.



Name	CPU_ sampl	INST_R samples	Clocks per...	CPU_CL %	INST_R %	CPU_CLK_UNH events	INST_RETIRED events	Segment	Offset	RVA
idctcol	18,313	14,346	1,277	25,98%	26,22%	29,300,800,000	22,953,600,000	0xFFFFFFFF	0x37F0	0x47F0
form_component_prediction	17,949	13,419	1,338	25,46%	24,52%	28,718,400,000	21,470,400,000	0xFFFFFFFF	0x4B00	0x5B00
motion_compensation	8,104	9,026	0,898	11,50%	16,50%	12,966,400,000	14,441,600,000	0xFFFFFFFF	0x2220	0x3220
Add_Block	7,707	5,038	1,530	10,93%	9,21%	12,331,200,000	8,060,800,000	0xFFFFFFFF	0x1DE0	0x2DE0
idctrow	6,220	5,449	1,141	8,82%	9,96%	9,952,000,000	8,718,400,000	0xFFFFFFFF	0x3640	0x4640
Flush_Buffer	2,457	1,568	1,567	3,49%	2,87%	3,931,200,000	2,508,800,000	0xFFFFFFFF	0x160	0x1160
decode_macroblock	1,779	614	2,897	2,52%	1,12%	2,846,400,000	982,400,000	0xFFFFFFFF	0x25B0	0x35B0
Fast_IDCT	1,577	1,128	1,398	2,24%	2,06%	2,523,200,000	1,804,800,000	0xFFFFFFFF	0x39F0	0x49F0
Decode_MPEG2_Non_Int...	1,399	873	1,603	1,98%	1,60%	2,238,400,000	1,396,800,000	0xFFFFFFFF	0xB10	0x1B10
form_prediction	833	539	1,545	1,18%	0,99%	1,332,800,000	862,400,000	0xFFFFFFFF	0x4FA0	0x5FA0
Decode_MPEG2_Intra_Block	737	479	1,539	1,05%	0,88%	1,179,200,000	766,400,000	0xFFFFFFFF	0x820	0x1820
Get_Bits	519	420	1,236	0,74%	0,77%	830,400,000	672,000,000	0xFFFFFFFF	0x2D0	0x12D0
Show_Bits	378	241	1,568	0,54%	0,44%	604,800,000	385,600,000	0xFFFFFFFF	0x140	0x1140

Events	Total
CPU_CLK_UNHALTED.CORE ...	18,...
INST_RETIRED.ANY samples	14,...
Clocks per Instructions Retired - CPI	1,,28
CPU_CLK_UNHALTED.CORE %	25,,98
INST_RETIRED.ANY %	26,,22
CPU_CLK_UNHALTED.CORE ...	29,...
INST_RETIRED.ANY events	22,...

Activity ID	Activity Result	Total Samples	Duration	Machine Name	Processor	Frequen
112	Sat Jun 26 13:59:12 2010 - Sampling Results [KOSTASOF]	130333	72	KOSTASOF	Intel(R) Atom(TM) Processor	1595

CPU ID	H/W Thread	Idle time
0	0	5,09%
Frequency (MHz)=1595		0,01%

Image 3.2 Vtune screenshot while executing Input_Inter.m2v

Παρακάτω παρουσιάζεται το διάγραμμα των συναρτήσεων που χρησιμοποιούνται κατά τη διάρκεια αποκωδικοποίησης του video Input_Inter.m2v.

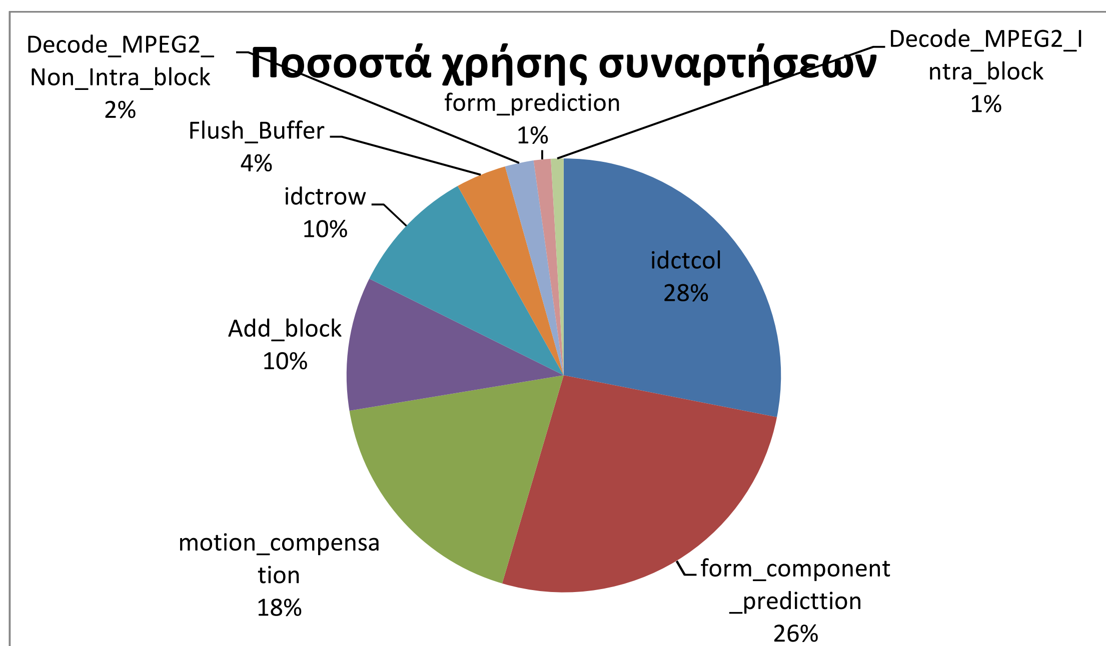


Fig 3.1 Function percentage

3.1.4 Συμπεράσματα αρχικών μετρήσεων

Από τις παραπάνω μετρήσεις εξάγεται το συμπέρασμα ότι το 96% του κόστους αποκωδικοποίησης του αλγορίθμου MPEG-2, συγκεντρώνεται σε 6 συναρτήσεις, τις idctcol, form_component_prediction, motion_compensation, add_block, idct_row και flush_buffer και γίνεται εύκολα αντιληπτό ότι αν βελτιώσουμε ως προς την ταχύτητα εκτέλεσης αυτές τις συναρτήσεις θα έχουμε και ταυτόχρονη μείωση του συνολικού κόστους αποκωδικοποίησης, περίπου κατά όσο έχει μειωθεί η βελτιωμένη συνάρτηση.

3.2. Στρατηγική για τη μείωση του κόστους

Η στρατηγική για μείωση του χρόνου αποκωδικοποίησης χωρίζεται σε δύο βασικά τμήματα: τη χρήση των δυνατοτήτων του επεξεργαστή που χρησιμοποιείται και στη συγκεκριμένη περίπτωση των intrinsic, όπως MMX, SSE, SSE2 ,SSE3 και SSSE3, αλλά και στην αλλαγή του reference code με μια πιο έξυπνη χρήση της C. Στη δεύτερη κατηγορία καλείται ο προγραμματιστής να αλλάξει μια υλοποίηση με μια άλλη η οποία είναι εξυπνότερη και γλιτώνει πράξης, όπως αποφυγή κάποιων if branch που μπορεί να κοστίσουν πολλούς κύκλους μηχανής, ή ακόμα και αλλαγή στον τρόπο που καλούνται οι συναρτήσεις και χρήση κάποιων trick για early termination χωρίς να χρειάζεται να εκτελεστεί μια συνάρτηση ολόκληρη. Σε αυτή τη διπλωματική εργασία χρησιμοποιούνται και οι δύο αυτοί τρόποι και αναφέρονται αναλυτικά παρακάτω, έχοντας ο καθένας το μερίδιό του στην τελική απόδοση του βελτιωμένου MPEG-2.

3.3. Επιμέρους συναρτήσεις

Όπως είδαμε στο κεφάλαιο 3.1 οι συναρτήσεις οι οποίες κοστίζουν επεξεργαστική ισχύ στον αποκωδικοποιητή MPEG-2 δεν είναι πολλές, παρά λίγες και συγκεκριμένες. Είναι λογικό η προσπάθεια για μείωση της απαίτησης σε ισχύ επεξεργαστή να επικεντρωθεί σε αυτές τις συναρτήσεις.

Παρατηρούμε ότι η συνάρτηση idct_row και idct_col καταλαμβάνουν το χ% του χρόνου επεξεργασίας.

3.3.1. IDCT

Όπως έχουμε δει και από τα στατιστικά δεδομένα η διαδικασία του IDCT στον αλγόριθμο MPEG-2 είναι η πλέον δαπανηρή διαδικασία κατά την αποκωδικοποίηση. Η συνάρτηση idctrow() εφαρμόζει τον IDCT σε κάθε σειρά του block, ενώ η συνάρτηση idctcol() την ακολουθεί και εφαρμόζει τον IDCT κατά στήλη. Οι δύο αυτές συναρτήσεις είναι παρόμοιες μεταξύ τους με τη διαφορά που αναφέρεται παραπάνω. Η idctrow() διαβάζει το block και κατόπιν εφαρμόζει την FAST IDCT κατά γραμμές και αποθηκεύει τα αποτελέσματα. Κατόπιν η idctcol() διαβάζει το νέο block από τη μνήμη και εφαρμόζει αντίστοιχα την FAST IDCT κατά στήλες. Τα δεδομένα που φορτώνονται για κάθε block έχουνε μέγεθος 16 bit ανά pixel με αποτέλεσμα να χωράει μια ολόκληρη γραμμή ενός block σε ένα καταχωρητή SSE που έχει μέγεθος 128 bit (μία γραμμή ενός block έχει 8 στοιχεία). Με αυτή τη μέθοδο μπορούμε να χρησιμοποιήσουμε τη δυνατότητα για SSE προγραμματισμό με επιτάχυνση του αρχικού κώδικα. Θα πρέπει τον κώδικα που είναι γραμμένος σε C code να τον μετατρέψουμε intrinsic code.

Παρουσιάζεται ο αρχικός κώδικας σε C

Φορτώνονται τα pixel μιας γραμμής

```
static void idctrow(blk)
short *blk;
{
  int x0, x1, x2, x3, x4, x5, x6, x7, x8;

  /* shortcut */
  if (!((x1 = blk[4]<<11) | (x2 = blk[6]) | (x3 = blk[2]) |
        (x4 = blk[1]) | (x5 = blk[7]) | (x6 = blk[5]) | (x7 = blk[3])))
  {
    blk[0]=blk[1]=blk[2]=blk[3]=blk[4]=blk[5]=blk[6]=blk[7]=blk[0]<<3;
    return;
  }

  x0 = (blk[0]<<11) + 128; /* for proper rounding in the fourth stage */

  /* first stage */
  x8 = W7*(x4+x5);
  x4 = x8 + (W1-W7)*x4;
  x5 = x8 - (W1+W7)*x5;
  x8 = W3*(x6+x7);
  x6 = x8 - (W3-W5)*x6;
  x7 = x8 - (W3+W5)*x7;

  /* second stage */
  x8 = x0 + x1;
  x0 -= x1;
  x1 = W6*(x3+x2);
  x2 = x1 - (W2+W6)*x2;
  x3 = x1 + (W2-W6)*x3;
  x1 = x4 + x6;
  x4 -= x6;
  x6 = x5 + x7;
  x5 -= x7;

  /* third stage */
  x7 = x8 + x3;
  x8 -= x3;
  x3 = x0 + x2;
  x0 -= x2;
  x2 = (181*(x4+x5)+128)>>8;
```

```
  x4 = (181*(x4-x5)+128)>>8;

  /* fourth stage */
  blk[0] = (x7+x1)>>8;
  blk[1] = (x3+x2)>>8;
  blk[2] = (x0+x4)>>8;
  blk[3] = (x8+x6)>>8;
  blk[4] = (x8-x6)>>8;
  blk[5] = (x0-x4)>>8;
  blk[6] = (x3-x2)>>8;
  blk[7] = (x7-x1)>>8;
}
```

3.3.1.1 Inverse Transform

Όπως φαίνεται και από τον reference code οι πράξεις που εφαρμόζονται στην idctrow αφορούνε τα pixel μιας γραμμής δηλαδή γειτονικά pixel κατά γραμμή. Όταν φορτώνω τα byte από τη μνήμη σε μια μεταβλητή XMM τότε αυτόματα φορτώνονται 8 τιμές γειτονικών pixel.

Ένα block έχει την παρακάτω μορφή:

Blk[0]	*Blk[1]*	*Blk[2]*	*Blk[3]*	*Blk[4]*	*Blk[5]*	*Blk[6]*	*Blk[7]*
Blk[8]	*Blk[9]*	*Blk[10]*	*Blk[11]*	*Blk[12]*	*Blk[13]*	*Blk[14]*	*Blk[15]*
Blk[16]	*Blk[17]*	*Blk[18]*	*Blk[19]*	*Blk[20]*	*Blk[21]*	*Blk[22]*	*Blk[23]*
Blk[24]	*Blk[25]*	*Blk[26]*	*Blk[27]*	*Blk[28]*	*Blk[29]*	*Blk[30]*	*Blk[31]*
Blk[32]	*Blk[33]*	*Blk[34]*	*Blk[35]*	*Blk[36]*	*Blk[37]*	*Blk[38]*	*Blk[39]*
Blk[40]	*Blk[41]*	*Blk[42]*	*Blk[43]*	*Blk[44]*	*Blk[45]*	*Blk[46]*	*Blk[47]*
Blk48]	*Blk[49]*	*Blk[50]*	*Blk[51]*	*Blk[52]*	*Blk[53]*	*Blk[54]*	*Blk[55]*
Blk[56]	*Blk[57]*	*Blk[58]*	*Blk[59]*	*Blk[60]*	*Blk[61]*	*Blk[62]*	*Blk[63]*

Table 3.2 Struct of a block

Φορτώνοντας από τη μνήμη τις τιμές ενός block σε καταχωρητές XMM θα έχω οχτώ καταχωρητές, όπου κάθε καταχωρητής θα περιέχει μία ολόκληρη γραμμή από το block. Ωστόσο οι πράξεις από ότι παρατηρώ είναι ανά στήλες, αφού ο FAST IDCT εφαρμόζεται στα pixel της γραμμής. Για να γίνει αυτό θα πρέπει να αντιστραφούνε οι στήλες και να γίνουνε γραμμές. Μια υλοποίηση αυτού φαίνεται παρακάτω:

```
// Transpose 8x8 matrix

      mm0 = _mm_loadu_si128((__m128i*)(blk));     //0 1 2 3 4 5 6 7
      mm1 = _mm_loadu_si128((__m128i*)(blk+8));   //0 1 2 3 4 5 6 7
      mm2 = _mm_loadu_si128((__m128i*)(blk+16));  //0 1 2 3 4 5 6 7
```

```
mm3 = _mm_loadu_si128((__m128i*)(blk+24)); //0 1 2 3 4 5 6 7
mm4 = _mm_loadu_si128((__m128i*)(blk+32)); //0 1 2 3 4 5 6 7
mm5 = _mm_loadu_si128((__m128i*)(blk+40)); //0 1 2 3 4 5 6 7
mm6 = _mm_loadu_si128((__m128i*)(blk+48)); //0 1 2 3 4 5 6 7
mm7 = _mm_loadu_si128((__m128i*)(blk+56)); //0 1 2 3 4 5 6 7

a0 = _mm_unpacklo_epi16(mm0,mm1);// 0 0 1 1 2 2 3 3
a1 = _mm_unpacklo_epi16(mm2,mm3);// 0 0 1 1 2 2 3 3
a2 = _mm_unpacklo_epi16(mm4,mm5);// 0 0 1 1 2 2 3 3
a3 = _mm_unpacklo_epi16(mm6,mm7);// 0 0 1 1 2 2 3 3

a4 = _mm_unpackhi_epi16(mm0,mm1);// 4 4 5 5 6 6 7 7
a5 = _mm_unpackhi_epi16(mm2,mm3);// 4 4 5 5 6 6 7 7
a6 = _mm_unpackhi_epi16(mm4,mm5);// 4 4 5 5 6 6 7 7
a7 = _mm_unpackhi_epi16(mm6,mm7);// 4 4 5 5 6 6 7 7

b0  = _mm_unpacklo_epi32(a0,a1);// 0 0 0 0 1 1 1 1
b1  = _mm_unpacklo_epi32(a2,a3);// 0 0 0 0 1 1 1 1
mm0 = _mm_unpacklo_epi64(b0,b1);// 0 0 0 0 0 0 0 0
mm1 = _mm_unpackhi_epi64(b0,b1);// 1 1 1 1 1 1 1 1

b0  = _mm_unpackhi_epi32(a0,a1);// 2 2 2 2 3 3 3 3
b1  = _mm_unpackhi_epi32(a2,a3);// 2 2 2 2 3 3 3 3
mm2 = _mm_unpacklo_epi64(b0,b1);// 2 2 2 2 2 2 2 2
mm3 = _mm_unpackhi_epi64(b0,b1);// 3 3 3 3 3 3 3 3

b0  = _mm_unpacklo_epi32(a4,a5);// 4 4 4 4 5 5 5 5
b1  = _mm_unpacklo_epi32(a6,a7);// 4 4 4 4 5 5 5 5
mm4 = _mm_unpacklo_epi64(b0,b1);// 4 4 4 4 4 4 4 4
mm5 = _mm_unpackhi_epi64(b0,b1);// 5 5 5 5 5 5 5 5

b0  = _mm_unpackhi_epi32(a4,a5);// 6 6 6 6 7 7 7 7
b1  = _mm_unpackhi_epi32(a6,a7);// 6 6 6 6 7 7 7 7
mm6 = _mm_unpacklo_epi64(b0,b1);// 6 6 6 6 6 6 6 6
mm7 = _mm_unpackhi_epi64(b0,b1);// 7 7 7 7 7 7 7 7
```

Πλέον οι καταχωρητές mm0-mm7 περιέχουνε τα pixels ανά στήλη, όπως φαίνεται και στο παρακάτω σχήμα. Ωστόσο, η παραπάνω υλοποίηση κοστίζει σε πράξεις (24 για κάθε block). Μήπως υπάρχει καλύτερη λύση;

Blk[0]	*Blk[8]*	*Blk[16]*	*Blk[24]*	*Blk[32]*	*Blk[40]*	*Blk[48]*	*Blk[56]*
Blk[1]	*Blk[9]*	*Blk[17]*	*Blk[25]*	*Blk[33]*	*Blk[41]*	*Blk[49]*	*Blk[57]*
Blk[2]	*Blk[10]*	*Blk[18]*	*Blk[26]*	*Blk[34]*	*Blk[42]*	*Blk[50]*	*Blk[58]*
Blk[3]	*Blk[11]*	*Blk[19]*	*Blk[27]*	*Blk[35]*	*Blk[43]*	*Blk[51]*	*Blk[59]*
Blk[4]	*Blk[12]*	*Blk[20]*	*Blk[28]*	*Blk[36]*	*Blk[44]*	*Blk[52]*	*Blk[60]*
Blk[5]	*Blk[13]*	*Blk[21]*	*Blk[29]*	*Blk[37]*	*Blk[45]*	*Blk[53]*	*Blk[61]*
Blk[6]	*Blk[14]*	*Blk[22]*	*Blk[30]*	*Blk[38]*	*Blk[46]*	*Blk[54]*	*Blk[62]*
Blk[7]	*Blk[15]*	*Blk[23]*	*Blk[31]*	*Blk[39]*	*Blk[47]*	*Blk[55]*	*Blk[63]*

Table 3.3 Struct of an inverse block

Η απάντηση είναι θετική. Ας φανταστούμε τα δεδομένα να διαβάζονται ήδη ανεστραμμένα, δηλαδή στη μνήμη αντί να αποθηκεύονται τα δεδομένα με ZIG-ZAG σειρά, να αποθηκεύονται με μία παραλλαγή του πίνακα ZIG-ZAG, τέτοια ώστε οι γραμμές του ZIG-ZAG να είναι στήλες του νέου πίνακα. Αυτό γίνεται κατά τη διάρκεια της συνάρτησης `Decode_MPEG2_Intra_Block()` η τροποποίηση της οποίας φαίνεται παρακάτω:

Reference code

```
j = scan[ld1->alternate_scan][i];

val = (val * ld1->quantizer_scale *
qmat[j]) >> 4;

bp[j] = sign ? -val : val;
```

Τροποποιημένος κώδικας

```
j = scan[ld1->alternate_scan][i];

val = (val * ld1->quantizer_scale *
qmat[j]) >> 4;

//transpose by using another table
j = scan2[ld1->alternate_scan][i];
bp[j] = sign ? -val : val;
```

Παρακάτω φαίνονται οι δύο πίνακες ZIG-ZAG, ο κανονικός και η παραλλαγή του για να αποθηκεύονται οι γραμμές στις στήλες.

```
/* zig-zag and alternate scan
patterns */
EXTERN unsigned char scan[2][64]
#ifdef GLOBAL
=
{
     { // Zig-Zag scan pattern
       0,1,8,16,9,2,3,10,

       17,24,32,25,18,11,4,5,

       12,19,26,33,40,48,41,34,

       27,20,13,6,7,14,21,28,

       35,42,49,56,57,50,43,36,

       29,22,15,23,30,37,44,51,

       58,59,52,45,38,31,39,46,

       53,60,61,54,47,55,62,63
       },


       { // Alternate scan pattern

       0,8,16,24,1,9,2,10,

       17,25,32,40,48,56,57,49,

       41,33,26,18,3,11,4,12,

       19,27,34,42,50,58,35,43,

       51,59,20,28,5,13,6,14,

       21,29,36,44,52,60,37,45,

       53,61,22,30,7,15,23,31,

       38,46,54,62,39,47,55,63
       }

}
```

```
/* zig-zag and alternate scan
patterns */
EXTERN unsigned char scan2[2][64]
#ifdef GLOBAL
=
{
      {//new zig-zag scan pattern
       0,8,1,2,9,16,24,17,

       10,3,4,11,18,25,32,40,

       33,26,19,12,5,6,13,20,

       27,34,41,48,56,49,42,35,

       28,21,14,7,15,22,29,36,

       43,50,57,58,51,44,37,30,

       23,31,38,45,52,59,60,53,

       46,39,47,54,61,62,55,63,
       },

      {//new alternate scan
      pattern
       0,1,2,3,8,9,16,17,

       10,11,4,5,6,7,15,14,

       13,12,19,18,24,25,32,33,

       26,27,20,21,22,23,28,29,

       30,31,34,35,40,41,48,49,

       42,43,36,37,38,39,44,45,

       46,47,50,51,56,57,58,59,

       52,53,54,55,60,61,62,63
       }

}
```

3.3.1.2 idctrow-idctcol

Ας δούμε πιο αναλυτικά τη μετατροπή του reference code σε intrinsic code για τη συνάρτηση idctrow ενώ παρόμοια θα είναι και η μετατροπή για την idctcol χωρίς ωστόσο το invert transpose στο τέλος.

<u>Συνάρτηση idctrow()</u>

Φόρτωμα των τιμών το pixel σε καταχωρητές xmm.

```
__m128i mm5,mm15,mm3,mm13,mm1,mm7,mm11,mm17,mm18,mm8;
__m128i mm0,mm10,mm4,mm14,mm6,mm16,mm2,mm12;
__m128i a0,a1,a2,a3,a4,a5,a6,a7,b0,tmp1;

//i have done the transpose by the stage of quantization

        //loading the block 8x8
        mm0 = _mm_load_si128((__m128i*)(blk));          //0 1 2 3 4 5 6 7
        mm1 = _mm_load_si128((__m128i*)(blk+8));        //0 1 2 3 4 5 6 7
        mm2 = _mm_load_si128((__m128i*)(blk+16));       //0 1 2 3 4 5 6 7
        mm3 = _mm_load_si128((__m128i*)(blk+24));       //0 1 2 3 4 5 6 7
        mm4 = _mm_load_si128((__m128i*)(blk+32));       //0 1 2 3 4 5 6 7
        mm5 = _mm_load_si128((__m128i*)(blk+40));       //0 1 2 3 4 5 6 7
        mm6 = _mm_load_si128((__m128i*)(blk+48));       //0 1 2 3 4 5 6 7
        mm7 = _mm_load_si128((__m128i*)(blk+56));       //0 1 2 3 4 5 6 7

        tmp1 = _mm_load_si128((__m128i*)(constants_32+0*4)); //1 1 1 1
        a3 = _mm_load_si128((__m128i*)(constants_32+2*4));   //128 128 128 128
```

Θα πρέπει για να γίνουνε οι πράξεις και να μην έχω over float να μετατρέψω τη X0 σε 32 bit, χρησιμοποιώντας 2 καταχωρητές (mm0,mm10) ,ένα για low significant bit και ένα για high significant bit. (βλ. παρατηρήσεις 2)

Αριστερή ολίσθηση κατά 11 και κατόπιν πρόσθεση με 128, όπως προστάζει ο reference code.

```
        //load x0 and convert to 32 bit
        mm10 = _mm_unpacklo_epi16 (mm0,mm0);    //mm10 has the low bits
        mm10 = _mm_madd_epi16 (mm10, tmp1);
        mm10 = _mm_slli_epi32 (mm10,11);
        mm10 = _mm_add_epi32 (mm10,a3);         //x0 = (blk[8*0]<<11) + 128;
        mm0 = _mm_unpackhi_epi16 (mm0,mm0);     //mm0 has the high bits
        mm0 = _mm_madd_epi16 (mm0, tmp1);
        mm0 = _mm_slli_epi32 (mm0, 11);
        mm0 = _mm_add_epi32 (mm0,a3);           //x0 = (blk[8*0]<<11) + 128
```

Εδώ ξεκινά το 1° στάδιο του FAST IDCT , και ακολουθούν οι πράξεις οι οποίες μετατρέπονται από c code σε intrinsic code.

```
        /*********************************
        *         first stage           *
        *                               *
        *********************************/

//loading constants
a4 = _mm_load_si128((__m128i*)(constants+12*8)); //a4={W7,W7,W7,W7,W7,W7,W7,W7}

   mm11 = _mm_unpacklo_epi16 (mm1, mm7);      //mm11 has the low bits
   mm18 = _mm_madd_epi16 (mm11, a4);          //W7*(x4+x5)
   mm11 = _mm_unpackhi_epi16 (mm1, mm7);      //mm11 has the high bits
   mm8 = _mm_madd_epi16 (mm11, a4);           //W7*(x4+x5)


   //load constants
   a2 = _mm_load_si128((__m128i*)(constants+2*8)); //a2 = {w1,…w1} 8 times
   a4 = _mm_load_si128((__m128i*)(constants+13*8));//a4 = {-w7,…-w7} 8 times

(W1-W7)*x4);     //x4 = (x8+(W1-W7)*x4);
   mm15 = _mm_unpacklo_epi16 (a4, a2);        //mm15 has the low bits
   mm11 = _mm_unpacklo_epi16 (mm1,mm1);       //mm11 has the low bits
```

Σχηματισμός της πράξης

x4 = (x8+(W1-W7)*x4)

```
mm11 = _mm_madd_epi16 (mm11, mm15);
mm11 = _mm_add_epi32 (mm11, mm18);

mm15 = _mm_unpackhi_epi16 (a4, a2);        //mm15 has the high bits
mm1 = _mm_unpackhi_epi16 (mm1,mm1);        //mm1 has the high bits
mm1 = _mm_madd_epi16 (mm1, mm15);
mm1 = _mm_add_epi32 (mm1, mm8);


//load constants
a2 = _mm_load_si128((__m128i*)(constants+3*8));   //a2 = {-w1,…-w1}
a3 = _mm_load_si128((__m128i*)(constants+6*8));   //a3 = {w3,…w3}
```

Σχηματισμός της πράξης

x5 = (x8-(W1+W7)*x5)

```
//x5 = (x8-(W1+W7)*x5);
mm15 = _mm_unpacklo_epi16 (a4, a2);//mm15 has the low bits (-w7,-w1)
mm17 = _mm_unpacklo_epi16 (mm7,mm7);
mm17 = _mm_madd_epi16 (mm17, mm15);
mm17 = _mm_add_epi32 (mm17, mm18);

a5 = _mm_unpackhi_epi16 (a4, a2);  //a5 has the high bits of (-w7,-w1)
mm7 = _mm_unpackhi_epi16 (mm7,mm7);
mm7 = _mm_madd_epi16 (mm7, mm15);
mm7 = _mm_add_epi32 (mm7, mm8);
```

Σχηματισμός της πράξης

x8 = W3*(x6+x7)

```
//  x8 = W3*(x6+x7);
a0 = _mm_unpacklo_epi16 (mm5, mm3);        //a0 has the low bits
mm18 = _mm_madd_epi16 (a0, a3);

a0 = _mm_unpackhi_epi16 (mm5, mm3);        //a0 has the higher bits
mm8 = _mm_madd_epi16 (a0, a3);


//load constants
//a3 has -w3 and a2 has w5
a4 = _mm_load_si128((__m128i*)(constants+7*8));
a2 = _mm_load_si128((__m128i*)(constants+8*8));
```

Σχηματισμός της πράξης

x6 = (x8-(W3-W5)*x6)

```
//x6 = (x8-(W3-W5)*x6);
a0 = _mm_unpacklo_epi16 (a4, a2);          //a0 has the low bits
mm15 = _mm_unpacklo_epi16 (mm5,mm5);
mm15 = _mm_madd_epi16 (mm15, a0);
mm15 = _mm_add_epi32 (mm15, mm18);         //mm15 = (x8-(W3-W5)*x6)

a0 = _mm_unpackhi_epi16 (a4, a2);          //a0 has the high significant bits
mm5 = _mm_unpackhi_epi16 (mm5,mm5);
mm5 = _mm_madd_epi16 (mm5, a0);
mm5 = _mm_add_epi32 (mm5, mm8);             //mm5 = (x8-(W3-W5)*x6)


//load constants
a2 = _mm_load_si128((__m128i*)(constants+9*8));
```

Σχηματισμός της πράξης

x7 = (x8-(W3+W5)*x7)

```
//x7 = (x8-(W3+W5)*x7);
a0 = _mm_unpacklo_epi16 (a4, a2);          //a0 has the low bits
a1 = _mm_unpacklo_epi16 (mm3,mm3);
 a0 = _mm_madd_epi16 (a0, a1);
mm13 = _mm_add_epi32 (a0, mm18);           //mm13 = (x8-(W3-W5)*x6)

a0 = _mm_unpackhi_epi16 (a4, a2);          //a0 has the high bits
a1 = _mm_unpackhi_epi16 (mm3,mm3);
 a0 = _mm_madd_epi16 (a0, a1);
mm3 = _mm_add_epi32 (a0, mm8);             //mm3 = (x8-(W3-W5)*x6)
```

Εδώ ξεκινά το 2ο στάδιο του FAST IDCT , και ακολουθούν οι πράξεις οι οποίες μετατρέπονται από c code σε intrinsic code.

```
/********************************
*          second stage         *
*                               *
********************************/


//load constants
a3 = _mm_load_si128((__m128i*)(constants+10*8));//a3 = {w6,…w6}


//x1 = (blk[4]<<11)
//convert  to 32 bit
mm14 = _mm_unpacklo_epi16 (mm4,mm4);   // mm14 has the low bits
mm14 = _mm_madd_epi16 (mm14, tmp1);
mm14 = _mm_slli_epi32 (mm14, 11);      //x1 = (blk[4]<<11)

mm4 = _mm_unpackhi_epi16 (mm4,mm4);    // mm14 has the high bits
mm4 = _mm_madd_epi16 (mm4, tmp1);
mm4 = _mm_slli_epi32 (mm4, 11);        //x1 = (blk[4]<<11)
```

Σχηματισμός της πράξης

x8 = x0 + x1

```
//x8 = x0 + x1
mm18 = _mm_add_epi32 (mm10,mm14);
mm8 = _mm_add_epi32 (mm0,mm4);
```

Σχηματισμός της πράξης

x0 -= x1

```
// x0 -= x1
mm10 = _mm_sub_epi32 (mm10,mm14);
mm0 = _mm_sub_epi32 (mm0,mm4);
```

Σχηματισμός της πράξης

x1 = W6*(x3+x2)

```
//x1 = W6*(x3+x2);
a0 = _mm_unpacklo_epi16 (mm6, mm2);   // a0 has the low bits
mm14 = _mm_madd_epi16 (a0, a3);       //x1 = W6*(x3+x2);

a0 = _mm_unpackhi_epi16 (mm6, mm2);   //a0 has the high bits
mm4 = _mm_madd_epi16 (a0, a3);        //x1 = W6*(x3+x2);



//load constants
a3 = _mm_load_si128((__m128i*)(constants+11*8));     //a3 = {-w6,…-w6}
a2 = _mm_load_si128((__m128i*)(constants+5*8));      //a2 = {-w2,…-w2}
```

Σχηματισμός της πράξης

x2 = (x1-(W2+W6)*x2)

```
//x2 = (x1-(W2+W6)*x2)
a0 = _mm_unpacklo_epi16 (a3, a2);     // a0 has the low bits
a1 = _mm_unpacklo_epi16 (mm6,mm6);
a0 = _mm_madd_epi16 (a0, a1);
mm16 = _mm_add_epi32 (a0, mm14);      //x2 = (x1-(W2+W6)*x2)

a0 = _mm_unpackhi_epi16 (a3, a2);     // a0 has the high bits
a1 = _mm_unpackhi_epi16 (mm6,mm6);
a0 = _mm_madd_epi16 (a0, a1);
mm6 = _mm_add_epi32 (a0, mm4);        //x2 = (x1-(W2+W6)*x2)



//load constants
a2 = _mm_load_si128((__m128i*)(constants+4*8)); //a2 = {w2,…w2}
```

Σχηματισμός της πράξης

x3 = (x1+(W2-W6)*x3)

```
// x3 = (x1+(W2-W6)*x3);
a0 = _mm_unpacklo_epi16 (a3, a2);     // a0 has the low bits
a1 = _mm_unpacklo_epi16 (mm2,mm2);
a0 = _mm_madd_epi16 (a0, a1);
mm12 = _mm_add_epi32 (a0, mm14);      // x3 = (x1+(W2-W6)*x3);

a0 = _mm_unpackhi_epi16 (a3, a2);     // a0 has the high bits
a1 = _mm_unpackhi_epi16 (mm2,mm2);
```

```
a0 = _mm_madd_epi16 (a0, a1);
mm2 = _mm_add_epi32 (a0, mm4);        // x3 = (x1+(W2-W6)*x3);
```

Σχηματισμός της πράξης

Σχηματισμός της πράξης

x4 -= x6

```
// x1 = x4 + x6;
mm14 = _mm_add_epi32 (mm11,mm15);
mm4 = _mm_add_epi32 (mm1,mm5);

//x4 -= x6;
mm11 = _mm_sub_epi32 (mm11,mm15);
mm1 = _mm_sub_epi32 (mm1,mm5);
```

Σχηματισμός της πράξης

x6 = x5 + x7

```
//x6 = x5 + x7;
mm15 = _mm_add_epi32 (mm17,mm13);
mm5 = _mm_add_epi32 (mm7,mm3);
```

Σχηματισμός της πράξης

x5 -= x7

```
// x5 -= x7;
mm17 = _mm_sub_epi32 (mm17,mm13);
mm7 = _mm_sub_epi32 (mm7,mm3);


/********************************
*           third stage         *
*                               *
********************************/
```

Σχηματισμός της πράξης

x7 = x8 + x3

```
// x7 = x8 + x3;
mm13 = _mm_add_epi32 (mm18,mm12);
mm3 = _mm_add_epi32 (mm8,mm2);
```

Σχηματισμός της πράξης

x8 -= x3

```
//x8 -= x3;
mm18 = _mm_sub_epi32 (mm18,mm12);
mm8 = _mm_sub_epi32 (mm8,mm2);
```

Σχηματισμός της πράξης

x3 = x0 + x2

```
//x3 = x0 + x2;
mm12 = _mm_add_epi32 (mm10,mm16);
mm2 = _mm_add_epi32 (mm0,mm6);
```

Σχηματισμός της πράξης

x3 = x0 + x2

```
//  x0 -= x2;
mm10 = _mm_sub_epi32 (mm10,mm16);
mm0 = _mm_sub_epi32 (mm0,mm6);

//load constants
a3 = _mm_load_si128((__m128i*)(constants_32+3*4));   //a3 = {181,…181}
a1 = _mm_load_si128((__m128i*)(constants_32+2*4));   //a1 = {128,…128}
```

Σχηματισμός της πράξης

x2=(181*(x4+x5)+128)>>8

παρατηρώ ότι η πράξη (x4+x5) είναι ένας αριθμός μεγαλύτερος από 16 bit. Στα intrinsics του SSE δεν υπάρχει πολλλαπλασιασμός 32 bit, έτσι εφαρμόζεται ο αλγόριθμος του Karatsuba η λειτουργία του οποίου εξηγείται αναλυτικά παρακάτω στις παρατηρήσεις(3).

```
//  x2 = (181*(x4+x5)+128)>>8;
a2 = _mm_add_epi32(mm11,mm17);        //x4+x5
a4 = _mm_srai_epi32(a2,16);           //Keep High 16-bits (signed)
a4 = _mm_madd_epi16(a4,a3);
a0 = _mm_mulhi_epu16(a2,a3);          //high-16 results of Low*181
a2 = _mm_mullo_epi16(a2,a3);          //low-16 results of Low*181
a0 = _mm_slli_epi32(a0,16);
a2 = _mm_or_si128(a2,a0);
a4 = _mm_slli_epi32(a4,16);
a4 = _mm_add_epi32(a2,a4);
a4 = _mm_add_epi32(a4,a1);
mm16 = _mm_srai_epi32(a4,8);


a2 = _mm_add_epi32(mm1,mm7);          //x4+x5
a4 = _mm_srai_epi32(a2,16);           //Keep High 16-bits (signed)
```

Πολλαπλασιασμός με τα 16 περισσότερο σημαντικά bit (βήμα 1)

Πολλαπλασιασμός με τα 16 λιγότερο σημαντικά bit (βήμα 2)

Άθροισμα των δύο αποτελεσμάτων για το τελικό αποτέλεσμα του πολλαπλασιασμού (βήμα 3)

```
a4 = _mm_madd_epi16(a4,a3);
a0 = _mm_mulhi_epu16(a2,a3);         //high-16 results of Low*181
a2 = _mm_mullo_epi16(a2,a3);         //low-16 results of Low*181
a0 = _mm_slli_epi32(a0,16);
a2 = _mm_or_si128(a2,a0);
a4 = _mm_slli_epi32(a4,16);
a4 = _mm_add_epi32(a2,a4);
a4 = _mm_add_epi32(a4,a1);
mm6 = _mm_srai_epi32(a4,8);



//x4 = (181*(x4-x5)+128)>>8
a2 = _mm_sub_epi32(mm11,mm17);       //x4-x5
a4 = _mm_srai_epi32(a2,16);          //Keep High 16-bits (signed)
a4 = _mm_madd_epi16(a4,a3);
a0 = _mm_mulhi_epu16(a2,a3);         //high-16 results of Low*181
a2 = _mm_mullo_epi16(a2,a3);         //low-16 results of Low*181
a0 = _mm_slli_epi32(a0,16);
a2 = _mm_or_si128(a2,a0);
a4 = _mm_slli_epi32(a4,16);
a4 = _mm_add_epi32(a2,a4);
a4 = _mm_add_epi32(a4,a1);
mm11 = _mm_srai_epi32(a4,8);


a2 = _mm_sub_epi32(mm1,mm7);         //x4+x5
a4 = _mm_srai_epi32(a2,16);          //Keep High 16-bits (signed)
a4 = _mm_madd_epi16(a4,a3);
a0 = _mm_mulhi_epu16(a2,a3);         //high-16 results of Low*181
a2 = _mm_mullo_epi16(a2,a3);         //low-16 results of Low*181
a0 = _mm_slli_epi32(a0,16);
a2 = _mm_or_si128(a2,a0);
a4 = _mm_slli_epi32(a4,16);
a4 = _mm_add_epi32(a2,a4);
a4 = _mm_add_epi32(a4,a1);
mm1 = _mm_srai_epi32(a4,8);



/********************************
*         fourth stage          *
*                                *
********************************/



//blk[0] = (x7+x1)>>8;
b0 = _mm_add_epi32 (mm13, mm14);     //x7+x1 lo
b0 = _mm_srai_epi32(b0 ,8);
a0 = _mm_add_epi32 (mm3, mm4);       //x7+x1 hi
a0 = _mm_srai_epi32(a0 ,8);
a0 = _mm_packs_epi32 (b0 ,a0);       //a0 packs-saturation to short
```

Η πράξη _mm_packs_epi32 χρησιμοποιείται για να βάλει σε ένα καταχωρητή τα δεδομένα δύο καταχωρητών κάνοντας saturation όπου χρειάζεται. Περισσότερη ανάλυση στις παρατηρήσεις(4).

```
//blk[8*1] = (x3+x2)>>8;
b0 = _mm_add_epi32 (mm12, mm16);     //x3+x2 lo
b0 = _mm_srai_epi32(b0 ,8);
a1 = _mm_add_epi32 (mm2, mm6);       //x3+x2 hi
a1 = _mm_srai_epi32(a1 ,8);
a1 = _mm_packs_epi32 (b0 ,a1);       //a3 packs-saturation to short



//blk[8*2] = (x0+x4)>>8;
b0 = _mm_add_epi32 (mm10, mm11);     //x0+x4 lo
b0 = _mm_srai_epi32(b0 ,8);
a2 = _mm_add_epi32 (mm0, mm1);       //x0+x4 hi
a2 = _mm_srai_epi32(a2 ,8);
a2 = _mm_packs_epi32 (b0 ,a2);       //a3 packs-saturation to short
```

```
// blk[8*3] = (x8+x6)>>8;
b0 = _mm_add_epi32 (mm18, mm15);       //x8+x6 lo
b0 = _mm_srai_epi32(b0 ,8);
a3 = _mm_add_epi32 (mm8, mm5);         //x8+x6 hi
a3 = _mm_srai_epi32(a3 ,8);
a3 = _mm_packs_epi32 (b0 ,a3);         //a3 packs-saturation to short


//blk[8*4] =(x8-x6)>>8;
b0 = _mm_sub_epi32 (mm18, mm15);       //x8-x6 lo
b0 = _mm_srai_epi32(b0 ,8);
a4 = _mm_sub_epi32 (mm8, mm5);         //x8-x6 hi
a4 = _mm_srai_epi32(a4 ,8);
a4 = _mm_packs_epi32 (b0 ,a4);         //a4 packs-saturation to short



//  blk[8*5] = (x0-x4)>>8;
b0 = _mm_sub_epi32 (mm10, mm11);       //x0-x4 lo
b0 = _mm_srai_epi32(b0 ,8);
a5 = _mm_sub_epi32 (mm0, mm1);         //x0-x4 hi
a5 = _mm_srai_epi32(a5 ,8);
a5 = _mm_packs_epi32 (b0 ,a5);         //a5 packs-saturation to short



// blk[8*6] = (x3-x2)>>8;
b0 = _mm_sub_epi32 (mm12, mm16);       //x3-x2 lo
b0 = _mm_srai_epi32(b0 ,8);
a6 = _mm_sub_epi32 (mm2, mm6);
a6 = _mm_srai_epi32(a6 ,8);            //x3-x2 hi
a6 = _mm_packs_epi32 (b0 ,a6);         //a3 packs-saturation to short



//  blk[8*7] = (x7-x1)>>8
b0 = _mm_sub_epi32 (mm13, mm14);       //x7-x1 lo
b0 = _mm_srai_epi32(b0 ,8);
a7 = _mm_sub_epi32 (mm3, mm4);         //x7-x1 hi
a7 = _mm_srai_epi32(a7 ,8);
a7 = _mm_packs_epi32 (b0 ,a7);         //a7 packs-saturation to short


//invert transpose
mm0 = _mm_unpacklo_epi16(a0,a1);       //0 1 0 1 0 1 0 1
mm1 = _mm_unpackhi_epi16(a0,a1);       //0 1 0 1 0 1 0 1

mm2 = _mm_unpacklo_epi16(a2,a3);       //2 3 2 3 2 3 2 3
mm3 = _mm_unpackhi_epi16(a2,a3);       //2 3 2 3 2 3 2 3

mm4 = _mm_unpacklo_epi16(a4,a5);       //4 5 4 5 4 5 4 5
mm5 = _mm_unpackhi_epi16(a4,a5);       //4 5 4 5 4 5 4 5

mm6 = _mm_unpacklo_epi16(a6,a7);       //6 7 6 7 6 7 6 7
mm7 = _mm_unpackhi_epi16(a6,a7);       //6 7 6 7 6 7 6 7

a0 = _mm_unpacklo_epi32(mm0,mm2);      //0 1 2 3 0 1 2 3
a1 = _mm_unpackhi_epi32(mm0,mm2);      //0 1 2 3 0 1 2 3
a2 = _mm_unpacklo_epi32(mm1,mm3);      //0 1 2 3 0 1 2 3
a3 = _mm_unpackhi_epi32(mm1,mm3);      //0 1 2 3 0 1 2 3

a4 = _mm_unpacklo_epi32(mm4,mm6);      //4 5 6 7 4 5 6 7
a5 = _mm_unpackhi_epi32(mm4,mm6);      //4 5 6 7 4 5 6 7
a6 = _mm_unpacklo_epi32(mm5,mm7);      //4 5 6 7 4 5 6 7
a7 = _mm_unpackhi_epi32(mm5,mm7);      //4 5 6 7 4 5 6 7

mm0 = _mm_unpacklo_epi64(a0,a4);       //0 1 2 3 4 5 6 7
mm1 = _mm_unpackhi_epi64(a0,a4);       //0 1 2 3 4 5 6 7
mm2 = _mm_unpacklo_epi64(a1,a5);       //0 1 2 3 4 5 6 7
mm3 = _mm_unpackhi_epi64(a1,a5);       //0 1 2 3 4 5 6 7
mm4 = _mm_unpacklo_epi64(a2,a6);       //0 1 2 3 4 5 6 7
mm5 = _mm_unpackhi_epi64(a2,a6);       //0 1 2 3 4 5 6 7
```

```
mm6 = _mm_unpacklo_epi64(a3,a7);       //0 1 2 3 4 5 6 7
mm7 = _mm_unpackhi_epi64(a3,a7);       //0 1 2 3 4 5 6 7
```

Στον παραπάνω κώδικα(idctrow) δεν υπάρχει εσκεμμένα αποθήκευση των block μετά το πέρας του transpose. Γιατί;

Η συνάρτηση idctcol εκτελείται ως γνωστό αμέσως μετά το πέρας της συνάρτησης idctrow. Η idctcol φορτώνει το block το οποίο και έχει κάνει ήδη idct κατά γραμμές η συνάρτηση idctrow. Υπάρχει κανένα πρόβλημα αντί να κάνω αποθήκευση τα αποτελέσματα του idctrow στη μνήμη να τα κρατήσω σε τοπικές μεταβλητές και να αντικαταστήσω τη συνάρτηση idctrow και idctcol με μία συνάρτηση idct; Προφανώς και το αποτέλεσμα θα είναι το ίδιο γλιτώνοντας αρκετές πράξεις αποθήκευσης και φορτώματος των block ακόμα και αν τα δεδομένα βρίσκονται στην Cache και όχι στη μνήμη Ram, θα έχω και πάλι κέρδος.

Παρατηρήσεις

1. Όταν η μνήμη από όπου φορτώνω δεδομένα σε καταχωρητές XMM δεν είναι aligned 16, τότε το φόρτωμα γίνεται με την εντολή _mm_loadu_si128. Ωστόσο, η συνάρτηση load unsigned κοστίζει αρκετά παραπάνω χρόνο από τη συνάρτηση mm_load_si128 που χρησιμοποιείται όταν τα δεδομένα είναι 16 byte aligned, περίπου το διπλάσιο ίσως χρόνο ή και παραπάνω σύμφωνα με μετρήσεις που γίνανε με βάση το Vtune. Η λύση βρίσκεται στο να κάνω τα block από όπου φορτώνω τα δεδομένα 16 byte aligned και έτσι θα μπορώ να τα φορτώσω με τη συνάρτηση mm_load_si128. Πλέον κατά την αποθήκευση αντί της _mm_storeu_si128 χρησιμοποιώ την _mm_store_si128.

   Η αλλαγή που γίνεται είναι στο αρχείο global.h:

   __declspec (aling(16)) short block [12][64];

2. Σε αρκετές περιπτώσεις είναι απαραίτητη η μετατροπή ενός 16 bit αριθμού σε 32 bit ώστε κατά τις πράξεις να μην έχω over float και χάσω κάποια από τα most significant bits. Η τεχνική που χρησιμοποιώ εδώ είναι η εξής: έστω ένας XMM καταχωρητής mm0 όπως φαίνεται παρακάτω:

X7	X6	X5	X4	X3	X2	X1	X0

Κάνοντας unpacklo_epi16 και unpackhi_epi16 στον παραπάνω καταχωρητή θα έχω αντίστοιχα τους παρακάτω δύο καταχωρητές:

Unpacklo_epi16

X3	X2	X1	X0	X3	X2	X1	X0

Unpackhi_epi16

X7	X6	X5	X4	X7	X6	X5	X4

Πλέον κάνοντας madd_epi16 τους δύο παραπάνω καταχωρητές με τον καταχωρητή:

0	1	0	1	0	1	0	1

Θα έχω τους εξής δύο καταχωρητές:

X3	X2	X1	X0

Και,

X7	X6	X5	X4

Ο αρχικός αριθμός 16 bit έχει μετατραπεί σε 32 bit και οι 8 αριθμοί που βρίσκονταν στον καταχωρητή XMM πλέον τοποθετούνται σε δύο καταχωρητές XMM.

3. *Πολλαπλασιασμός με τον αλγόριθμο Karatsuba.* Ο αλγόριθμος του Karatsuba χρησιμοποιείται για πολλαπλασιασμό δύο πολύ μεγάλων αριθμών και μας δίνει τη δυνατότητα αντί για ένα πολλαπλασιασμό να εκτελέσουμε 2 πολλαπλασιασμούς και να έχουμε το ίδιο αποτέλεσμα. Στην περίπτωση μας το πρόβλημα που αντιμετωπίζουμε είναι ότι η Intel δεν έχει κάποια intrinsic πράξη που να εκτελεί πολλαπλασιασμό 32 bit αριθμών, παρά μόνο 16 bit. Έτσι η προσπάθεια που γίνεται είναι να αντικατασταθεί αυτός ο πολλαπλασιασμός με δύο πολλαπλασιασμούς 16 bit. Ας δούμε αναλυτικότερα τον αλγόριθμο: Έστω ένας 32 bit αριθμός της μορφής:

a = 10010101010101010100011111000101 και ένας αριθμός b = 181(bin 10110101) όπως και στην περίπτωσή μας. Ο αλγόριθμος του Karatsuba ισχυρίζεται ότι ο πολλαπλασιασμός a*b = a_uper16*b<<16 + a_lower16*b. Όπου a_uper16 είναι τα 16 περισσότερο σημαντικά bit του a και a_lower16 είναι τα λιγότερο σημαντικά bit του a. Έτσι για να εκτελέσω τον πολλαπλασιασμό, θα πρέπει αρχικά να κάνω τον πολλαπλασιασμό του 181 με τα 16 σημαντικότερα bit του a που στο παράδειγμα μας είναι ο αριθμός a_uper16 = 1001010101010101. Για να πάρω όμως τα 16 περισσότερο σημαντικά bit και να πετάξω τα λιγότερο σημαντικά, αρκεί να κάνω ολίσθηση 16 θέσεις δεξιά τον καταχωρητή που περιέχει τον a. Ο καταχωρητής που περιέχει το a θα είναι της μορφής 00000000000000001001010101010101. Η πράξη madd_epi16 όπως έχουμε αναλύσει και προηγουμένως κάνει πολλαπλασιασμό και πρόσθεση. Στην περίπτωσή μας, αν κάνουμε madd_epi16 τον καταχωρητή που περιέχει τον a και τον καταχωρητή που περιέχει τον b, θα πάρουμε ουσιαστικά το γινόμενο 1001010101010101*10110101 που είναι και το ζητούμενο. Το αποτέλεσμα θα είναι αποθηκευμένο σε ένα 32 bit καταχωρητή και πλέον πρέπει να το ολισθήσω 16 bit αριστερά. Για να ολοκληρωθεί ο πολλαπλασιασμός θα πρέπει να γίνει και ο πολλαπλασιασμός των λιγότερο σημαντικών bit με το 181. Ο πολλαπλασιασμό αυτός όμως θα πρέπει να είναι unsigned λόγω του ότι είναι λάθος να βάζω πρόσημο όταν πολλαπλασιάζω ένα αριθμό (181) με τα 16 λιγότερο σημαντικά bit ενός άλλου και να βάζω πρόσημο, αφού το πρόσημο μπαίνει στα περισσότερο σημαντικά bit όπως και έχω κάνει με την πράξη madd_epi16 προηγουμένως. Θέλω το αποτέλεσμα να είναι unsigned αρά θα χρησιμοποιήσω τις πράξεις intrinsic _mm_mulhi_epu16 και _mm_mullo_epi16. Οι δύο αυτές πράξεις δουλεύουνε ως εξής:

_mm_mulhi_epu16: μη προσημασμένος πολλαπλασιασμός των 8 περισσότερο σημαντικών bit δύο 16 bit αριθμών και αποθήκευση του αποτελέσματος σε καταχωρητή 16 bit.

_mm_mullo_epi16: πολλαπλασιασμός των 8 λιγότερο σημαντικών bit δύο 16 bit αριθμών και αποθήκευση του αποτελέσματος σε καταχωρητή 16 bit.

Για να ενοποιήσω τα δύο αυτά αποτελέσματα και να πάρω το a_lower16 αρκεί να κάνω ολίσθηση το αποτέλεσμα του mulhi κατά 16 θέσεις αριστερά και πλέον να κάνω or με το αποτέλεσμα του mullo.
Έχοντας a_higher16 και a_lower16 αρκεί να κάνω πρόσθεση τα δύο αυτά αποτελέσματα με add_epi32 για να πάρω το αποτέλεσμα του πολλαπλασιασμού.

4. Κατά την αποθήκευση των block μετά το πέρας της FAST IDCT είναι απαραίτητη η πράξη της μετατροπής των 32 bit αριθμών που χρησιμοποιούνται μέχρι εκείνη τη στιγμή σε 16 bit με ταυτόχρονο saturate αυτών. Όλη αυτή τη διαδικασία μπορούμε να την κάνουμε με μία μόνο πράξη την _mm_packs_epi32. Έχοντας σε δύο καταχωρητές τα περισσότερο και τα λιγότερο σημαντικά bit μιας γραμμής block θα πρέπει να τα τοποθετήσω σε ένα καταχωρητή XMM. Η πράξη _mm_packs_epi32 υλοποιεί την παρακάτω μακροεντολή:

```
r0 := SignedSaturate(a0)
r1 := SignedSaturate(a1)
r2 := SignedSaturate(a2)
r3 := SignedSaturate(a3)
r4 := SignedSaturate(b0)
r5 := SignedSaturate(b1)
r6 := SignedSaturate(b2)
r7 := SignedSaturate(b3),
```

που στην περίπτωσή μας είναι ακριβώς ότι χρειαζόμαστε.

3.3.1.3 FAST IDCT   μέσα από τη συνάρτηση Decode_MPEG2_Intra_Block

Η συνάρτηση FAST IDCT καλείται μέσα από τη συνάρτηση motion compensation για όλα τα block τα οποία αποκωδικοποιούνται, ακόμα και αν σε ένα block όλες οι τιμές είναι μηδενικές. Το να εκτελούνται ωστόσο οι πράξεις του IDCT όταν όλες οι τιμές των pixel σε ένα block είναι μηδενικές δεν είναι και ότι καλύτερο ως προς την ταχύτητα του κώδικα. Είναι εξάλλου προφανές ότι όταν όλες οι τιμές των pixel ενός block είναι μηδενικές τότε και το αποτέλεσμα της εφαρμογής του FAST IDCT πάνω σε αυτό το block θα είναι και αυτό μηδενικό για όλα τα pixel του block. Άρα σε αυτή την περίπτωση θα είναι καλό να αποφύγω την εκτέλεση του FAST IDCT. Η καλύτερη σε απόδοση επιλογή που έχω να κάνω χωρίς να επηρεάζεται το πρότυπο του MPEG-2 είναι να κληθεί η συνάρτηση του FAST IDCT κατά τη διάρκεια της αποκωδικοποίησης ενός block, και να καλείται εφόσον υπάρχει έστω και μια μη μηδενική τιμή κάποιου pixel στο αποκωδικοποιούμενο block. Η κατάλληλη θέση για να κληθεί η συνάρτηση FAST_IDCT είναι μέσα από τις δύο συναρτήσεις όπου γίνεται και η αποκωδικοποίηση των δεδομένων. Αυτές οι δύο συναρτήσεις είναι: **Decode_MPEG2_Intra_Block** και **Decode_MPEG2_Non_Intra_Block.** Η συνάρτηση θα πρέπει να κληθεί όταν tab->tun == 64, σε αντίθετη περίπτωση δηλαδή όταν tab->run == 65 τότε έχουμε escape και προφανώς δεν χρειάζεται να κάνουμε FAST IDCT καθώς όλα τα στοιχεία του block θα είναι μηδέν.

Ακολουθεί ένα κομμάτι του κώδικα από τη συνάρτηση **Decode_MPEG2_Intra_Block** όπου και καλείται η FAST_IDCT.

```
    if (tab->run==64) /* end_of_block */
    {
#ifdef TRACE
      if (Trace_Flag)
        printf("): EOB\n");
#endif /* TRACE */

        Fast_IDCT(bp);
      return;
    }
```

Κλήση της συνάρτησης FAST_IDCT μέσα από τις συναρτήσεις Decode_MPEG2_Intra_Block και Decode_MPEG2_Non_Intra_Block για να γίνεται κλήση της FAST_IDCT μόνο όταν είναι απαραίτητο(έστω και ένα ixel έχει μη μηδενική τιμή).

3.3.1.4 idct1x1-idct2x2-idct4x4

Όπως έχει αναφερθεί στην ενότητα 1.1.4 της εισαγωγής ο διακριτός μετασχηματισμός συνημιτόνου (DCT) είναι μια διαδικασία μετασχηματισμού η οποία έχει την ιδιότητα να συγκεντρώνει την ενέργεια στα pixel του block που είναι κοντά στο DC, όπως φαίνεται και στο παρακάτω σχήμα.

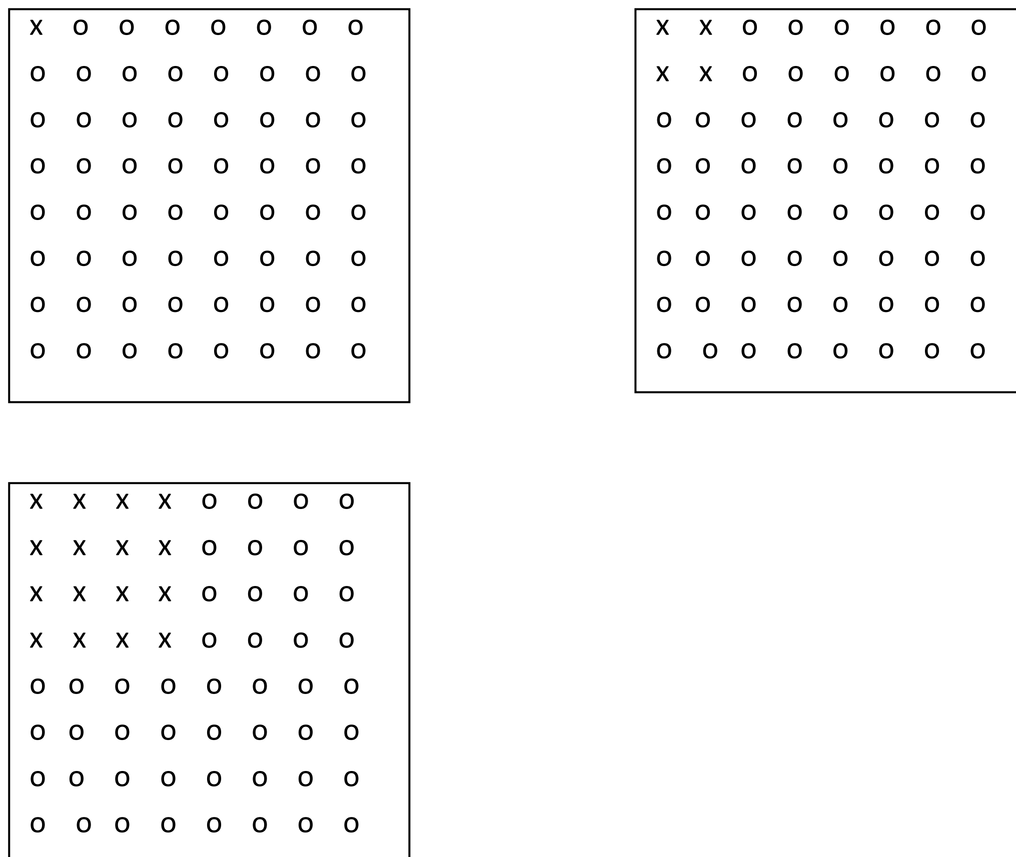


Image 3.3 Struct of block 1x1, 2x2, 4x4

Ύστερα από μετρήσεις που γίνανε σε κάποια video, όπως dantes.m2v προέκυψε ένα πολύ ενδιαφέρον στατιστικό. Τα block που έχουνε μόνο DC είναι 9%, όπως και αυτά που έχουνε μη μηδενικές τιμές μόνο στα τέσσερα πάνω αριστερά pixel(block 2x2) είναι 6% (εικόνα 2). Μία βελτίωση που θα μπορούσε να γίνει, είναι να δημιουργηθούνε idct1x1, idct2x2 και idct4x4 αντίστοιχα για κάθε περίπτωση και οι πράξεις του idct να μειωθούνε αρκετά καθώς τα περισσότερα pixel θα είναι 0 με αποτέλεσμα πολλές πράξεις να μηδενίζονται. Μάλιστα στην περίπτωση του idct1x1 οι πράξεις είναι ελάχιστες με συνέπεια να υπάρχει σχετικά σημαντική βελτίωση.

	1x1 block	2x2 block	4x4 block	all
Dantes.m2v	3786722	1034764	626149	5760848
Input_Inter.m2v	1000556	868326	1705098	7930955
Vin-4-p.m2v	634766	542845	1260492	4635259

Table 3.4 Number of blocks 1x1, 2x2, 4x4

	1x1 block	2x2 block	4x4 block	all
Dantes.m2v	65%	18%	11%	100%
Input_Inter.m2v	12,6%	11%	21,5%	100%
Vin-4-p.m2v	13,7%	11,7%	27%	100%

Table 3.5 Percentage of blocks 1x1, 2x2, 4x4

Idct1x1

Κατά τη διάρκεια της συνάρτησης idct1x1 όπως αναφέρθηκε και προηγουμένως θα έχω μόνο DC, δηλαδή το πάνω αριστερό pixel θα έχει μη μηδενική τιμή σε αντίθεση με όλα τα υπόλοιπα pixel που θα έχουνε μηδενική τιμή(σχήμα).

Παρατηρώντας τον reference code της συνάρτησης FAST_IDCT παρατηρώ ότι αν την εφαρμόσουμε σε block που έχει μόνο DC τότε ουσιαστικά κατά τη διάρκεια του idctrow αντιγράφεται η τιμή του DC σε όλα τα pixel της 1ης γραμμής του block. Με την εφαρμογή του idctcol είναι προφανές ότι θα γίνει το ίδιο. Τελικά προκύπτει ένα block όπου όλες οι τιμές του είναι ίδιες και ίσες με τη DC τιμή του block.

Ακολουθεί ο κώδικας του idct1x1:

```
static void idct1x1(blk)
short *blk;
{

        __m128i mm0;
        int a;

        a = (blk[0]+4>>3);

        mm0 = _mm_set1_epi16(a);


        //store
        _mm_store_si128 ((__m128i*)(blk) , mm0);
        _mm_store_si128 ((__m128i*)(blk+8) , mm0);
        _mm_store_si128 ((__m128i*)(blk+16) , mm0);
        _mm_store_si128 ((__m128i*)(blk+24) , mm0);
        _mm_store_si128 ((__m128i*)(blk+32) , mm0);
        _mm_store_si128 ((__m128i*)(blk+40) , mm0);
        _mm_storeu_si128 ((__m128i*)(blk+48) , mm0);
        _mm_store_si128 ((__m128i*)(blk+56) , mm0);

}
```

Εξετάζοντας το reference code παρατηρώ ότι κατά τη διάρκεια του idctrow έχω αριστερή ολίσθηση κατά 11 θέσεις όταν φορτώνεται το block[0]. Πριν την αποθήκευση για χρήση από το idctcol έχω δεξιά ολίσθηση κατά 8 θέσεις, άρα συνολικά έχω 3 θέσεις αριστερά. Στη συνέχεια καλείται η idctcol όπου με την ίδια νοοτροπία έχω τελικά ολίσθηση κατά 6 θέσεις δεξιά. Το συμπέρασμα είναι ότι συνολικά θα έχω ολίσθηση του DC κατά 3 θέσεις δεξιά.

Τοποθέτηση σε 16 bit θέσεις xmm καταχωρητή τις ίδιες τιμές a που είναι η τιμή του DC ολισθημένη κατά 3 θέσεις δεξιά.

Idct2x2

Η συνάρτηση idct2x2 όπως αναφέρθηκε και προηγουμένως θα έχει μη μηδενικές τιμές μόνο στα 4 πάνω αριστερά pixel(σχήμα). Οι πράξεις που θα πραγματοποιηθούν θα είναι μειωμένες σε σχέση με την εκτέλεση ολόκληρου του κώδικα fast_idct για 4 μόνο pixel. Ας δούμε όμως πιο αναλυτικά τον κώδικα που προκύπτει ύστερα από μηδενισμό αρκετών στοιχείων, άρα και παράκαμψη κάποιων πράξεων κατά τη διάρκεια των τριών σταδίων του idctrow και idctcol αντίστοιχα.

Μετά τις απλοποιήσεις που γίνανε ο κώδικας σε C που προκύπτει για τη συνάρτηση idctcol είναι ο παρακάτω:

Παρουσιάζεται ο αρχικός κώδικας σε C

```
static void idctrow(blk)
short *blk;
{
  int x0, x4;

  x0 = (blk[0]<<11) + 128;

  x8 = W7*x4;
  x4 = W1*x4;
  block[8] = (x0+x4)>>8;
  block[15] = (x0-x4)>>8;
  x2 = (181*(x4+x8)+128)>>8;
  x4 = (181*(x4-x8)+128)>>8;

  block[9] = (x0+x2)>>8;
  block[10] = (x0+x4)>>8;
  block[11] = (x0+x8)>>8;
  block[12] = (x0-x8)>>8;
  block[13] = (x0-x4)>>8;
```

Οι πράξεις εύκολα προκύπτουν αν αντικαταστήσω στον reference code τις τιμές των x1=x2=x3=x5=x6=x7=0 αφού μόνο το block[0] και block[1] έχουνε μη μηδενικές τιμές.

```
  block[14] = (x0-x2)>>8;

}
```

Οι πράξεις εύκολα προκύπτουν αν αντικαταστήσω στον reference code τις τιμές των x1=x2=x3=x5=x6=x7=0 αφού μόνο το block[0] και block[1] έχουνε μη μηδενικές τιμές.

```
static void idctcol(blk)
short *blk;
{
  int x0, x4;

  x0 = (block[i]<<8) + 8192;

  x8 = W7*x4 + 4;
  x4 = (x8+(W1-W7)*x4)>>3;
  x5 = x8>>3;

  x1 = x4;

  x2 = (181*(x4+x5)+128)>>8;
  x4 = (181*(x4-x5)+128)>>8;

  block[i] = iclp[(x0+x1)>>14];
  block[i+8] = iclp[(x0+x2)>>14];
  block[i+16] = iclp[(x0+x4)>>14];
  block[i+24] = iclp[(x0+x5)>>14];
  block[i+32] = iclp[(x0-x5)>>14];
  block[i+40] = iclp[(x0-x4)>>14];
  block[i+48] = iclp[(x0-x2)>>14];
  block[i+56] = iclp[(x0-x1)>>14];
}
```

Όπως έχουμε κάνει και στην περίπτωση του γενικού FAST IDCT όπου και χρησιμοποιούμε μόνο μία συνάρτηση την οποία και ονομάζουμε idct, έτσι και εδώ οι δύο συναρτήσεις idctcol2x2 και idctrow2x2 θα συγχωνευτούνε σε μία, την idct2x2. Παρουσιάζεται ενδεικτικά η συνάρτηση:

```
static void idct_2x2(blk)
short *blk;
{

        //gadem
        __m128i mm5,mm15,mm3,mm13,mm1,mm7,mm11,mm17,mm18,mm8;
        __m128i mm0,mm10,mm4,mm14,mm6,mm16,mm2,mm12;
        __m128i a0,a1,a2,a3,a4,a5,a6,a7,b0,tmp1;


        __m128i mmzero,mm128,mm181,mm8192,mmw1,mmw7,b1,b2,b3,mmfour;

                //i have done the transpose by the stage of quantization

                //loading the block 8x8
                mm0 = _mm_load_si128((__m128i*)(blk));        //0 1 2 3 4 5 6 7
                mm1 = _mm_load_si128((__m128i*)(blk+8));      //0 1 2 3 4 5 6 7

                mmzero = _mm_setzero_si128(); //0 0 0 0
                tmp1 = _mm_load_si128((__m128i*)(constants_32+0*4)); //1 1 1 1
                mm128 = _mm_load_si128((__m128i*)(constants_32+2*4));//128 128 128 128
                mm181 = _mm_load_si128((__m128i*)(constants_32+3*4));//181,181,181,181
                mm8192 = _mm_load_si128((__m128i*)(constants_32+4*4))//8192,…8192
                mmfour = _mm_load_si128((__m128i*)(constants_32+1*4));//4,4,4,4

                //x0 = (blk[8*0]<<11) + 128;
                mm10 = _mm_unpacklo_epi16 (mm0,mm0);
```

Εφόσον έχω idct2x2 αρκεί να φορτώσω τις δύο πρώτες γραμμές.

Φόρτωμα των σταθερών από τους αντίστοιχους πίνακες.

```
mm10 = _mm_madd_epi16 (mm10, tmp1);
mm10 = _mm_slli_epi32 (mm10,11);
mm10 = _mm_add_epi32 (mm10,mm128);
mm0 = _mm_unpackhi_epi16 (mm0,mm0);
mm0 = _mm_madd_epi16 (mm0, tmp1);
mm0 = _mm_slli_epi32 (mm0, 11);
mm0 = _mm_add_epi32 (mm0,mm128);




/********************************
*            idct row          *
*                              *
********************************/


//loading constants
mmw7 = _mm_load_si128((__m128i*)(constants+12*8));//W7,… W7


//x8 = W7*x4;
mm11 = _mm_unpacklo_epi16 (mm1, mmzero);
mm18 = _mm_madd_epi16 (mm11, mmw7);           //W7*x4
mm11 = _mm_unpackhi_epi16 (mm1, mmzero);
mm8 = _mm_madd_epi16 (mm11, mmw7);            //W7*x4


//block[8*3] = (x0+x8)>>8;
b0 = _mm_add_epi32 (mm10, mm18);
b0 = _mm_srai_epi32(b0 ,8);
a3 = _mm_add_epi32 (mm0, mm8);
a3 = _mm_srai_epi32(a3 ,8);
a3 = _mm_packs_epi32 (b0 ,a3);                //a3 packs to short




//block[8*4] = (x0-x8)>>8;
b0 = _mm_sub_epi32 (mm10, mm18);
b0 = _mm_srai_epi32(b0 ,8);
a4 = _mm_sub_epi32 (mm0, mm8);
a4 = _mm_srai_epi32(a4 ,8);
a4 = _mm_packs_epi32 (b0 ,a4);




//load constants
mmw1 = _mm_load_si128((__m128i*)(constants+2*8));    //w1,…w1

//x4 = W1*x4;
mm11 = _mm_unpacklo_epi16 (mm1,mmzero);
mm11 = _mm_madd_epi16 (mm11, mmw1);

mm1 = _mm_unpackhi_epi16 (mm1,mmzero);
mm1 = _mm_madd_epi16 (mm1, mmw1);


//block[8*0] = (x0+x4)>>8;
b0 = _mm_add_epi32 (mm10, mm11);
b0 = _mm_srai_epi32(b0 ,8);
a0 = _mm_add_epi32 (mm0, mm1);
a0 = _mm_srai_epi32(a0 ,8);
a0 = _mm_packs_epi32 (b0 ,a0);                        //a0 packs to short


//block[8*7] = (x0-x4)>>8;
b0 = _mm_sub_epi32 (mm10, mm11);
b0 = _mm_srai_epi32(b0 ,8);
a7 = _mm_sub_epi32 (mm0, mm1);
a7 = _mm_srai_epi32(a7 ,8);
a7 = _mm_packs_epi32 (b0 ,a7);                       //a7 packs to short
```

```
//x2 = (181*(x4+x8)+128)>>8;
b2 = _mm_add_epi32(mm11,mm18);
b1 = _mm_srai_epi32(b2,16);
b1 = _mm_madd_epi16(b1,mm181);
b3 = _mm_mulhi_epu16(b2,mm181);
b2 = _mm_mullo_epi16(b2,mm181);
b3 = _mm_slli_epi32(b3,16);
b2 = _mm_or_si128(b2,b3);
b1 = _mm_slli_epi32(b1,16);
b1 = _mm_add_epi32(b2,b1);
b1 = _mm_add_epi32(b1,mm128);
mm16 = _mm_srai_epi32(b1,8);


b2 = _mm_add_epi32(mm1,mm8);
b1 = _mm_srai_epi32(b2,16);
b1 = _mm_madd_epi16(b1,mm181);
b3 = _mm_mulhi_epu16(b2,mm181);
b2 = _mm_mullo_epi16(b2,mm181);
b3 = _mm_slli_epi32(b3,16);
b2 = _mm_or_si128(b2,b3);
b1 = _mm_slli_epi32(b1,16);
b1 = _mm_add_epi32(b2,b1);
b1 = _mm_add_epi32(b1,mm128);
mm6 = _mm_srai_epi32(b1,8);


//x4 = (181*(x4-x8)+128)>>8
b2 = _mm_sub_epi32(mm11,mm18);
b1 = _mm_srai_epi32(b2,16);
b1 = _mm_madd_epi16(b1,mm181);
b3 = _mm_mulhi_epu16(b2,mm181);
b2 = _mm_mullo_epi16(b2,mm181);
b3 = _mm_slli_epi32(b3,16);
b2 = _mm_or_si128(b2,b3);
b1 = _mm_slli_epi32(b1,16);
b1 = _mm_add_epi32(b2,b1);
b1 = _mm_add_epi32(b1,mm128);
mm11 = _mm_srai_epi32(b1,8);


b2 = _mm_sub_epi32(mm1,mm8);
b1 = _mm_srai_epi32(b2,16);
b1 = _mm_madd_epi16(b1,mm181);
b3 = _mm_mulhi_epu16(b2,mm181);
b2 = _mm_mullo_epi16(b2,mm181);
b3 = _mm_slli_epi32(b3,16);
b2 = _mm_or_si128(b2,b3);
b1 = _mm_slli_epi32(b1,16);
b1 = _mm_add_epi32(b2,b1);
b1 = _mm_add_epi32(b1,mm128);
mm1 = _mm_srai_epi32(b1,8);


//blk[8*1] = (x0+x2)>>8;
b0 = _mm_add_epi32 (mm10, mm16);
b0 = _mm_srai_epi32(b0 ,8);
a1 = _mm_add_epi32 (mm0, mm6);
a1 = _mm_srai_epi32(a1 ,8);
a1 = _mm_packs_epi32 (b0 ,a1);                              //a1 packs to short
```

```
//blk[8*2] = (x0+x4)>>8;
b0 = _mm_add_epi32 (mm10, mm11);
b0 = _mm_srai_epi32(b0 ,8);
a2 = _mm_add_epi32 (mm0, mm1);
a2 = _mm_srai_epi32(a2 ,8);
a2 = _mm_packs_epi32 (b0 ,a2);


//blk[8*5] = (x0-x4)>>8;
b0 = _mm_sub_epi32 (mm10, mm11);
b0 = _mm_srai_epi32(b0 ,8);
a5 = _mm_sub_epi32 (mm0, mm1);
a5 = _mm_srai_epi32(a5 ,8);
a5 = _mm_packs_epi32 (b0 ,a5);


//blk[8*6] = (x0-x2)>>8;
b0 = _mm_sub_epi32 (mm10, mm16);
b0 = _mm_srai_epi32(b0 ,8);
a6 = _mm_sub_epi32 (mm0, mm6);
a6 = _mm_srai_epi32(a6 ,8);
a6 = _mm_packs_epi32 (b0 ,a6);


//invert transpose
mm0 = _mm_unpacklo_epi16(a0,a1);
mm1 = _mm_unpackhi_epi16(a0,a1);
mm2 = _mm_unpacklo_epi16(a2,a3);
mm3 = _mm_unpackhi_epi16(a2,a3);
mm4 = _mm_unpacklo_epi16(a4,a5);
mm5 = _mm_unpackhi_epi16(a4,a5);
mm6 = _mm_unpacklo_epi16(a6,a7);
mm7 = _mm_unpackhi_epi16(a6,a7);
a0 = _mm_unpacklo_epi32(mm0,mm2);
a1 = _mm_unpackhi_epi32(mm0,mm2);
a2 = _mm_unpacklo_epi32(mm1,mm3);
a3 = _mm_unpackhi_epi32(mm1,mm3);
a4 = _mm_unpacklo_epi32(mm4,mm6);
a5 = _mm_unpackhi_epi32(mm4,mm6);
a6 = _mm_unpacklo_epi32(mm5,mm7);
a7 = _mm_unpackhi_epi32(mm5,mm7);
mm0 = _mm_unpacklo_epi64(a0,a4);
mm1 = _mm_unpackhi_epi64(a0,a4);
mm2 = _mm_unpacklo_epi64(a1,a5);
mm3 = _mm_unpackhi_epi64(a1,a5);
mm4 = _mm_unpacklo_epi64(a2,a6);
mm5 = _mm_unpackhi_epi64(a2,a6);
mm6 = _mm_unpacklo_epi64(a3,a7);
mm7 = _mm_unpackhi_epi64(a3,a7);
```

Εδώ εκτελείται η inverse transpose η οποία έχει αναλυθεί παραπάνω κατά τη διάρκεια υλοποίησης της FAST_IDCT.

```
/********************************
*                              *
*          idct_column          *
*                               *
********************************/


//x0 = (blk[8*0]<<8) + 8192;
mm10 = _mm_unpacklo_epi16 (mm0,mm0);
mm10 = _mm_madd_epi16 (mm10, tmp1);
mm10 = _mm_slli_epi32 (mm10, 8);
mm10 = _mm_add_epi32 (mm10,mm8192);
```

```
mm0 = _mm_unpackhi_epi16 (mm0,mm0);
mm0 = _mm_madd_epi16 (mm0, tmp1);
mm0 = _mm_slli_epi32 (mm0, 8);
mm0 = _mm_add_epi32 (mm0,mm8192);


//x8 = W7*x4 + 4
mm18 = _mm_unpacklo_epi16 (mm1, mmzero);
mm18 = _mm_madd_epi16 (mm18, mmw7);
mm18 = _mm_add_epi32 (mm18, mmfour);
mm8 = _mm_unpackhi_epi16 (mm1, mmzero);
mm8 = _mm_madd_epi16 (mm8, mmw7);
mm8 = _mm_add_epi32 (mm8, mmfour);



//x4 = (x8+(W1-W7)*x4)>>3;
a2 =_mm_load_si128((__m128i*)(constants+2*8));                //w1,…w1
a4 = _mm_load_si128((__m128i*)(constants+13*8));              //-w7,…-w7
mm11 = _mm_unpacklo_epi16 (a4, a2);
mm15 = _mm_unpacklo_epi16 (mm1,mm1);
mm11 = _mm_madd_epi16 (mm11, mm15);
mm11 = _mm_add_epi32 (mm11, mm18);
mm11 = _mm_srai_epi32 (mm11, 3);
mm17 = _mm_unpackhi_epi16 (a4, a2);
mm15 = _mm_unpackhi_epi16 (mm1,mm1);
mm17 = _mm_madd_epi16 (mm17, mm15);
mm17 = _mm_add_epi32 (mm17, mm8);
mm1 = _mm_srai_epi32 (mm17, 3);


//x5 = x8>>3;
mm17 = _mm_srai_epi32 (mm18, 3);
mm7 = _mm_srai_epi32 (mm8, 3);



//x1 = x4

//block[i] = iclp[(x0+x4)>>14];
a4 = _mm_add_epi32 (mm10, mm11);
a4 = _mm_srai_epi32(a4 ,14);
a2 = _mm_add_epi32 (mm0, mm1);
a2 = _mm_srai_epi32(a2 ,14);
a4 = _mm_packs_epi32 (a4 ,a2);
_mm_store_si128 ((__m128i*)blk , a4);        //store


//block[i+56] = iclp[(x0-x4)>>14];
a4 = _mm_sub_epi32 (mm10, mm11);
a4 = _mm_srai_epi32(a4 ,14);
a2 = _mm_sub_epi32 (mm0, mm1);
a2 = _mm_srai_epi32(a2 ,14);
a4 = _mm_packs_epi32 (a4 ,a2);
_mm_store_si128 ((__m128i*)(blk+56) , a4);   //store



//x2 = (181*(x4+x5)+128)>>8;
a2 = _mm_add_epi32(mm11,mm17);
a4 = _mm_srai_epi32(a2,16);
a4 = _mm_madd_epi16(a4,mm181);
a0 = _mm_mulhi_epu16(a2,mm181);
a2 = _mm_mullo_epi16(a2,mm181);
a0 = _mm_slli_epi32(a0,16);
a2 = _mm_or_si128(a2,a0);
a4 = _mm_slli_epi32(a4,16);
a4 = _mm_add_epi32(a2,a4);
a4 = _mm_add_epi32(a4,mm128);
mm16 = _mm_srai_epi32(a4,8);


a2 = _mm_add_epi32(mm1,mm7);
```

```
a4 = _mm_srai_epi32(a2,16);
a4 = _mm_madd_epi16(a4,mm181);
a0 = _mm_mulhi_epu16(a2,mm181);
a2 = _mm_mullo_epi16(a2,mm181);
a0 = _mm_slli_epi32(a0,16);
a2 = _mm_or_si128(a2,a0);
a4 = _mm_slli_epi32(a4,16);
a4 = _mm_add_epi32(a2,a4);
a4 = _mm_add_epi32(a4,mm128);
mm6 = _mm_srai_epi32(a4,8);


//x4 = (181*(x4-x5)+128)>>8
a2 = _mm_sub_epi32(mm11,mm17);
a4 = _mm_srai_epi32(a2,16);
a4 = _mm_madd_epi16(a4,mm181);
a0 = _mm_mulhi_epu16(a2,mm181);
a2 = _mm_mullo_epi16(a2,mm181);
a0 = _mm_slli_epi32(a0,16);
a2 = _mm_or_si128(a2,a0);
a4 = _mm_slli_epi32(a4,16);
a4 = _mm_add_epi32(a2,a4);
a4 = _mm_add_epi32(a4,mm128);
mm11 = _mm_srai_epi32(a4,8);


a2 = _mm_sub_epi32(mm1,mm7);
a4 = _mm_srai_epi32(a2,16);
a4 = _mm_madd_epi16(a4,mm181);
a0 = _mm_mulhi_epu16(a2,mm181);
a2 = _mm_mullo_epi16(a2,mm181);
a0 = _mm_slli_epi32(a0,16);
a2 = _mm_or_si128(a2,a0);
a4 = _mm_slli_epi32(a4,16);
a4 = _mm_add_epi32(a2,a4);
a4 = _mm_add_epi32(a4,mm128);
mm1 = _mm_srai_epi32(a4,8);


//blk[8*1] = iclp[(x0+x2)>>14];
tmp1 = _mm_add_epi32 (mm10, mm16);
tmp1 = _mm_srai_epi32(tmp1 ,14);
a1 = _mm_add_epi32 (mm0, mm6);
a1 = _mm_srai_epi32(a1 ,14);
a4 = _mm_packs_epi32 (tmp1 ,a1);
_mm_store_si128 ((__m128i*)(blk+8) , a4);


//blk[8*2] = iclp[(x0+x4)>>14];
a4 = _mm_add_epi32 (mm10, mm11);
a4 = _mm_srai_epi32(a4 ,14);
a2 = _mm_add_epi32 (mm0, mm1);
a2 = _mm_srai_epi32(a2 ,14);
a4 = _mm_packs_epi32 (a4 ,a2);
_mm_store_si128 ((__m128i*)(blk+16) , a4);


//blk[8*3] = iclp[(x0+x5)>>14];
a4 = _mm_add_epi32 (mm10, mm17);
a4 = _mm_srai_epi32(a4 ,14);
a2 = _mm_add_epi32 (mm0, mm7);
a2 = _mm_srai_epi32(a2 ,14);
a4 = _mm_packs_epi32 (a4 ,a2);
_mm_store_si128 ((__m128i*)(blk+24) , a4);
```

```
            //blk[8*4] = iclp[(x0-x5)>>14];
            a4 = _mm_sub_epi32 (mm10, mm17);
            a4 = _mm_srai_epi32(a4 ,14);
            a2 = _mm_sub_epi32 (mm0, mm7);
            a2 = _mm_srai_epi32(a2 ,14);
            a4 = _mm_packs_epi32 (a4 ,a2);
            _mm_store_si128 ((__m128i*)(blk+32) , a4);

            //blk[8*5] = iclp[(x0-x4)>>14];
            a4 = _mm_sub_epi32 (mm10, mm11);
            a4 = _mm_srai_epi32(a4 ,14);
            a2 = _mm_sub_epi32 (mm0, mm1);
            a2 = _mm_srai_epi32(a2 ,14);
            a4 = _mm_packs_epi32 (a4 ,a2);
            _mm_store_si128 ((__m128i*)(blk+40) , a4);



            //blk[8*6] = iclp[(x0-x2)>>14];
            a4 = _mm_sub_epi32 (mm10, mm16);
            a4 = _mm_srai_epi32(a4 ,14);
            a2 = _mm_sub_epi32 (mm0, mm6);
            a2 = _mm_srai_epi32(a2 ,14);
            a4 = _mm_packs_epi32 (a4 ,a2);
            _mm_store_si128 ((__m128i*)(blk+48) , a4);

}
```

Για να κληθούν οι συναρτήσεις idct, idct1x1 και idct2x2 και idct4x4 προφανώς και θα πρέπει να ξεχωρίζουμε σε ποια εκ των περιπτώσεών μας εμπίπτει η περίπτωση του αποκωδικοποιούμενου block και αναλόγως να επιλεγεί η κατάλληλη FAST IDCT που θα εφαρμοστεί. Αυτή η διαδικασία θα γίνει πριν την κλίση της συνάρτησης FAST_IDCT κατά τη διάρκεια αποκωδικοποίησης των block στη συνάρτηση *Decode_MPEG2_Intra_Block* και *Decode_MPEG2_Non_Intra_Block.* Έστω μια μεταβλητή orval. Θα πρέπει να βρούμε μια σχέση σύμφωνα με την οποία ανάλογα με την τιμή που θα παίρνει αυτή η μεταβλητή να καλείται και η αντίστοιχη συνάρτηση. Αρχικά θέλουμε να ξεχωρίσουμε τα block που είναι 1x1, 2x2, 4x4 και 8x8. Η σκέψη είναι ΄ότι θα πρέπει ανά σειρά και ανά στήλη να να ξέρουμε πόσα bit έχουμε μη μηδενικές τιμές. Στην περίπτωση του block 1x1 θα πρέπει μόνο το πάνω αριστερά pixel να είναι μη μηδενικό, στην περίπτωση του 2x2 τα τέσσερα πάνω αριστερά pixel να είναι μη μηδενικά κτλ. Ύστερα από σκέψη κατέληξα στην έκφραση που ανάλογα με τις τιμές που λαμβάνει μου προσδιορίζει και την περίπτωση του block που έχω. Αυτή η έκφραση είναι η: **orval |= ((j>>3)|(j&7))**.

Ας δούμε αναλυτικότερα πως αυτή λειτουργεί. Αρχικά η τιμή της orval είναι 0. Κατά την αποκωδικοποίηση ενός block η τιμή του j είναι η τιμή της θέσης του pixel που διαβάζεται. Με τον τρόπο που είναι γραμμένες οι συναρτήσεις *Decode_MPEG2_Intra_Block* και *Decode_MPEG2_Non_Intra_Block,* διαβάζονται μόνο τα μη μηδενικά στοιχεία με αποτέλεσμα όταν διαβάζεται μια τιμή του j να

σημαίνει ότι σε εκείνη τη θέση j υπάρχει μη μηδενική τιμή στο pixel. Η πράξη **j>>3** μας δίνει τη γραμμή που βρίσκεται το j. Αν το j βρίσκεται στην $2^{η}$ γραμμή τότε παίρνει τιμή 1, στη $3^{η}$ 2 κτλ. Η πράξη **j&7** μας βοηθά να κρατήσουμε σε ποια στήλη της γραμμής βρίσκεται το j. Έτσι αν βρίσκεται στην $1^{η}$ στήλη θα πάρει τιμή 0, στη $2^{η}$ 1 κτλ. Συνδυάζοντας τις δύο αυτές πράξεις με ‘or’ θα έχω σαν αποτέλεσμα την ακριβή θέση του pixel, δηλαδή σε ποια στήλη και ποια γραμμή βρίσκεται.

	Orval = 0	Orval = 1	Orval = 2 ή 3	Άλλο
Idct type	Idct1x1	Idct2x2	Idct4x4	Idct8x8

Table 3.6 Οι τιμές που παίρνει η orval

Αλλάζοντας τη συνάρτηση FAST_IDCT και κάνοντάς την να παίρνει και ένα δεύτερο όρισμα, την τιμή της orval θα είναι δυνατός ο διαχωρισμός των τύπων της εκάστοτε idct που θα πρέπει να εκτελεστεί.

Η λειτουργία των δύο συναρτήσεων *Decode_MPEG2_Intra_Block* και *Decode_MPEG2_Non_Intra_Block* θα παρουσιαστεί παρακάτω με την ευκαιρία της ενσωμάτωσης της συνάρτησης *saturate* σε αυτές.

3.3.2. Addblock

Η συνάρτηση addblock είναι και αυτή μία από τις πλέον δαπανηρές συναρτήσεις όπως φαίνεται και στη μέτρηση του της απόδοσης του reference code με το πρόγραμμα Vtune analyzer. Η συνάρτηση add_block είναι και αυτή μία συνάρτηση που ανήκει στην οικογένεια των συναρτήσεων motion compensation και μάλιστα είναι το τελευταίο βήμα για την ολοκλήρωσή της καθώς εκεί γίνεται το τελικό άθροισμα των residuals. Για να βελτιωθεί η συνάρτηση ως προς την ταχύτητα αρκεί να μετατρέψουμε των κώδικα που είναι γραμμένος σε C, σε κώδικα intrinsic για να εκμεταλλευτούμε τη δυνατότητα του SSE επεξεργαστή μας.

Παρατίθεται ο αρχικός κώδικας σε intrinsic ο οποίος είναι υπογραμμισμένος και δίπλα του είναι ο πηγαίος κώδικας σε C.

```
static void Add_Block(comp,bx,by,dct_type,addflag)
int comp,bx,by,dct_type,addflag;
{
 int cc,i,j, iincr;
 unsigned char *rfp,*rfph;
 short *bp;

 int mmncr;
 //variables
 __m128i mm0,mm1,mm2,mm3,mm4,mm5,mm6;
 __m64 mmone,mmtwo;
```

```
/* derive color component index */
/* equivalent to ISO/IEC 13818-2 Table 7-1 */
cc = (comp<4) ? 0 : (comp&1)+1; /* color component index */

if (cc==0)
{
 /* luminance */

 if (picture_structure==FRAME_PICTURE)
  if (dct_type)
  {
   /* field DCT coding */
   rfp = current_frame[0]
      + Coded_Picture_Width*(by+((comp&2)>>1)) + bx + ((comp&1)<<3);
   iincr = (Coded_Picture_Width<<1);
  }
  else
  {
   /* frame DCT coding */
   rfp = current_frame[0]
      + Coded_Picture_Width*(by+((comp&2)<<2)) + bx + ((comp&1)<<3);
   iincr = Coded_Picture_Width;
  }
 else
 {
  /* field picture */
  rfp = current_frame[0]
     + (Coded_Picture_Width<<1)*(by+((comp&2)<<2)) + bx + ((comp&1)<<3);
  iincr = (Coded_Picture_Width<<1);
 }
}
else
{
 /* chrominance */

 /* scale coordinates */
 if (chroma_format!=CHROMA444)
  bx >>= 1;
 if (chroma_format==CHROMA420)
  by >>= 1;
 if (picture_structure==FRAME_PICTURE)
 {
  if (dct_type && (chroma_format!=CHROMA420))
  {
   /* field DCT coding */
   rfp = current_frame[cc]
      + Chroma_Width*(by+((comp&2)>>1)) + bx + (comp&8);
   iincr = (Chroma_Width<<1);
  }
  else
  {
   /* frame DCT coding */
   rfp = current_frame[cc]
      + Chroma_Width*(by+((comp&2)<<2)) + bx + (comp&8);
   iincr = Chroma_Width ;
  }
 }
 else
 {
  /* field picture */
  rfp = current_frame[cc]
     + (Chroma_Width<<1)*(by+((comp&2)<<2)) + bx + (comp&8);
  iincr = (Chroma_Width<<1);
 }
}

bp = ld->block[comp];


if (addflag)
```

```
{
        //iincr += 8;

        //mm0=0
        mm0 = _mm_setzero_si128();
        mmncr = iincr<<1;


        for(i=0; i<4; i++)
        {

                mm1 = _mm_load_si128((__m128i*)(bp));            //load bp
                mm2 = _mm_load_si128((__m128i*)(bp+8));          //load bp+8

                mm3 = _mm_loadl_epi64((__m128i*)(rfp));          //load 1st rfp
                mm4 = _mm_loadl_epi64((__m128i*)(rfp+iincr));    //load 2nd rfp

                mm3 = _mm_unpacklo_epi8(mm3,mm0);                //unpack 1st rfp and do it 16bit
                mm4 = _mm_unpacklo_epi8(mm4,mm0);                //unpack 2nd rfp and do it 16bit

                mm1 = _mm_add_epi16(mm1,mm3);                    //add
                mm2 = _mm_add_epi16(mm2,mm4);

                mm1 = _mm_packus_epi16(mm1,mm0);                         //saturate to 8bit
                mm2 = _mm_packus_epi16(mm2,mm0);

                mmone = _mm_movepi64_pi64 (mm1);                         //do it 64 bit
                mmtwo = _mm_movepi64_pi64 (mm2);


                _mm_storel_epi64((__m128i*)(rfp), mm1);
                _mm_storel_epi64((__m128i*)(rfp+iincr), mm2);

                bp += 16;
                rfp += mmncr;

        }


else
{
         //iincr+=8;
        mm2 = _mm_set1_epi16(128);

        for(i=0;i<8;i++)
        {


                mm1 = _mm_load_si128((__m128i*)(bp));            //load bp
                mm1 = _mm_adds_epi16(mm1,mm2);                   //bp+128
                mm1 = _mm_packus_epi16(mm1,mm1);
                _mm_storel_epi64((__m128i*)(rfp), mm1);

                bp+=8;
                rfp+=iincr;

        }

}
}
```

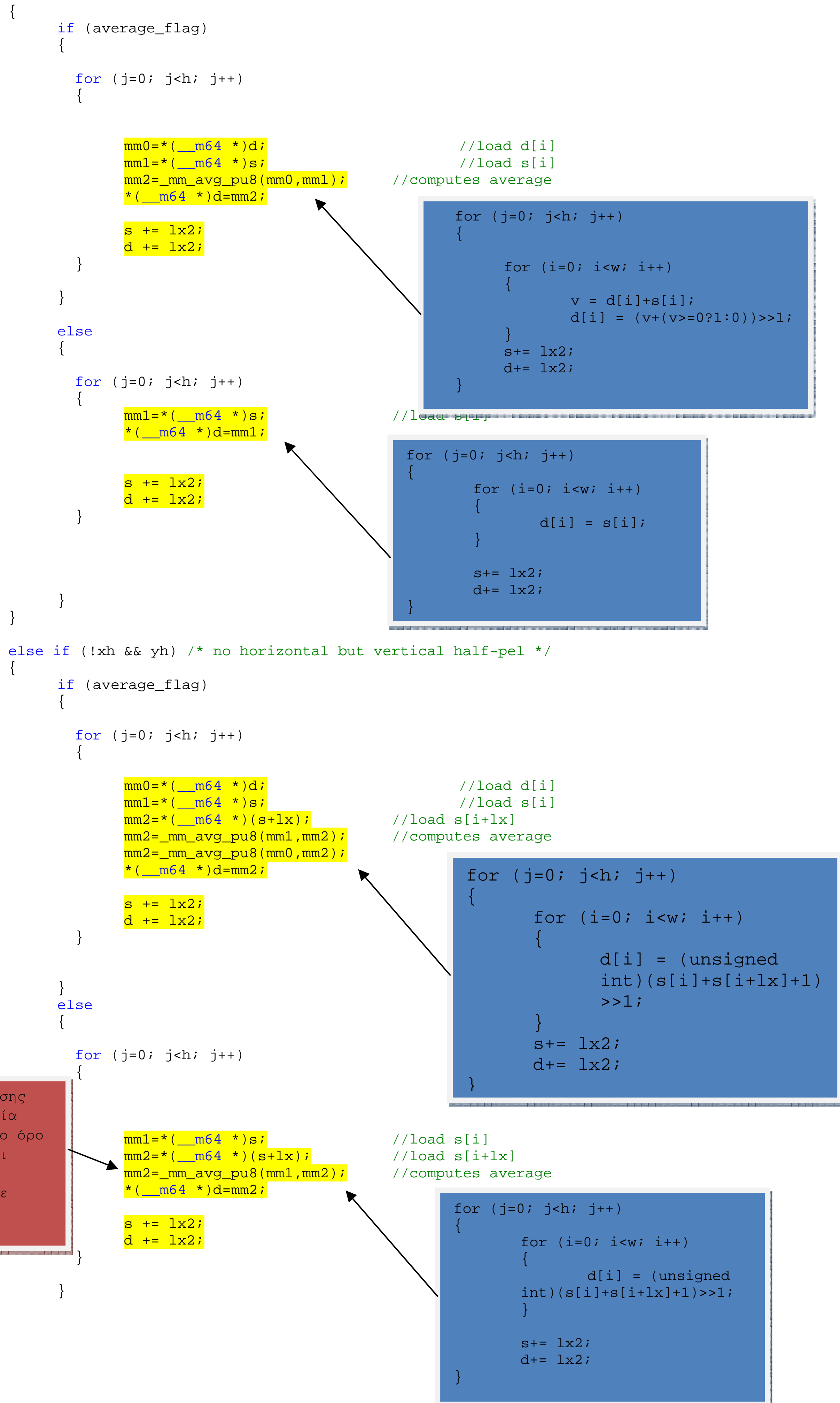
```
for (i=0; i<8; i++)
  {
   for (j=0; j<8; j++)
   {
    *rfp = Clip[*bp++ + *rfp];
    rfp++;
   }

   rfp+= iincr;
  }
```

```
for (i=0; i<8; i++)
{
 for (j=0; j<8; j++)
 *rfp++ = Clip[*bp++ + 128];

 rfp+= iincr;
}
```

3.3.3. Motion Compensation

Η συνάρτηση *form_component_prediction* είναι αρχικά η δεύτερη πιο απαιτητική σε επεξεργαστική ισχύ συνάρτηση. Εδώ ουσιαστικά εκτελείται η πράξη του motion compensation όπως και έχει αναλυθεί στο 1ο κεφάλαιο. Η κυρίως πράξη αυτής της συνάρτησης είναι η παρεμβολή μισού pixel και οι περιπτώσεις είναι άμεσα συνδεδεμένες με το τι πρότυπο YUV χρησιμοποιούμε, 4:2:0, 4:2:2 ή 4:4:4. Έτσι στην περίπτωση του 4:2:0 το μέγεθος του macroblock θα είναι 16x16 για το Y και 8x8 για τα U και V αντίστοιχα. Δεδομένου ότι οι τιμές των pixel είναι μεγέθους 8bit στην περίπτωση του Y χρειαζόμαστε καταχωρητές SSE (128bit), ενώ για τα U και V αρκούν καταχωρητές MMX (64bit). Έτσι, για να γίνει η μετατροπή στον κώδικα και για να μπορέσουμε να εκμεταλλευτούμε τη δυνατότητα των SSE καταχωρητών, ελέγχουμε σε ποια των περιπτώσων βρισκόμαστε και επιλέγουμε αναλόγως παρεμβολή με χρήση SSE (macroblock 16x16) ή MMX (macroblock 8x8). Παρατίθεται ο νέος κώδικας και δίπλα του είναι ο reference code.

```
static void form_component_prediction(src,dst,lx,lx2,w,h,x,y,dx,dy,average_flag)
 unsigned char *src;
 unsigned char *dst;
int lx;          /* raster line increment */
int lx2;
int w,h;
int x,y;
int dx,dy;
int average_flag;
/* flag that signals bi-directional or Dual-Prime
averaging (7.6.7.1 and 7.6.7.4). if average_flag==1,
a previously formed prediction has been stored in
                          pel_pred[] */
{
  int xint;
  int yint;
  int xh;
  int yh;
  int j;
  unsigned char *s;
  unsigned char *d;

  //gadem
  __m64 mm0,mm1,mm2,mm3,mm4,mm5,mm6,mmone;

  __m128i xmm0,xmm1,xmm2,xmm3,xmm4,xmm5,xmm6,xmmone8;


  mmone = _mm_set1_pi8 (1);
  xmmone8 = _mm_set1_epi8(1);


  /* half pel scaling for integer vectors */
  xint = dx>>1;
  yint = dy>>1;

  /* derive half pel flags */
  xh = dx & 1;
  yh = dy & 1;

  /* compute the linear address of pel_ref[][] and pel_pred[][]
     based on cartesian/raster cordinates provided */
  s = src + lx*(y+yint) + x + xint;
  d = dst + lx*y + x;

  if (w == 8)
  {
        if (!xh && !yh) /* no horizontal nor vertical half-pel */
```



```
{
      if (average_flag)
      {

        for (j=0; j<h; j++)
        {


              mm0=*(__m64 *)d;                          //load d[i]
              mm1=*(__m64 *)s;                          //load s[i]
              mm2=_mm_avg_pu8(mm0,mm1);         //computes average
              *(__m64 *)d=mm2;

              s += lx2;
              d += lx2;
        }

      }

      else
      {

        for (j=0; j<h; j++)
        {
              mm1=*(__m64 *)s;                  //load s[i]
              *(__m64 *)d=mm1;


              s += lx2;
              d += lx2;
        }



      }
}

else if (!xh && yh) /* no horizontal but vertical half-pel */
{
      if (average_flag)
      {

        for (j=0; j<h; j++)
        {

              mm0=*(__m64 *)d;                          //load d[i]
              mm1=*(__m64 *)s;                          //load s[i]
              mm2=*(__m64 *)(s+lx);             //load s[i+lx]
              mm2=_mm_avg_pu8(mm1,mm2);         //computes average
              mm2=_mm_avg_pu8(mm0,mm2);
              *(__m64 *)d=mm2;

              s += lx2;
              d += lx2;
        }


      }
      else
      {

        for (j=0; j<h; j++)
        {


              mm1=*(__m64 *)s;                  //load s[i]
              mm2=*(__m64 *)(s+lx);             //load s[i+lx]
              mm2=_mm_avg_pu8(mm1,mm2);         //computes average
              *(__m64 *)d=mm2;

              s += lx2;
              d += lx2;
        }

      }
```

```
                }

                else if (xh && !yh) /* horizontal but no vertical half-pel */
                {
                      if (average_flag)
                      {
                      for (j=0; j<h; j++)
                        {
                              mm0=*(__m64 *)d;                     //load d[i]
                              mm1=*(__m64 *)s;                     //load s[i]
                              mm2=*(__m64 *)(s+1);                 //load s[i+1]

                              mm2=_mm_avg_pu8(mm1,mm2);     //computes average
                              mm2=_mm_avg_pu8(mm0,mm2);
                              *(__m64 *)d=mm2;

                              s += lx2;
                              d += lx2;
                        }

                      }
```

```
for (j=0; j<h; j++)
{
for (i=0; i<w; i++)
{
v = d[i] + ((unsignedint)
(s[i]+s[i+1]+1)>>1);
d[i] = (v+(v>=0?1:0))>>1;
}
s+= lx2;
d+= lx2;
}
```

```
                      else
                      {
                        for (j=0; j<h; j++)
                        {

                              mm1=*(__m64 *)s;              //load s[i]
                              mm2=*(__m64 *)(s+1);          //load s[i+1]
                              mm2=_mm_avg_pu8(mm1,mm2);     //computes average
                              *(__m64 *)d=mm2;

                              s += lx2;
                              d += lx2;
                        }

                      }
              }
```

```
for (j=0; j<h; j++)
{
for (i=0; i<w; i++)
{
d[i] = (unsigned
int)(s[i]+s[i+1]+1)>>1;
}
s+= lx2;
d+= lx2;
}
```

```
                else /* if (xh && yh) horizontal and vertical half-pel */
                {
                      if (average_flag)
                      {

                        for (j=0; j<h; j++)
                        {

                              //d = s[i] + s[i+1] + s[i+lx] + s[i+lx+1]
                              //t = avg(s[i],s[i+1])
                              //z = avg(s[i+lx],s[i+lx+1])
                              //d = avg(s+t) - (((s[i] ^ s[i+1])|(s[i+lx]^s[i+lx+1]))&(z^t))&1

                              mm0 = *(__m64 *)d;     //load d[i]
                              mm1 = *(__m64 *)s;     //load s[i]
                              mm2 = *(__m64 *)(s+1); //load s[i+1]
                              mm5 = _mm_xor_si64(mm1,mm2);//(s[i] ^ s[i+1])

                              mm2 = _mm_avg_pu8(mm1,mm2);//computes average t
                              mm3 = *(__m64 *)(s+lx);               //load s[i+lx]
                              mm4 = *(__m64 *)(s+lx+1);      //load s[i+lx+1]
                              mm6 = _mm_xor_si64 (mm3,mm4); //(s[i+lx]^s[i+lx+1])

                              mm3 = _mm_avg_pu8(mm3,mm4);    //computes average z
```

```
for (j=0; j<h; j++)
{
for (i=0; i<w; i++)
{
v = d[i] + ((unsigned
int)(s[i]+s[i+1]+s[i+lx]+s[i+lx+1]
+2)>>2);
d[i] = (v+(v>=0?1:0))>>1;
}

s+= lx2;
d+= lx2;
}
```

```
            mm1 = _mm_avg_pu8(mm3,mm2);   //computes avg(z,t)

            mm2 = _mm_xor_si64(mm2,mm3);  //(z^t)

            mm5 = _mm_or_si64(mm5,mm6);
            //((s[i] ^ s[i+1])|(s[i+lx]^s[i+lx+1]))

            mm2 = _mm_and_si64(mm2,mm5);
            //(((s[i] ^ s[i+1])|(s[i+lx]^s[i+lx+1]))&(z^t))

            mm2 = _mm_and_si64(mm2,mmone);
            mm1 = _mm_sub_pi8(mm1,mm2);
            mm0 = _mm_avg_pu8(mm0,mm1);

            *(__m64 *)d = mm0;


            s += lx2;
            d += lx2;
      }

    }
    else
    {


      for (j=0; j<h; j++)
      {

            //t = avg(s[i],s[i+1])
            //z = avg(s[i+lx],s[i+lx+1])
            //d = avg(s+t) - (((s[i] ^ s[i+1])|(s[i+lx]^s[i+lx+1]))&(z^t))&1

            mm1 = *(__m64 *)s;            //load s[i]
            mm2 = *(__m64 *)(s+1);        //load s[i+1]
            mm5 = _mm_xor_si64(mm1,mm2);  //(s[i] ^ s[i+1])

            mm2 = _mm_avg_pu8(mm1,mm2);   //computes average t
            mm3 = *(__m64 *)(s+lx);               //load s[i+lx]
            mm4 = *(__m64 *)(s+lx+1);     //load s[i+lx+1]
            mm6 = _mm_xor_si64 (mm3,mm4); //(s[i+lx]^s[i+lx+1])

            mm3 = _mm_avg_pu8(mm3,mm4);   //computes average z
            mm1 = _mm_avg_pu8(mm3,mm2);   //computes avg(z,t)

            mm2 = _mm_xor_si64(mm2,mm3);  //(z^t)
            mm5 = _mm_or_si64(mm5,mm6);   //((s[i] ^
                                          s[i+1])|(s[i+lx]^s[i+lx+1]))

            mm2 = _mm_and_si64(mm2,mm5);
            //(((s[i] ^ s[i+1])|(s[i+lx]^s[i+lx+1]))&(z^t))

            mm2 = _mm_and_si64(mm2,mmone);

            mm1 = _mm_sub_pi8(mm1,mm2);

            *(__m64 *)d = mm1;


            s += lx2;
            d += lx2;
      }


    }
  }
}
```

```
for (j=0; j<h; j++)
{
for (i=0; i<w; i++)
{
d[i] = (unsigned
int)(s[i]+s[i+1]+s[i+lx]+s[i+lx+1]
+2)>>2;
}

s+= lx2;
d+= lx2;
}
```

```
//SSE-128bit registers

else
{

  if (!xh && !yh) /* no horizontal nor vertical half-pel */
  {
        if (average_flag)
        {

          for (j=0; j<h; j++)
          {

                xmm0 = _mm_loadu_si128((__m128i*)(d));        //load d[i]
                xmm1 = _mm_loadu_si128((__m128i*)(s));        //load s[i]

                xmm2 = _mm_avg_epu8(xmm0,xmm1);       //computes average
                _mm_storeu_si128 ((__m128i*)(d) , xmm2);


                s+= lx2;
                d+= lx2;


          }

        }


        else
        {

          for (j=0; j<h; j++)
          {
                        //memcpy
                        //load s[i]
                        xmm1 = _mm_loadu_si128((__m128i*)(s));

                        _mm_storeu_si128 ((__m128i*)(d) , xmm1);


                s+= lx2;
                d+= lx2;
          }



        }
  }




  else if (!xh && yh) /* no horizontal but vertical half-pel */
  {
        if (average_flag)
        {

          for (j=0; j<h; j++)
          {

                xmm0 = _mm_loadu_si128((__m128i*)(d));        //load d[i]
                xmm1 = _mm_loadu_si128((__m128i*)(s));        //load s[i]
                xmm2 = _mm_loadu_si128((__m128i*)(s+lx));     //load s[i+lx]

                xmm2=_mm_avg_epu8(xmm1,xmm2);                 //computes average
                xmm2=_mm_avg_epu8(xmm0,xmm2);

                _mm_storeu_si128 ((__m128i*)(d) , xmm2);


                s+= lx2;
```

```
for (j=0; j<h; j++)
{
for (i=0; i<w; i++)
{
v = d[i]+s[i];
d[i] = (v+(v>=0?1:0))>>1;
}

s+= lx2;
d+= lx2;
}
```

```
for (j=0; j<h; j++)
{
for (i=0; i<w; i++)
{
d[i] = s[i];
}

s+= lx2;
d+= lx2;
}
```

```
for (j=0; j<h; j++)
{
for (i=0; i<w; i++)
{
d[i] = (unsigned
int)(d[i]+s[i+lx]+1)>>1;
}

s+= lx2;
d+= lx2;
}
```

```
                d+= lx2;
            }


        }
        else
        {

            for (j=0; j<h; j++)
            {


                xmm1 = _mm_loadu_si128((__m128i*)(s));        //load s[i]
                xmm2 = _mm_loadu_si128((__m128i*)(s+lx));     //load s[i+lx]

                xmm2 = _mm_avg_epu8(xmm1,xmm2);               //computes average

                _mm_storeu_si128 ((__m128i*)(d) , xmm2);

                s+= lx2;
                d+= lx2;
            }
```

```
for (j=0; j<h; j++)
{
for (i=0; i<w; i++)
{
d[i] = (unsigned
int)(s[i]+s[i+lx]+1)>>1;
}

s+= lx2;
d+= lx2;
}
```

```
        }
    }
    else if (xh && !yh) /* horizontal but no vertical half-pel */
    {
        if (average_flag)
        {

            for (j=0; j<h; j++)
            {


                xmm0 = _mm_loadu_si128((__m128i*)(d));        //load d[i]
                xmm1 = _mm_loadu_si128((__m128i*)(s));        //load s[i]
                xmm2 = _mm_loadu_si128((__m128i*)(s+1));      //load s[i+1]

                xmm2 = _mm_avg_epu8(xmm1,xmm2);               //computes average
                xmm2 = _mm_avg_epu8(xmm2,xmm0);
                _mm_storeu_si128 ((__m128i*)(d) , xmm2);


                s+= lx2;
                d+= lx2;
            }
```

```
for (j=0; j<h; j++)
{
for (i=0; i<w; i++)
{
v = d[i] + ((unsigned
int)(s[i]+s[i+1]+1)>>1);
d[i] = (v+(v>=0?1:0))>>1;
}

s+= lx2;
d+= lx2;
}
```

```
        }
        else
        {
            for (j=0; j<h; j++)
            {

                xmm1 = _mm_loadu_si128((__m128i*)(s));        //load s[i]



                xmm2 = _mm_loadu_si128((__m128i*)(s+1));      //load s[i+1]
                xmm2 = _mm_avg_epu8(xmm1,xmm2);               //computes average
                _mm_storeu_si128 ((__m128i*)(d) , xmm2);



                s+= lx2;
                d+= lx2;
            }
```

```
for (j=0; j<h; j++)
{
for (i=0; i<w; i++)
{
d[i] = (unsigned
int)(s[i]+s[i+1]+1)>>1;
}

s+= lx2;
d+= lx2;
}
```

```


        }
    }
    else /* if (xh && yh) horizontal and vertical half-pel */
    {
        if (average_flag)
        {
```

Υπολογισμός για το ζεύγος των bit με χρήση της xor(παρατήρηση 3)

Υπολογισμός μέσου όρου δύο γειτονικών pixel με χρήση της _mm_avg_epu8(παρατήρηση).

```
for (j=0; j<h; j++)
{
for (i=0; i<w; i++)
{
v = d[i] + ((unsigned int)(s[i]+s[i+1]+s[i+lx]+
s[i+lx+1]+2)>>2);
d[i] = (v+(v>=0?1:0))>>1;
}

s+= lx2;
d+= lx2;
}
```

Υπολογισμός μέσου όρου των δύο μέσων όρων που έχουν υπολογιστεί με χρήση της _mm_avg_epu8 για 3η φορά και κατόπιν χρήση της xor για να περιοριστούμε σε μία περίπτωση όπου και παρουσιάζεται το σφάλμα.

```
for (j=0; j<h; j++)
{

    //d = s[i] + s[i+1] + s[i+lx] + s[i+lx+1]
    //t = avg(s[i],s[i+1])
    //z = avg(s[i+lx],s[i+lx+1])
    //d = avg(s+t) - (((s[i] ^
    s[i+1])|(s[i+lx]^s[i+lx+1]))&(z^t))&1

    xmm0 = _mm_loadu_si128((__m128i*)(d));          //load d[i]
    xmm1 = _mm_loadu_si128((__m128i*)(s));          //load s[i]
    xmm2 = _mm_loadu_si128((__m128i*)(s+1));        //load s[i+1]
    xmm5 = _mm_xor_si128(xmm1,xmm2);                //(s[i] ^ s[i+1])
    xmm2 = _mm_avg_epu8(xmm1,xmm2);                 //computes avg t

    xmm3 = _mm_loadu_si128((__m128i*)(s+lx));       //load s[i+lx]
    xmm4 = _mm_loadu_si128((__m128i*)(s+lx+1));     //load s[i+lx+1]
    xmm6 = _mm_xor_si128 (xmm3 , xmm4);          //(s[i+lx]^s[i+lx+1])
    xmm3 = _mm_avg_epu8(xmm3,xmm4);                 //computes avg z

    xmm1 = _mm_avg_epu8(xmm3,xmm2);                 //computes avg(z,t)
    xmm2 = _mm_xor_si128(xmm2,xmm3);                //(z^t)

    xmm5 = _mm_or_si128(xmm5,xmm6);                 //((s[i] ^
                                                    s[i+1])|(s[i+lx]^
                                                    s[i+lx+1]))

    xmm2 = _mm_and_si128(xmm2,xmm5);                //(((s[i] ^
                                                    s[i+1])|(s[i+lx]^
                                                    s[i+lx+1]))&(z^t))

    xmm2 = _mm_and_si128(xmm2,xmmone8);
    xmm1 = _mm_sub_epi8(xmm1,xmm2);
    xmm0 = _mm_avg_epu8(xmm0,xmm1);

    _mm_storeu_si128 ((__m128i*)(d) , xmm0);


    s+= lx2;
    d+= lx2;
}

}

else
{

for (j=0; j<h; j++)
{

    //t = avg(s[i],s[i+1])
    //z = avg(s[i+lx],s[i+lx+1])
    //d = avg(s+t) - (((s[i] ^
    s[i+1])|(s[i+lx]^s[i+lx+1]))&(z^t))&1


    xmm1 = _mm_loadu_si128((__m128i*)(s));          //load s[i]
    xmm2 = _mm_loadu_si128((__m128i*)(s+1));        //load s[i+1]
    xmm5 = _mm_xor_si128(xmm1,xmm2);                //(s[i] ^ s[i+1])

    xmm2 = _mm_avg_epu8(xmm1,xmm2);                 //computes avg t

    xmm3 = _mm_loadu_si128((__m128i*)(s+lx));       //load s[i+lx]
    xmm4 = _mm_loadu_si128((__m128i*)(s+lx+1));     //load s[i+lx+1]
    xmm6 = _mm_xor_si128 (xmm3 , xmm4);          //(s[i+lx]^s[i+lx+1])

    xmm3 = _mm_avg_epu8(xmm3,xmm4);                 //computes avg z
    xmm1 = _mm_avg_epu8(xmm3,xmm2);                 //computes avg(z,t)

    xmm2 = _mm_xor_si128(xmm2,xmm3);                //(z^t)

    xmm5 = _mm_or_si128(xmm5,xmm6);                 //((s[i] ^
                                                    s[i+1])|(s[i+lx]^
                                                    s[i+lx+1]))
```

```
for (j=0; j<h; j++)
{
for (i=0; i<w; i++)
{
d[i] = (unsigned
int)(s[i]+s[i+1]+s[i+lx]+
s[i+lx+1]+2)>>2;
}

s+= lx2;
d+= lx2;
}
```

```
                xmm2 = _mm_and_si128(xmm2,xmm5);            //(((s[i] ^
                                                            s[i+1])|(s[i+lx]^
                                                            s[i+lx+1]))&(z^t))

                xmm2 = _mm_and_si128(xmm2,xmmone8);
                xmm1 = _mm_sub_epi8(xmm1,xmm2);

                _mm_storeu_si128 ((__m128i*)(d) , xmm1);

                s+= lx2;
                d+= lx2;
            }

        }
    }

  }
}
```

Παρατηρήσεις

1. Όπως αναφέρεται και παραπάνω θα πρέπει να γίνει διαχωρισμός για τον αν η παρεμβολή θα γίνει για macroblock 16x16 είτε για macroblock 8x8. Για να γίνει ο διαχωρισμός στους δύο κώδικες αρκεί ένα if branch όπου αν w=8 τότε θα εκτελείται ο κώδικας με MMX καταχωρητές και αν w=16 τότε εκτελείται ο κώδικας με SSE καταχωρητές.

2. Στον αλγόριθμο του MPEG-2 περιγράφεται η διαδικασία της παρεμβολής μισού pixel, δηλαδή υπάρχει η περίπτωση το motion vector που αποκωδικοποιείται να αντιστοιχεί ανάμεσα σε 2 pixel. Έτσι θα πρέπει η τιμή να παίρνει το μέσο όρο των τιμών των 2 pixel. Η πράξη που περιγράφει την παραπάνω διαδικασία είναι η (a+b+1)>>1. Στη βιβλιοθήκη με τα intrinsic υπάρχει η πράξη που μας λύνει τα χέρια όσον αφορά την horizontal ή vertical ακρίβεια μισού pixel. Αυτή η πράξη είναι η _mm_avg_pu8(mma,mmb) για mmx registers και η _mm_avg_epu8(mma,mmb) για SSE registers. Δηλαδή με μία μόνο εντολή μπορώ να εκτελέσω τις 3 προσθέσεις και μια ολίσθηση.

3. Η παραπάνω παρατήρηση με τη χρήση του μέσου όρου, λειτουργεί όταν έχω horizontal ή vertical παρεμβολή μισού pixel. Τι γίνεται όταν έχω παρεμβολή μισού pixel horizontal και vertical ταυτόχρονα; Η πράξη σε αυτή την περίπτωση είναι η (a+b+c+d+2)>>2. Η πράξη αυτή μπορεί να γίνει με 3 πράξεις μέσου όρου (_mm_avg_pu8()) καθώς δεν υπάρχει η συγκεκριμένη πράξη((a+b+c+d+2)>>2) σε intrinsic και θα προσπαθήσουμε να χρησιμοποιήσουμε τις ήδη υπάρχουσες πράξεις.
   Η πράξη (a+b+c+d+2)>>2 είναι παρόμοια με την πράξη ((a+b+1)>>1 + (c+d+1)>>1 +1)>>1. Άρα, αρκεί μια πράξη _mm_avg_pu8(mma,mmb) για το (a+b+1)>>1, μία _mm_avg_pu8(mmc,mmd) για (c+d+1)>>1 και άλλη μια για

τον μεταξύ τους μέσο όρο. Ωστόσο σε χρήση αυτών των πράξεων θα υπάρχει ένα μικρό σφάλμα όταν μόνο ένας εκ των 4 αριθμών έχει σαν τελευταίο bit 1, ενώ οι άλλοι 3 έχουνε σαν τελευταίο bit το 0. Σε όλες τις άλλες περιπτώσεις το αποτέλεσμα των δύο αυτών πράξεων είναι το ίδιο. Το πότε έχω σφάλμα γίνεται αντιληπτό και από τον παρακάτω πίνακα όπου και φαίνονται οι δυνατές τιμές των δύο διαφορετικών προσεγγίσεων για τον υπολογισμό της ακρίβειας μισού pixel για ταυτόχρονη παρεμβολή horizontal και vertical.

Least bit				Original half pixel	A	B	Average A,B
a	b	c	d	(a+b+c+d+2)>>2	(a+b+1)>>1	(c+d+1)>>1	(A+B+1)>>1
0	0	0	0	0	0	0	0
0	0	0	1	0	0	1	1
0	0	1	0	0	0	1	1
0	0	1	1	1	0	1	1
0	1	0	0	0	1	0	1
0	1	0	1	1	1	1	1
0	1	1	0	1	1	1	1
0	1	1	1	1	1	1	1
1	0	0	0	0	1	0	1
1	0	0	1	1	1	1	1
1	0	1	0	1	1	1	1
1	0	1	1	1	1	1	1
1	1	0	0	1	1	0	1
1	1	0	1	1	1	1	1
1	1	1	0	1	1	1	1
1	1	1	1	1	1	1	1

Table 3.7

Έχοντας βρει πλέον την περίπτωση σφάλματος αρκεί στη συγκεκριμένη περίπτωση να εφαρμοστεί μια πράξη τέτοια ώστε να υπάρξει διόρθωση (correction) του αποτελέσματος ώστε να γίνει ίσο με το αποτέλεσμα της πράξης (a+b+c+d+2)>>2.

Η σκέψη για να γίνει αυτή η διόρθωση είναι με κάποιο τρόπο να μπορέσουμε να απομονώσουμε την περίπτωση του σφάλματος, δηλαδή μόνο σε έναν εκ των τεσσάρων αριθμών το least significant bit να είναι 1. Η πράξη xor μπορεί να ξεχωρίσει αν δύο bit είναι διαφορετικά, πχ αν a=0 και b=1 τότε a xor b=1, ενώ αν a και b έχουνε ίδια τιμή τότε a xor b =0. Κάνοντας (a xor b)or(c xor d) εξάγεται το συμπέρασμα αν 1 ή και τα 2 ζεύγη least significant bit είναι της μορφής 0, 1 και συνδυάζοντας τις δύο παραστάσεις με or ουσιαστικά καταλήγουμε στο συμπέρασμα της απομόνωσης δύο

πιθανών μορφών least significant bit 0,0,0,1 ή 0,1,0,1 αν το αποτέλεσμα της or είναι 1. Αρκεί τώρα να μπορέσουμε να ξεχωρίσουμε τις δύο παραπάνω περιπτώσεις. Αυτό είναι απλό να υπολογιστεί χρησιμοποιώντας ξανά τη δυαδική πράξη xor, αυτή τη φορά ανάμεσα στα δύο αποτελέσματα average που έχω υπολογίσει (φαίνεται και από τον παραπάνω πίνακα) ώστε να μας είναι γνωστό αν και τα δύο προηγούμενα αποτελέσματα είναι 1 ή μόνο το ένα εκ των δύο, το οποίο είναι και το ζητούμενο. Τελικά η ζητούμενη παράσταση που μας δίνει την περίπτωση του σφάλματος είναι:
(((a ^ b)|(c ^ d))&(avg(a,b)^avg(c,d))).

Το σφάλμα όπως εξηγήθηκε και όπως φαίνεται και από τον πιίνακα είναι ότι στην παραπάνω περίπτωση χρήσης 3 μέσων όρων το αποτέλεσμα θα είναι μεγαλύτερο κατά 1, έτσι αφαιρώντας το αποτέλεσμα της πράξης (((a ^ b)|(c ^ d))&(avg(a,b)^avg(c,d))) προκύπτει το αρχικό και ταυτόχρονα ακριβές αποτέλεσμα για horizontal και vertical παρεμβολής μισού pixel ταυτόχρονα.
Η τελική σχέση θα είναι: (a+b+c+d+2)>>2 = ((a+b+1)>>1+(c+d+1)>>1)>>1 -(((a ^ b)|(c ^ d))&(((a+b+1)>>1)^((c+d+1)>>1))) την οποία και εφαρμώζουμε ώστε μέσα από τη χρήση τριών μέσων όρων να καταλήξουμε στο αρχικό αποτέλεσμα.

3.3.4. Saturate

Η συνάρτηση saturate είναι και αυτή στην κατηγορία των συναρτήσεων που κοστίζουν αρκετούς κύκλους μηχανής, όχι τόσο γιατί είναι πολύπλοκη στην υλοποίησή της, αλλά γιατί εφαρμόζεται για όλα τα pixel ενός video. Ουσιαστικά η συνάρτηση saturate περιορίζει τις τιμές εισόδου της σε τιμές από -2048 έως και 2047. Εφαρμόζεται πριν τη συνάρτηση της FAST_IDCT και ουσιαστικά είναι αυτή που περιορίζει τις τιμές εισόδου σε 12bit όπως αναφέρεται χαρακτηριστικά και από το πρότυπο του H262.

Η αρχική ιδέα για βελτιστοποίηση είναι από κώδικα με εντολές C να γίνει μετατροπή σε κώδικα με εντολές intrinsic. Παρακάτω παρουσιάζεται το εγχείρημα και επισυνάπτεται τόσο ο reference code όσο και ο παραγόμενος κώδικας με τη χρήση intrinsic.

Saturate C code

```
static void Saturate(Block_Ptr)
short *Block_Ptr;
{
  int i, sum, val;

  sum = 0;

  /* ISO/IEC 13818-2 section 7.4.3:
Saturation */
  for (i=0; i<64; i++)
  {
    val = Block_Ptr[i];

    if (val>2047)
      val = 2047;
    else if (val<-2048)
      val = -2048;

    Block_Ptr[i] = val;
    sum+= val;
  }



 /* ISO/IEC 13818-2 section 7.4.4:
Mismatch control */
  if ((sum&1)==0)
    Block_Ptr[63]^= 1;

}
```

saturate intrinsics

```
static void Saturate(Block_Ptr)
short *Block_Ptr;
{
  int i, sum;
  int  val;
  short *k;
  __m128i mm0,mm1,mm2,mm3;

  sum = 0;

  mm0 = _mm_set1_epi16(30720);
  mm1 = _mm_setzero_si128();

  k=(short *)malloc(sizeof(short)*8);

  for(i=0;i<8;i++)
  {
        mm2=_mm_loadu_si128 ((__m128i *) Block_Ptr);

        mm3=_mm_subs_epi16 (mm2, mm0);
        mm3=_mm_adds_epi16 (mm3, mm0);
        mm3=_mm_adds_epi16 (mm3, mm0);
        mm3=_mm_subs_epi16 (mm3, mm0);

        //store back
        _mm_storeu_si128 ((__m128i*)Block_Ptr , mm3);

        mm1=_mm_add_epi16 (mm1, mm3);

        //next 8 blocks
        Block_Ptr+=8;

  }


 _mm_storeu_si128 ((__m128i *)k, mm1);


  for(i=0;i<8;i++)
  {
          sum+=k[i];
  }

  _mm_empty();

    /* ISO/IEC 13818-2 section 7.4.4: Mismatch
control */
  if ((sum&1)==0)
    Block_Ptr[63]^= 1;

}
```

Όπως προαναφέρθηκε η συνάρτηση saturate είναι μία συνάρτηση που περιορίζει τις αποκωδικοποιούμενες τιμές των pixel σε τιμές 12bit. Έτσι ο κώδικας που χρήζει μετατροπής είναι η πράξη αν κάτι είναι μεγαλύτερο από 2047, κάνε το 2047 και αν κάτι είναι μικρότερο από -2048, κάνε το -2048. Η πράξη αυτή του if μπορεί να πραγματοποιηθεί με τη χρήση δύο αφαιρέσεων και δύο προσθέσεων.

Οι αριθμοί που θα πρέπει να γίνουνε saturate έχουνε μέγεθος 16bit. Αρχικά αυτοί οι αριθμοί φορτώνονται σε SSE καταχωρητές. Κάνοντας μία αφαίρεση του εκάστοτε

αριθμού με τον αριθμό 30720 (30720 = 32512-2048) και ταυτόχρονα saturate, θα έχω το αποτέλεσμα ενός αριθμού ο οποίος έχει γίνει saturate ως προς το αρνητικό του κομμάτι, δηλαδή οτιδήποτε είναι μικρότερο του -2048 γίνεται saturate και γίνεται 2048. Κάνοντας δύο διαδοχικές προσθέσεις προκύπτει ένας αριθμός που έχει γίνει saturate πλέον ως προς το θετικό του μέρος, δηλαδή οτιδήποτε μεγαλύτερο του 2047 θα γίνει ίσο με 2047 μετά την αφαίρεση του αριθμού που προστέθηκε. Ας δούμε ένα παράδειγμα, έστω ο αριθμός 22532.

Βήμα 1: saturate(22523 – 30464) = saturate(-8197) = -2048

Βήμα 2: saturate(-2048 + 30464+ 30464) = 32512

Βήμα 3: 32512 – 30464 = 2048

Μήπως υπάρχει καλύτερη λύση; Η απάντηση είναι θετική. Όπως προαναφέραμε με τον παραπάνω τρόπο η πράξη του saturation γίνεται για όλα τα αποκωδικοποιούμενα block και κατ' επέκταση για όλα τα pixel. Αν κι εδώ εφαρμόζαμε αυτό που σκεφτήκαμε και για την κλήση της FAST_IDCT, δηλαδή να καλέσουμε τη συνάρτηση μόνο για τα pixel που είναι μη μηδενικά. Αυτό θα μπορούσαμε να τι κάνουμε μέσα από τις δύο συναρτήσεις **Decode_MPEG2_Intra_Block** και **Decode_MPEG2_Non_Intra_Block** ώστε να γλιτώσουμε τις «τσάμπα» πράξεις για τα pixel που έχουνε μηδενική τιμή. Παρακάτω παρουσιάζεται η τελική μορφή των δύο αυτών συναρτήσεων:

```
void Decode_MPEG2_Intra_Block(comp,dc_dct_pred)
int comp;
int dc_dct_pred[];
{
  int val, i, j, sign, nc, cc, run;
  unsigned int code;
  DCTtab *tab;
  short *bp;
  int *qmat;
  struct layer_data *ld1;
  int xorval;

  int orval;

  orval = 0;

  /* with data partitioning, data always goes to base layer */
  ld1 = (ld->scalable_mode==SC_DP) ? &base : ld;
  bp = ld1->block[comp];

  if (base.scalable_mode==SC_DP)
    if (base.priority_breakpoint<64)
      ld = &enhan;
    else
      ld = &base;

  cc = (comp<4) ? 0 : (comp&1)+1;

  qmat = (comp<4 || chroma_format==CHROMA420)
         ? ld1->intra_quantizer_matrix
```

```
        : ld1->chroma_intra_quantizer_matrix;

  /* ISO/IEC 13818-2 section 7.2.1: decode DC coefficients */
  if (cc==0)
    val = (dc_dct_pred[0]+= Get_Luma_DC_dct_diff());
  else if (cc==1)
    val = (dc_dct_pred[1]+= Get_Chroma_DC_dct_diff());
  else
    val = (dc_dct_pred[2]+= Get_Chroma_DC_dct_diff());

  if (Fault_Flag) return;

  bp[0] = val << (3-intra_dc_precision);
  xorval = bp[0];

  nc=0;

#ifdef TRACE
  if (Trace_Flag)
    printf("DCT(%d)i:",comp);
#endif /* TRACE */

  /* decode AC coefficients */
  for (i=1; ; i++)
  {
    code = Show_Bits(16);
    if (code>=16384 && !intra_vlc_format)
      tab = &DCTtabnext[(code>>12)-4];
    else if (code>=1024)
    {
      if (intra_vlc_format)
        tab = &DCTtab0a[(code>>8)-4];
      else
        tab = &DCTtab0[(code>>8)-4];
    }
    else if (code>=512)
    {
      if (intra_vlc_format)
        tab = &DCTtab1a[(code>>6)-8];
      else
        tab = &DCTtab1[(code>>6)-8];
    }
    else if (code>=256)
      tab = &DCTtab2[(code>>4)-16];
    else if (code>=128)
      tab = &DCTtab3[(code>>3)-16];
    else if (code>=64)
      tab = &DCTtab4[(code>>2)-16];
    else if (code>=32)
      tab = &DCTtab5[(code>>1)-16];
    else if (code>=16)
      tab = &DCTtab6[code-16];
    else
    {
      if (!Quiet_Flag)
        printf("invalid Huffman code in Decode_MPEG2_Intra_Block()\n");
      Fault_Flag = 1;
      return;
    }

    Flush_Buffer(tab->len);

#ifdef TRACE
    if (Trace_Flag)
    {
      printf(" (");
      Print_Bits(code,16,tab->len);
    }
#endif /* TRACE */


    if (tab->run==64) /* end_of_block */
    {
#ifdef TRACE
      if (Trace_Flag)
        printf("): EOB\n");
#endif /* TRACE */
```

Για την δημιουργία της πράξης miss match (βλέπε παρατήρηση 2)

```
          bp[63] ^= (xorval&1)^1;

          Fast_IDCT(bp,orval);
        return;
      }

      if (tab->run==65) /* escape */
      {
#ifdef TRACE
        if (Trace_Flag)
        {
          putchar(' ');
          Print_Bits(Show_Bits(6),6,6);
        }
#endif /* TRACE */

        i+= run = Get_Bits(6);

#ifdef TRACE
        if (Trace_Flag)
        {
          putchar(' ');
          Print_Bits(Show_Bits(12),12,12);
        }
#endif /* TRACE */

        val = Get_Bits(12);
        if ((val&2047)==0)
        {
          if (!Quiet_Flag)
            printf("invalid escape in Decode_MPEG2_Intra_Block()\n");
          Fault_Flag = 1;
          return;
        }
        if((sign = (val>=2048)))
          val = 4096 - val;
      }
      else
      {
        i+= run = tab->run;
        val = tab->level;
        sign = Get_Bits(1);

#ifdef TRACE
        if (Trace_Flag)
          printf("%d",sign);
#endif /* TRACE */
      }

      if (i>=64)
      {
        if (!Quiet_Flag)
          fprintf(stderr,"DCT coeff index (i) out of bounds (intra2)\n");
        Fault_Flag = 1;
        return;
      }

#ifdef TRACE
      if (Trace_Flag)
        printf("): %d/%d",run,sign ? -val : val);
#endif /* TRACE */

      j = scan[ld1->alternate_scan][i];
      val = (val * ld1->quantizer_scale * qmat[j]) >> 4;
        //gadem
        //transpose by using another table
        j = scan2[ld1->alternate_scan][i];
       //bp[j] = sign ? -val : val;
        //val = sign ? -val : val;
        val = (val^(-sign))+sign;
        if(val>2047)
                val=2047;
        else if (val < -2048)
                val=-2048;
        bp[j] = val;
        xorval ^= val;
        orval |= ((j>>3)|(j&7));
```

Αντικατάσταση της έκφρασης `val = (val^(-sign))+sign` με ισοδύναμή της αποφεύγοντας το `if` (βλέπε παρατήρηση 2)

Η συνάρτηση του saturation γίνεται μόνο όταν οι τιμές των pixel είναι μη μηδενικές.

Για την δημιουργία της πράξης miss match (βλέπε παρατήρηση 1)

```
        nc++;

        if (base.scalable_mode==SC_DP && nc==base.priority_breakpoint-63)
               ld = &enhan;

        }
}
```

Παρατηρήσεις

1. Από τον παραπάνω κώδικα βλέπω ότι πλέον τα if καλούνται μόνο στις περιπτώσεις που οι τιμές των pixel είναι μη μηδενικές καθώς ο κώδικας ακολουθεί το VLD με αποτέλεσμα τα pixel που είναι μηδενικά να μην εκτελούνε την πράξη του saturate. Το πρόβλημα πλέον εστιάζεται στο γεγονός ότι κατά τη διάρκεια του saturation και στο τέλος της συνάρτησης εκτελείται μια πράξη που κατά την ορολογία του αλγορίθμου H262 ονομάζεται mismatch control. Κατά το πρότυπο του H262 mismatch control σημαίνει ότι όταν το άθροισμα των saturated pixel έχει σαν τελευταίο bit 0, τότε θα πρέπει το τελευταίο pixel του block να γίνει 0 αν είναι 1 και 1 αν είναι 0, δηλαδή περιγράφεται από την πράξη:

```
if ((sum&1)==0)
    Block_Ptr[63]^= 1;
```

Έτσι και στην πράξη miss match που θα υλοποιήσω μέσα στην παραπάνω συνάρτηση θα πρέπει να υπολογίσω το αρχικά το άθροισμα των saturated τιμών των pixel. Παρατηρώ ωστόσο, ότι στη συνάρτηση αυτή μου είναι χρήσιμο μόνο το τελευταίο bit του αθροίσματος. Γιατί να κάνω πρόσθεση που κοστίζει παραπάνω από μια δυαδική πράξη όταν με τη δυαδική πράξη μπορώ να πάρω το ίδιο αποτέλεσμα; Μπορώ με δυαδική πράξη να πάρω το ζητούμενο;

Εμείς χρειαζόμαστε να προσομοιώσουμε την πράξη της πρόσθεσης για το τελευταίο bit. Αυτό μπορεί να γίνει με τη δυαδική πράξη xor.

Input		Output
A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

Table 3.8 XOR

Παρατηρώ από τον πίνακα αληθείας ότι η πράξη της xor προσομοιώνει την πράξη της πρόσθεσης δύο δυαδικών αριθμών ως προς το τελευταίο bit. Αν από τον παραπάνω πίνακα πάρω τις τιμές input, είτε κάνω την πράξη της πρόσθεσης είτε την πράξη xor θα πάρω το ίδιο αποτέλεσμα.

Η πράξη **sum&1 = xorval&1** όπως εξηγήσαμε παραπάνω, έτσι θα πρέπει ουσιαστικά να κατασκευάσω την πράξη

```
if ((xorval&1)==0)
    Block_Ptr[63]^= 1;
```

με τέτοιο τρόπο ώστε να αποφύγω την πράξη if που κοστίζει αρκετούς κύκλους μηχανής αν φανταστούμε ότι είναι πιθανό να κάνει λάθος πρόβλεψη διακλάδωσης.

Η πράξη **Block_Ptr [63] ^= (xorval&1)^1** προσωμοιώνει επαρκώς το ζητούμενο, καθώς έχω (xorval&1)^1 που είναι ουσιαστικά η πράξη Block_Ptr[63]^= 1.

Αν το αποτέλεσμα (xorval&1)^1 = 0, τότε xorval&1 = 1 και είναι γνωστό ότι number^0 =number, άρα και εδώ έχω:

```
if ((xorval&1)==1)
    Block_Ptr[63]^= 0; ->Block_Ptr[63]= 0;
```

Αν το αποτέλεσμα (xorval&1)^1 = 1, τότε xorval&1 = 0 άρα θα έχω:

```
if ((xorval&1)==0)
    Block_Ptr[63]^= 1;
```

2. Αναλύοντας τις δύο συναρτήσεις **Decode_MPEG2_Intra_Block** και **Decode_MPEG2_Non_Intra_Block** οι οποίες περιέχουν την πράξη **val = sign ? -val : val** η οποία είναι ουσιαστικά ένα if branch, μήπως γίνεται να το αποφύγω όπως και στην περίπτωση με το miss match control;

Η πράξη **val = sign ? -val : val** στην πραγματικότητα είναι η πράξη:

```
If (sign == 1)
    val = -val;
else val = val;
```

Οι πιθανότητες για λάθος πρόβλεψη διακλάδωσης σε αυτή την περίπτωση είναι αυξημένες καθώς αντιλαμβανόμαστε ότι οι πιθανότητες το val να είναι είτε θετικό είτε αρνητικό είναι οι ίδιες σε κάθε περίπτωση.

Η λύση που προτείνω είναι να αντικατασταθεί η πράξη **val = sign ? -val : val** με την ισοδύναμή της **val = (val^(-sign)) + sign.**

Εξήγηση:

Αν sign = 1, τότε –sign = -1 , το οποίο σε δυαδική μορφή είναι ο αριθμός 1111111111111111. Αν κάνουμε τον αριθμό αυτό xor με val τότε το παραγόμενο αποτέλεσμα είναι το συμπλήρωμα ως προς 2 του val. Τέλος, προσθέτοντας το val που είναι 1 θα έχω τελικά val = -val.

Αν sign = 0, τότε –sign = 0 , το οποίο σε δυαδική μορφή είναι ο αριθμός 0000000000000000. Αν κάνουμε τον αριθμό αυτό xor με val τότε το παραγόμενο αποτέλεσμα είναι ο ίδιος ο αριθμός val. Τέλος, προσθέτοντας το val που είναι 0 θα έχω τελικά val = val.

3.3.5. Miss match control

Στην παράγραφο 3.3.4 μιλήσαμε για τη λειτουργία miss match control κατά την οποία σε ορισμένα block στην τιμή του τελευταίου pixel θα τοποθετηθεί τιμή ίση με 1. Όπως είδαμε όμως στην παράγραφο 3.3.1.4 αρκετά από τα block έχουνε αρκετές μηδενικές τιμές εκτός των άνω αριστερά pixel, με αποτέλεσμα να οδηγηθούμε στη λύση εναλλακτικών μορφών idct, τα idct1x1, idct2x2 και idct4x4 τα οποία μας οδηγούν σε μια συνολικά “φθηνότερη” FAST_IDCT. Η χρήση ωστόσο της λειτουργίας του miss match είναι αντιληπτό ότι δημιουργεί block όπως φαίνονται στο παρακάτω σχήμα.

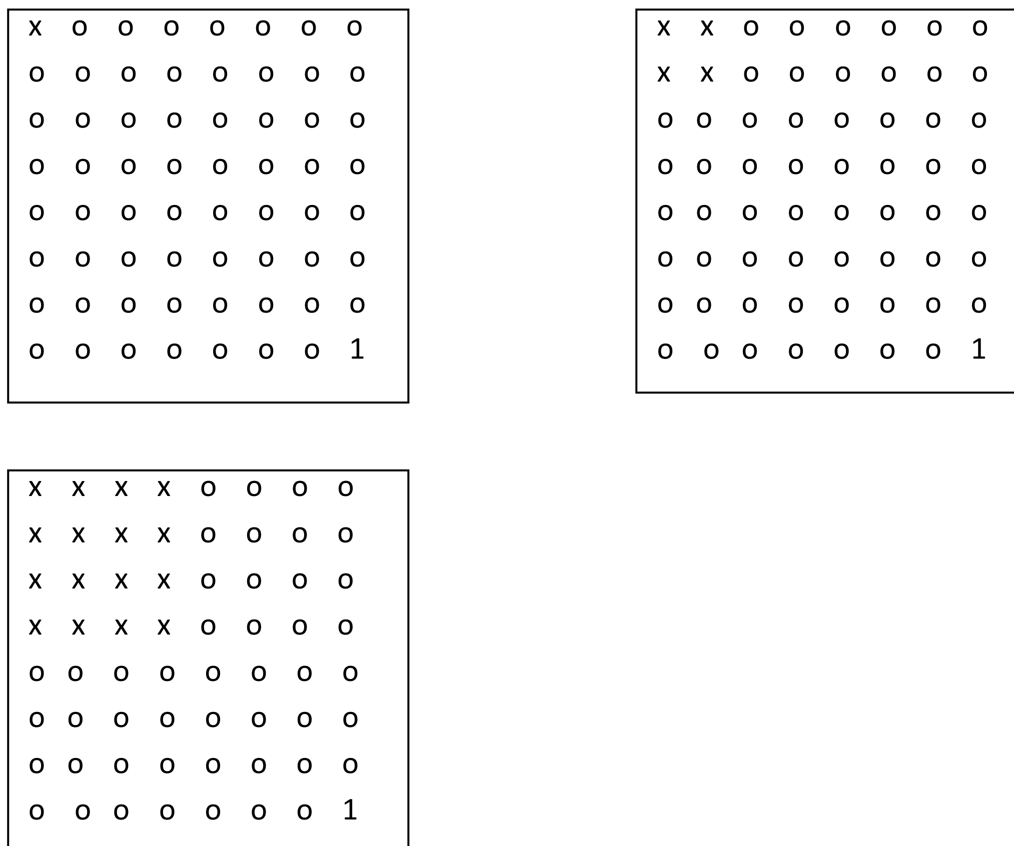


Image 3.4 Struct of block 1x1_M, 2x2_M, 3x3_M

Σύμφωνα με τους ελέγχους για εκτέλεση του FAST_IDCT τα παραπάνω block θα εκτελέσουν τον κλασσικό idct και όχι μια από τις παραλλαγές idct1x1, idct2x2 ή idct4x4 αν και είναι πολύ κοντά στην υλοποίησή τους.

Με βάση τα παραπάνω δεδομένα οδηγηθήκαμε στη δημιουργία άλλων τριών συναρτήσεων, των idct1x1_M, idct2x2_M και idct4x4_M. Οι συναρτήσεις αυτές είναι παρεμφερείς με τις συγγενικές τους idct1x1, idct2x2 και idct4x4 αντίστοιχα, με τη διαφορά του τελευταίου pixel άρα και της τελευταίας γραμμής που πλέον κατά τη διάρκεια της idct_row δεν είναι μηδενισμένα, παρά έχουνε υποστεί idct καθώς το pixel που βρίσκονται στη θέση 63 του κάθε block έχουνε πλέον την τιμή 1. Επίσης είναι αρκετά πρωτοφανές ότι λόγω της τιμής 1 και του ότι κατά την idct_row έχω

εφαρμογή της στα διπλανά pixel μιας γραμμής, η τελευταία αυτή γραμμή θα περιέχει συγκεκριμένες τιμές για κάθε pixel της με αποτέλεσμα να μπορούμε να φορτώσουμε αυτές τις τιμές σε ένα πίνακα και από εκεί να τις χρησιμοποιούμε παρά να εκτελούμε την idct_row για την τελευταία γραμμή . Ο πίνακας αυτός έχει τις εξής τιμές: 2, -6, 9, -11, 11, -9, 6, -2 οι οποίες είναι οι τιμές που προκύπτουν στη γραμμή 8 του κάθε block για blk[63]=1.

__Idct_1x1_M__

Παρατίθεται ο κώδικας idct1x1_M τόσο σε γλώσσα C όσο και με χρήση intrinsic, ο οποίος και είναι ο βελτιωμένος και ταχύτερος.

Idct1x1_M σε γλώσσα C

```
void idct1x1_M(short *block){

int x0, x2, x4, x5;
int i;


block[0]=block[1]=block[2]=block[3]=block[4]=block[5]=block[6]=block[7]=block[0]<<3;

memcpy(block+56, idct_table, 16);


for(i=0; i<8; i++)
{
        x0 = (block[8*0+i]<<8) + 8192;
        x5 = block[8*7+i];


        x4 = (W7*x5 + 4)>>3;
        x5 = (4 - W1*x5)>>3;

        x2 = (181*(x4+x5)+128)>>8;
        x4 = (181*(x4-x5)+128)>>8;

        block[8*0+i] = block[8*2+i] = iclp[(x0+x4)>>14];
        block[8*1+i] = iclp[(x0+x2)>>14];
        block[8*3+i] = iclp[(x0+x5)>>14];
        block[8*4+i] = iclp[(x0-x5)>>14];
        block[8*5+i] = block[8*7+i] = iclp[(x0-x4)>>14];
        block[8*6+i] = iclp[(x0-x2)>>14];
}


}
```

Είναι προφανές και από την εισαγωγή σε αυτή την παράγραφο ότι ο πίνακας idct_table περιέχει τις τιμές 2, -6, 9, -11, 11, -9, 6, -2, που είναι η idct_row της τελευταίας γραμμής.

Idct1x1_M σε γλώσσα intrinsic

```
static void idct_1x1_M(blk)
short *blk;
{

        __m128i mm1,mm7,mm11,mm17;
        __m128i mm0,mm10,mm6,mm16;
        __m128i a0,a1,a2,a4,tmp1;


        __m128i mmzero,mm128,mm181,mm8192,mmw7,mmfour;

        int a;



        mmzero = _mm_setzero_si128();                                //0,…0
        tmp1 = _mm_load_si128((__m128i*)(constants_32+0*4));         //1,…1
        mm128 = _mm_load_si128((__m128i*)(constants_32+2*4));        //128,…128
        mm181 = _mm_load_si128((__m128i*)(constants_32+3*4));        //181…181
        mm8192 = _mm_load_si128((__m128i*)(constants_32+4*4));       //8192,…8192
        mmfour = _mm_load_si128((__m128i*)(constants_32+1*4));       //4,…4
        mmw7 = _mm_load_si128((__m128i*)(constants+12*8));           //W7,… W7


        a = blk[0]<<3;

        mm0 = _mm_set1_epi16(a);
        mm7 = _mm_load_si128((__m128i*)(constants+15*8));


        //load x0 and convert to 32 bit
        mm10 = _mm_unpacklo_epi16 (mm0,mm0);
        mm10 = _mm_madd_epi16 (mm10, tmp1);
        mm10 = _mm_slli_epi32 (mm10, 8);
        mm10 = _mm_add_epi32 (mm10,mm8192);
        mm0 = _mm_unpackhi_epi16 (mm0,mm0);
        mm0 = _mm_madd_epi16 (mm0, tmp1);
        mm0 = _mm_slli_epi32 (mm0, 8);
        mm0 = _mm_add_epi32 (mm0,mm8192);

        //x5 = mm7


        //  x4 = (W7*x5 + 4) >>3
        mm11 = _mm_unpacklo_epi16 (mm7, mmzero);
        mm11 = _mm_madd_epi16 (mm11, mmw7);                          //W7*x5
        mm11 = _mm_add_epi32 (mm11, mmfour);                         //W7*x5 + 4
        mm11 = _mm_srai_epi32(mm11,3);
        mm1 = _mm_unpackhi_epi16 (mm7, mmzero);
        mm1 = _mm_madd_epi16 (mm1, mmw7);                            //W7*x5
        mm1 = _mm_add_epi32 (mm1, mmfour);                           //W7*x5 + 4
        mm1 = _mm_srai_epi32(mm1,3);


        //x5 = (4 - W1*x5)>>3;
        a2 =_mm_load_si128((__m128i*)(constants+2*8));               //w1,…w1
        mm17 = _mm_unpacklo_epi16 (mm7, mmzero);
        mm17 = _mm_madd_epi16 (mm17, a2);                            //W1*x5
        mm17 = _mm_sub_epi32(mmfour,mm17);
        mm17 = _mm_srai_epi32(mm17,3);
        mm7 = _mm_unpackhi_epi16 (mm7, mmzero);
        mm7 = _mm_madd_epi16 (mm7, a2);                              //W1*x5
        mm7 = _mm_sub_epi32(mmfour,mm7);
        mm7 = _mm_srai_epi32(mm7,3);



        //x2 = (181*(x4+x5)+128)>>8;
        a2 = _mm_add_epi32(mm11,mm17);                               //x4+x5
        a4 = _mm_srai_epi32(a2,16);
        a4 = _mm_madd_epi16(a4,mm181);
        a0 = _mm_mulhi_epu16(a2,mm181);
        a2 = _mm_mullo_epi16(a2,mm181);
        a0 = _mm_slli_epi32(a0,16);
        a2 = _mm_or_si128(a2,a0);
        a4 = _mm_slli_epi32(a4,16);
        a4 = _mm_add_epi32(a2,a4);
```

```
a4 = _mm_add_epi32(a4,mm128);
mm16 = _mm_srai_epi32(a4,8);


a2 = _mm_add_epi32(mm1,mm7);
a4 = _mm_srai_epi32(a2,16);
a4 = _mm_madd_epi16(a4,mm181);
a0 = _mm_mulhi_epu16(a2,mm181);
a2 = _mm_mullo_epi16(a2,mm181);
a0 = _mm_slli_epi32(a0,16);
a2 = _mm_or_si128(a2,a0);
a4 = _mm_slli_epi32(a4,16);
a4 = _mm_add_epi32(a2,a4);
a4 = _mm_add_epi32(a4,mm128);
mm6 = _mm_srai_epi32(a4,8);


//x4 = (181*(x4-x5)+128)>>8
a2 = _mm_sub_epi32(mm11,mm17);                                      //x4-x5
a4 = _mm_srai_epi32(a2,16);
a4 = _mm_madd_epi16(a4,mm181);
a0 = _mm_mulhi_epu16(a2,mm181);
a2 = _mm_mullo_epi16(a2,mm181);
a0 = _mm_slli_epi32(a0,16);
a2 = _mm_or_si128(a2,a0);
a4 = _mm_slli_epi32(a4,16);
a4 = _mm_add_epi32(a2,a4);
a4 = _mm_add_epi32(a4,mm128);
mm11 = _mm_srai_epi32(a4,8);

a2 = _mm_sub_epi32(mm1,mm7);
a4 = _mm_srai_epi32(a2,16);
a4 = _mm_madd_epi16(a4,mm181);
a0 = _mm_mulhi_epu16(a2,mm181);
a2 = _mm_mullo_epi16(a2,mm181);
a0 = _mm_slli_epi32(a0,16);
a2 = _mm_or_si128(a2,a0);
a4 = _mm_slli_epi32(a4,16);
a4 = _mm_add_epi32(a2,a4);
a4 = _mm_add_epi32(a4,mm128);
mm1 = _mm_srai_epi32(a4,8);



//blk[8*1] = iclp[(x0+x2)>>14];
tmp1 = _mm_add_epi32 (mm10, mm16);                          //x0+x2 lo
tmp1 = _mm_srai_epi32(tmp1 ,14);
a1 = _mm_add_epi32 (mm0, mm6);                              //x0+x2 hi
a1 = _mm_srai_epi32(a1 ,14);
a4 = _mm_packs_epi32 (tmp1 ,a1);
_mm_store_si128 ((__m128i*)(blk+8) , a4);                   //store



//blk[8*0] = blk[8*2] = iclp[(x0+x4)>>14];
a4 = _mm_add_epi32 (mm10, mm11);                            //x0+x4 lo
a4 = _mm_srai_epi32(a4 ,14);
a2 = _mm_add_epi32 (mm0, mm1);                              //x0+x4 hi
a2 = _mm_srai_epi32(a2 ,14);
a4 = _mm_packs_epi32 (a4 ,a2);                              //a4 packs-
_mm_store_si128 ((__m128i*)(blk) , a4);                     //store
_mm_store_si128 ((__m128i*)(blk+16) , a4);                  //store




// blk[8*3] = iclp[(x0+x5)>>14];
a4 = _mm_add_epi32 (mm10, mm17);                            //x0+x5 lo
a4 = _mm_srai_epi32(a4 ,14);
a2 = _mm_add_epi32 (mm0, mm7);                              //x0+x5 hi
a2 = _mm_srai_epi32(a2 ,14);
a4 = _mm_packs_epi32 (a4 ,a2);
_mm_store_si128 ((__m128i*)(blk+24) , a4);
```

```
        // blk[8*4] = iclp[(x0-x5)>>14];
        a4 = _mm_sub_epi32 (mm10, mm17);                              //x0-x5 lo
        a4 = _mm_srai_epi32(a4 ,14);
        a2 = _mm_sub_epi32 (mm0, mm7);                                //x0-x5 hi
        a2 = _mm_srai_epi32(a2 ,14);
        a4 = _mm_packs_epi32 (a4 ,a2);
        _mm_store_si128 ((__m128i*)(blk+32) , a4);

        //  blk[8*5] = blk[8*7] = iclp[(x0-x4)>>14];
        a4 = _mm_sub_epi32 (mm10, mm11);
        a4 = _mm_srai_epi32(a4 ,14);
        a2 = _mm_sub_epi32 (mm0, mm1);                                //x0+x4 hi
        a2 = _mm_srai_epi32(a2 ,14);
        a4 = _mm_packs_epi32 (a4 ,a2);
        _mm_store_si128 ((__m128i*)(blk+40) , a4);
        _mm_store_si128 ((__m128i*)(blk+56) , a4);

        // blk[8*6] = iclp[(x0-x2)>>14];
        a4 = _mm_sub_epi32 (mm10, mm16);
        a4 = _mm_srai_epi32(a4 ,14);
        a2 = _mm_sub_epi32 (mm0, mm6);                                //x0-x2 hi
        a2 = _mm_srai_epi32(a2 ,14);
        a4 = _mm_packs_epi32 (a4 ,a2);
        _mm_store_si128 ((__m128i*)(blk+48) , a4);



        //store
        _mm_store_si128 ((__m128i*)(blk) , mm0);
        _mm_store_si128 ((__m128i*)(blk+8) , mm0);
        _mm_store_si128 ((__m128i*)(blk+16) , mm0);
        _mm_store_si128 ((__m128i*)(blk+24) , mm0);
        _mm_store_si128 ((__m128i*)(blk+32) , mm0);
        _mm_store_si128 ((__m128i*)(blk+40) , mm0);
        _mm_storeu_si128 ((__m128i*)(blk+48) , mm0);
        _mm_store_si128 ((__m128i*)(blk+56) , mm0);

}
```

Η υλοποίηση των idct_2x2_M και idct_4x4_M είναι παρόμοια με τη συνάρτηση idct_1x1 και είναι πλεόνασμα να την παραθέσω.

3.5 Απόδοση των αλλαγών 3.1-3.4

Έχοντας κάνει τις αλλαγές όπως περιγράφονται στις παραπάνω συναρτήσεις και έχοντας βεβαιωθεί ότι το παραγόμενο αρχείο είναι ίδιο με το αρχικό που εξάγεται από τον reference code θα πρέπει να γίνει μια μέτρηση για να παρατηρηθεί η απόδοση των παραπάνω όχι μόνο για να εξαχθεί το συμπέρασμα του κατά πόσο γρηγορότερη είναι η αποκωδικοποίηση, αλλά και γιατί είναι λογικό να έχουνε αλλάξει οι ισορροπίες στο κόστος των συναρτήσεων. Έτσι, κάποιες συναρτήσεις θα αλλάξουνε θέση στο πόσο δαπανηρές είναι, και άλλες που είχανε ένα ποσοστό περίπου στο 1%, λόγω της μείωσης του χρόνου πλέον το ποσοστό τους θα αλλάξει και θα πρέπει ίσως να τις βελτιστοποιήσουμε εφόσον κάτι τέτοιο είναι δυνατό.

Dantes.m2v

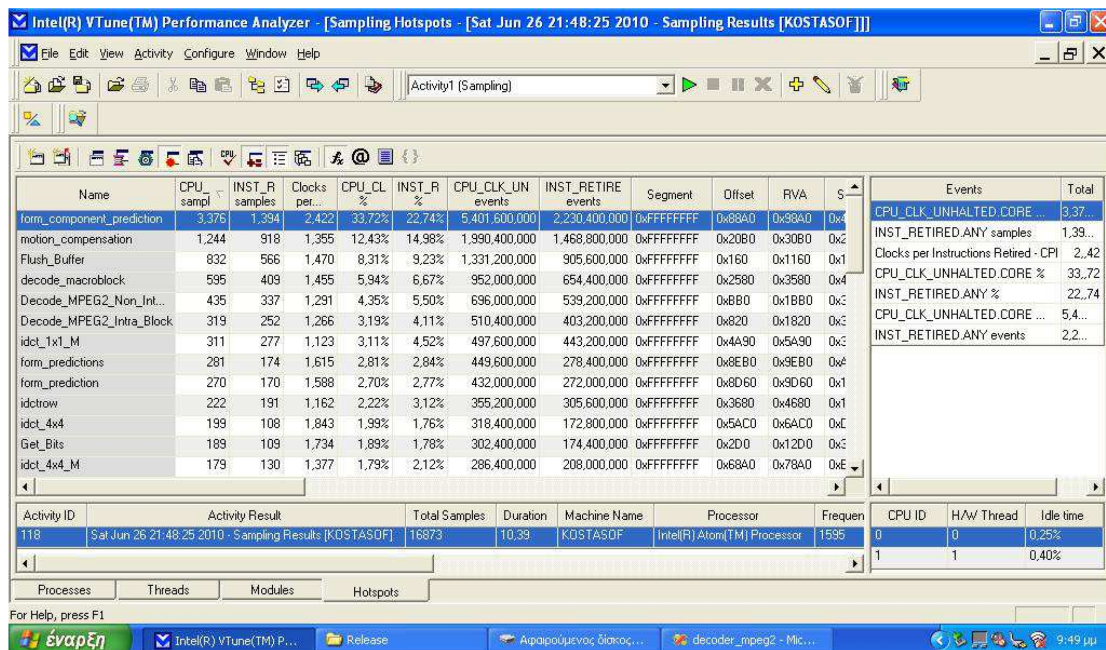


Image 3.5 Screenshot of Vtune while executing new MPEG-2

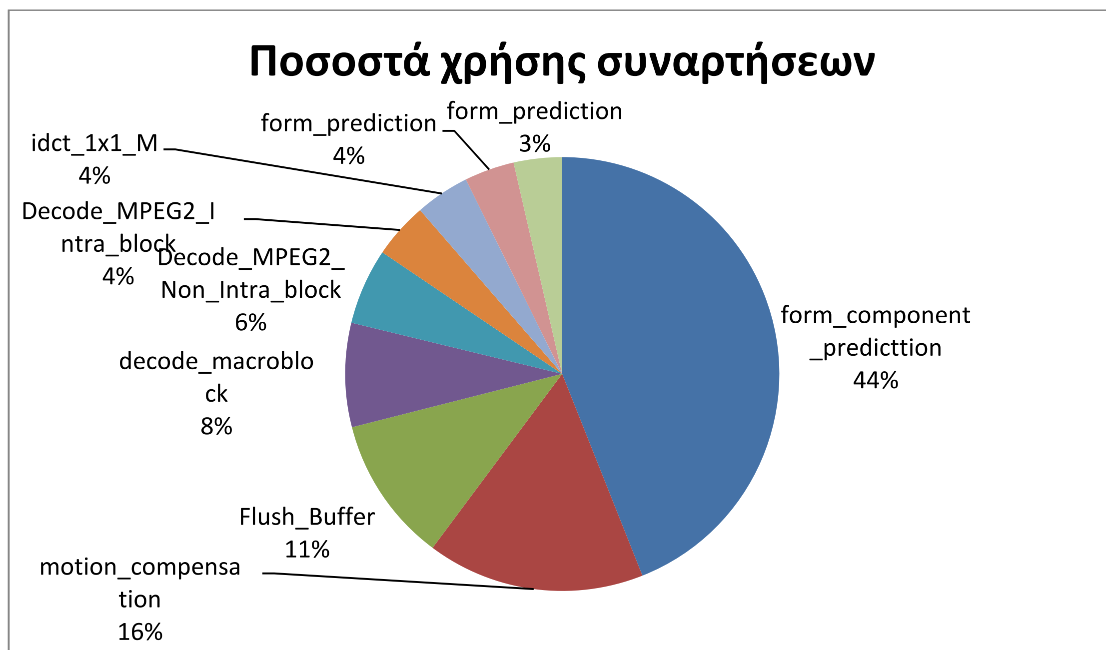


Fig 3.3

Από την παραπάνω πίτα παρατηρείται το φαινόμενο συναρτήσεις που παίζανε πρωταγωνιστικό ρόλο στην κατανάλωση ισχύος κατά την εκτέλεση του reference code να έχουνε χάσει τη θέση τους και άλλες συναρτήσεις που βρισκόντουσαν

χαμηλότερα σε θέση να έχουνε ανέλθει σε ποσοστό χρόνου εκτέλεσης του νέου κώδικα, όπως η Flush_Buffer που πλέον βρίσκεται στην 3η θέση.

Ας δούμε αναλυτικότερα τη βελτίωση κάθε συνάρτησης ξεχωριστά.

Fast_idct

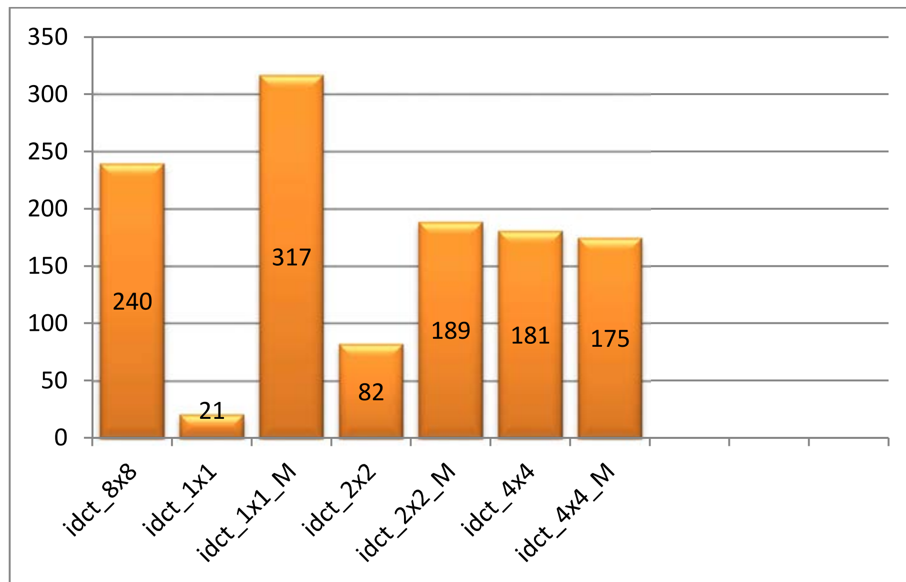


Fig 3.4 Επιμέρους συναρτήσεις Idct οι οποίες συνθέτουν την καινούρια idct.

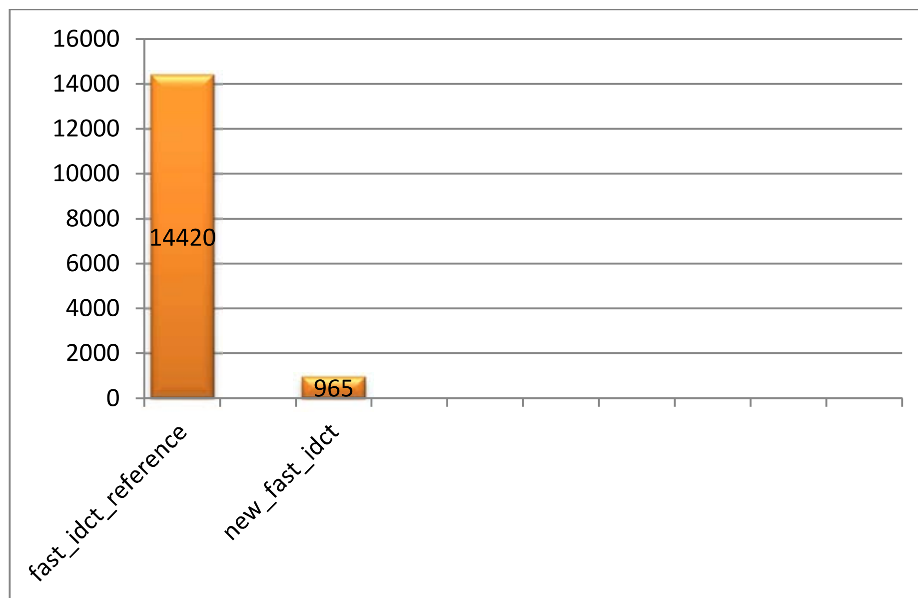


Fig 3.5 Σύγκριση παλαιάς και νέας idct

Από το παραπάνω διάγραμμα φαίνεται ότι πλέον συνολικά η idct έχει μειωθεί κατά 93%, αλλά αυτό ίσως οφείλεται και στο χαμηλό bitrate του dantes.m2v με αποτέλεσμα πάρα πολλά από τα pixel να έχουνε μηδενικές τιμές.

Form_component_prediction

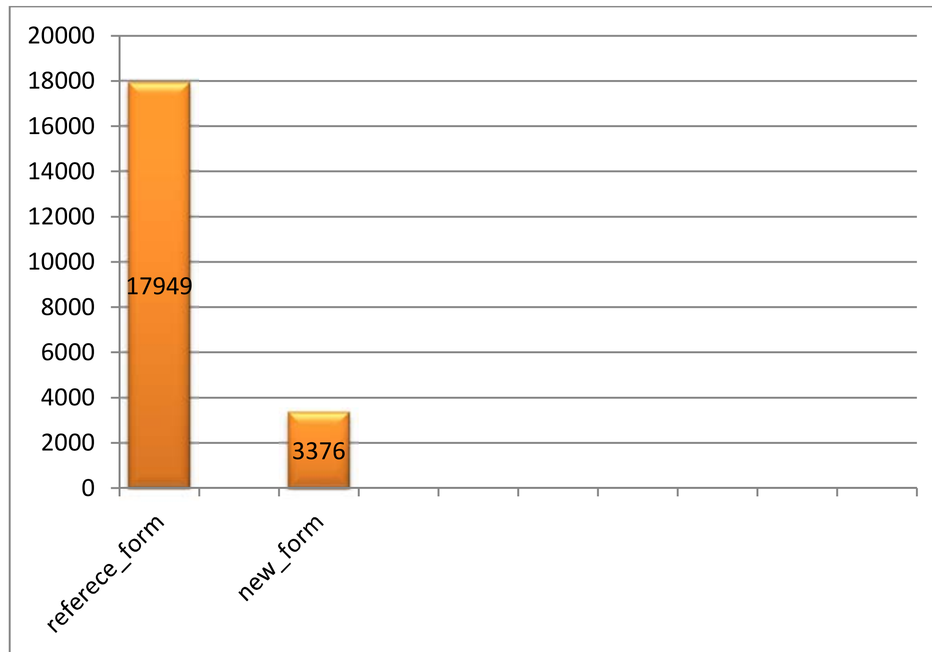


Fig 3.6 Σύγκριση παλαιάς και νέας form_component_prediction

Add_Block

Η συνάρτηση add_block έχει γίνει inline με αποτέλεσμα να μη φαίνεται στο Vtune.

Decode_Mpeg2_Non_Intra και Decode_Mpeg2_Intra

Οι δύο αυτές συναρτήσεις είναι οι μοναδικές που αυξάνονται ελάχιστα κάτι όμως απολύτως λογικό καθώς ενσωματώνουν τη λειτουργία της saturate και λειτουργίες όπως επιλογή του τύπου fast_idct και inverse_transpose.

<u>Συνολική βελτίωση του dantes.m2v</u>

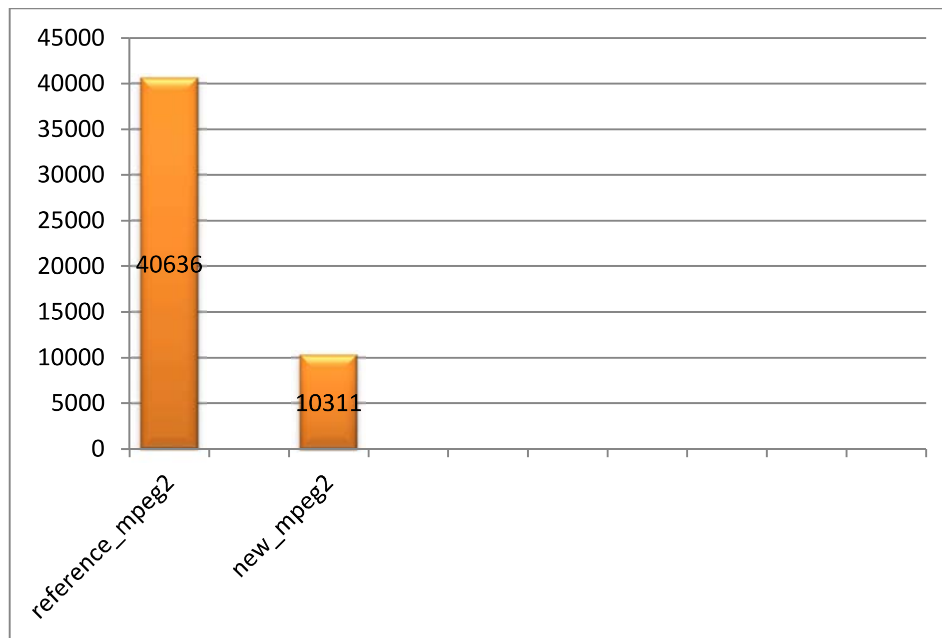


Fig 3.7 Σύγκριση παλαιάς και νέας συνάρτησης MPEG-2

Από το παραπάνω διάγραμμα προς το παρόν έχουμε μία βελτίωση στον mpeg2 κατά 74%, δηλαδή εκτελείται με ρυθμό 3.94 φορές πιο γρήγορα.

Input_Inter.m2v

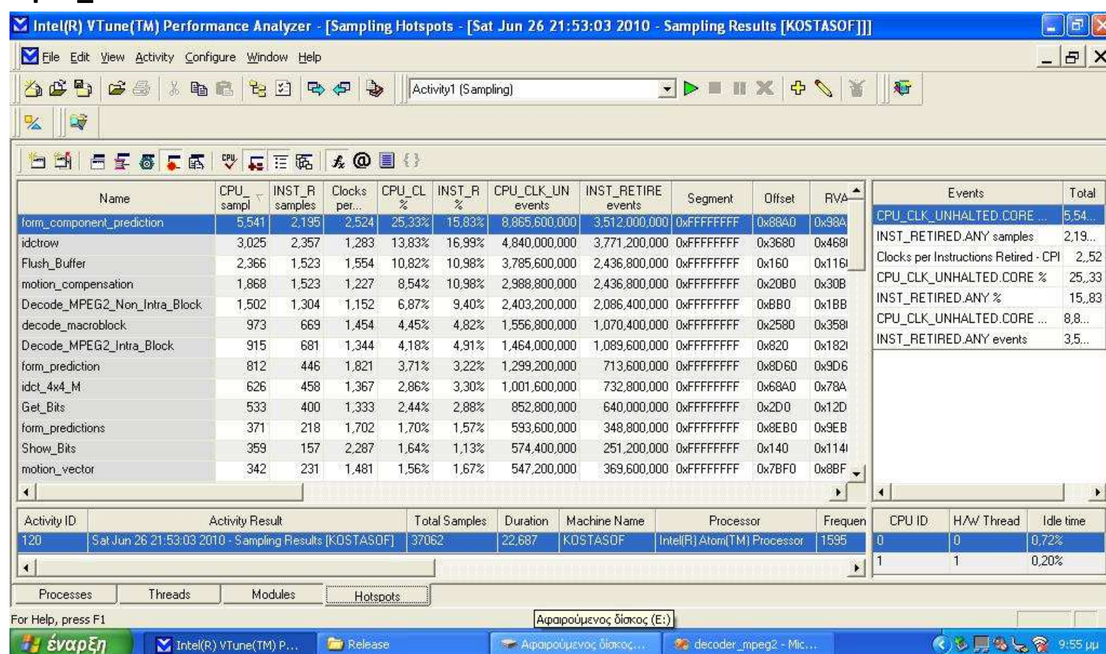


Image 3.6 Screenshot of Vtune while executing new MPEG-2

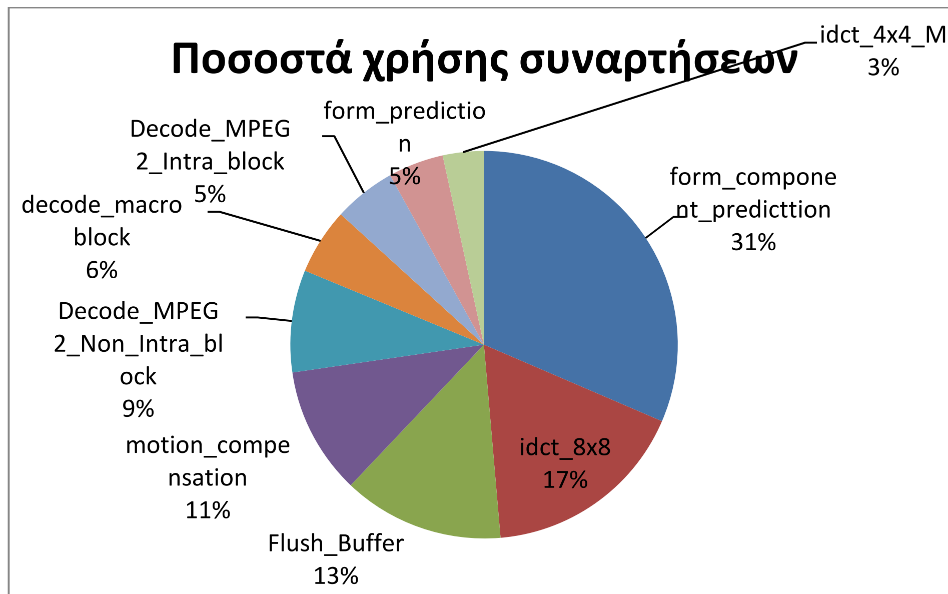


Fig 3.8

Από το παραπάνω γράφημα γίνεται αντιληπτό ότι σε αυτό το video έχουμε διαφορετικά δεδομένα, καθώς παρατηρούμε τη συνάρτηση idct_8x8 να έχει αναρριχηθεί στη 2η θέση κόστους, κάτι απολύτως λογικό καθώς πλέον το video έχει bitrate 4Mbps με αποτέλεσμα τα pixel που θα έχουνε μηδενικές τιμές να είναι μειωμένα σε σχέση με την περίπτωση του dantes.m2v το οποίο και έχει bitrate 2Mbps.

Ας δούμε όμως αναλυτικότερα τις συναρτήσεις μία προς μία.

Fast_idct

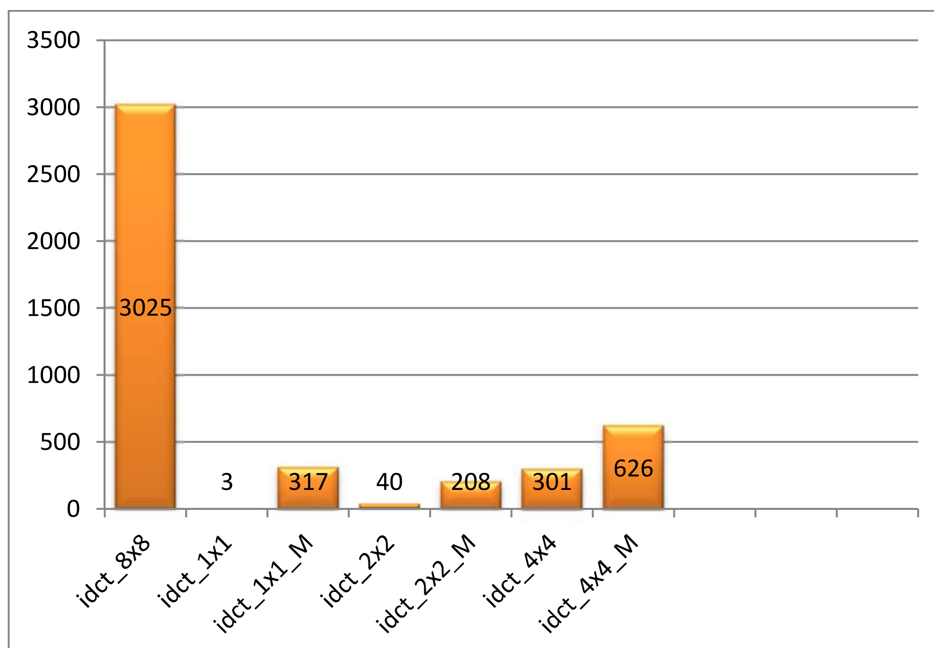


Fig 3.9 Επιμέρους συναρτήσεις Idct οι οποίες συνθέτουν την καινούρια idct.

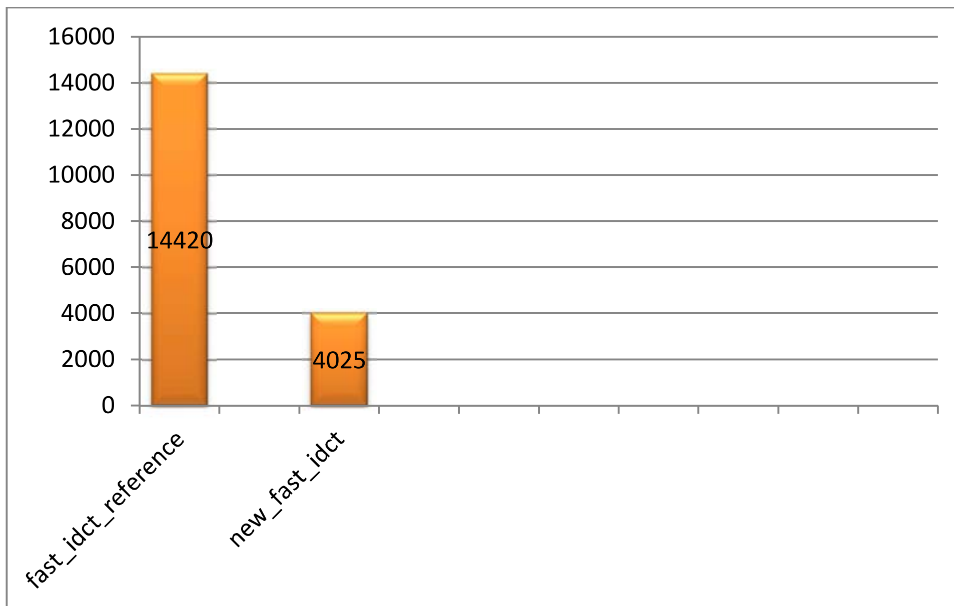


Fig 3.10 Σύγκριση παλαιάς και νέας συνάρτησης idct

Από το παραπάνω γράφημα παρατηρώ βελτίωση του idct του reference code κατά 83,6% , δηλαδή τρέχει το νέο idct σε ταχύτητα 6,1 φορές γρηγορότερα.

Form component prediction

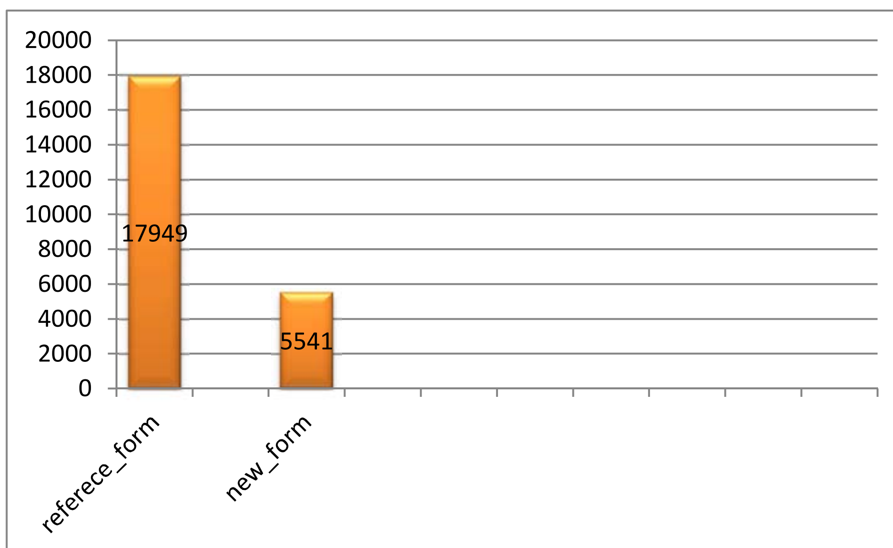


Fig 3.11 Σύγκριση παλαιάς και νέας συνάρτησης form_component_prediction

Η συνάρτηση της form_component_prediction έχει βελτιωθεί κατά 70%, δηλαδή τρέχει 3,23 φορές ταχύτερα από τον προκάτοχό της.

Add_Block

Η συνάρτηση add_block έχει γίνει inline με αποτέλεσμα να μη φαίνεται στο Vtune.

Συνολική βελτίωση του Input_Inter.m2v

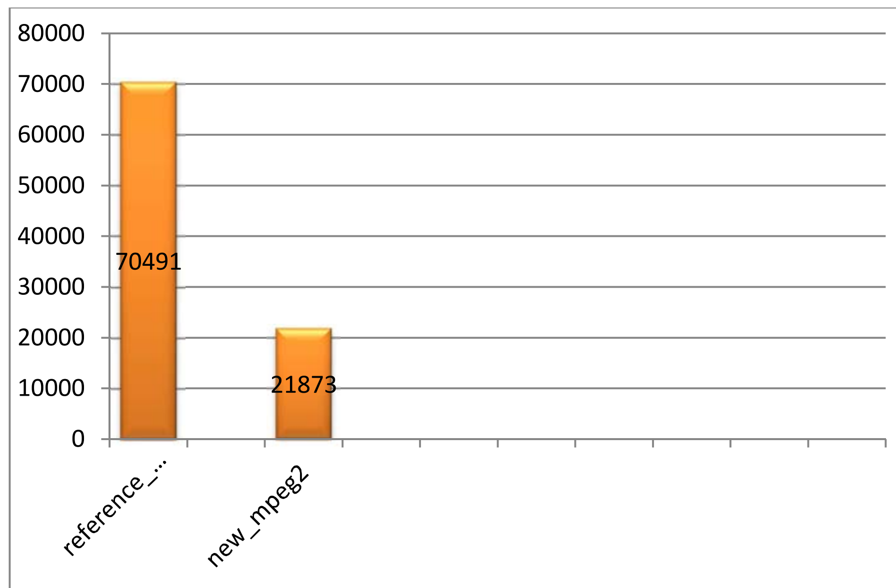


Fig 3.12 Σύγκριση του reference code με το νέο MPEG-2

Από το παραπάνω διάγραμμα προς το παρόν έχουμε μία βελτίωση στον mpeg2 κατά 69%, δηλαδή εκτελείται με ρυθμό 3,22 φορές πιο γρήγορα.

Συμπέρασμα μέτρησης

Γενικά είδαμε ότι με τις παραπάνω αλλαγές βελτιώσαμε αρκετά τόσο τις συναρτήσεις μία προς μία, αλλά αυτό είχε και άμεσο αντίκτυπο στο να τρέχουμε τον κώδικα περισσότερο από 3 φορές πιο γρήγορα, ακόμα και για Bitrate 4Mbps.

Τα αποτελέσματα για το video dantes.m2v είναι παραπάνω από αισιόδοξα καθώς πριν καν ολοκληρώσουμε τη δουλειά μας στη βελτιστοποίηση πετύχαμε πάνω από 3,5x. Αντίθετα, ίσως περιμέναμε περισσότερα για το video Input_Inter.m2v, καθώς το υψηλό Bitrate δε μας βοήθησε να αξιοποιήσουμε στο έπακρο τις δυνατότητες των συναρτήσεων idct_1x1,idct_2x2 και idct4x4, καθώς και τις παραλλαγές τους για mismatch=1.

Σύμφωνα με τις μετρήσεις θα ήτανε φρόνιμο να βελτιώσουμε συναρτήσεις όπως η flush_buffer ώστε να ρίξουμε ακόμα περισσότερο το χρόνο αποκωδικοποίησης και

να φτάσουμε στο τελικό αποτέλεσμα που ελπίζουμε να είναι πάνω από 3,6x ακόμα και για video σαν το Input_Inter.m2vμ δηλαδή video με υψηλό Bitrate.

3.6 Flush_Buffer

Παρατηρώντας τη συνάρτηση Flush_Buffer διαπιστώνει κανείς ότι πάρα πολλοί κύκλοι μηχανής χάνονται στις προσβάσεις μνήμης και κυρίως για load κάποιων τιμών(layer data). Μετά από αυτή την παρατήρηση είναι εύκολο να διαπιστώσει κανείς ότι γίνονται πολλά load στην ίδια συνάρτηση χωρίς να έχει αλλάξει η τιμή στη μνήμη του layer data ακόμα και όταν αυτό γίνεται σε πολλά επίπεδα χωρίς αλλαγή. Η σκέψη είναι, μήπως εάν χρησιμοποιούσαμε μια global μεταβλητή και αποθηκεύαμε σε αυτή την τιμή του ld->Incnt, που χρησιμοποιείται πάρα πολύ, θα είχαμε καλύτερη απόδοση στη συνάρτηση;

Αρχικά έγινε ορισμός 4 μεταβλητών στη global.h

```
EXTERN int G_Incnt;
EXTERN unsigned char *G_Rdptr;
EXTERN unsigned int G_Bfr;

EXTERN unsigned char *G_Rdmax;
```

Στη συνέχεια κατά τη διάρκεια της αρχικοποίησης του Buffer θα πρέπει να δώσουμε τιμές σε αυτές τις μεταβλητές και από εκεί και πέρα να χρησιμοποιούμε αυτές τις μεταβλητές για να γλιτώσουμε τις προσβάσεις μνήμης.

```
void Initialize_Buffer()
{
  G_Incnt = ld->Incnt = 0;
  G_Rdptr = ld->Rdbfr + 2048;
  G_Rdmax = G_Rdptr;

#ifdef VERIFY
  /*  only the verifier uses this particular bit counter
   *  Bitcnt keeps track of the current parser position with respect
   *  to the video elementary stream being decoded, regardless
   *  of whether or not it is wrapped within a systems layer stream
   */
  ld->Bitcnt = 0;
#endif

  G_Bfr = 0;
  Flush_Buffer(0); /* fills valid data into bfr */
}
```

```
void Flush_Buffer32()
{
  int Incnt;
```

```
  G_Bfr = 0;

  Incnt = G_Incnt;
  Incnt -= 32;

  if (System_Stream_Flag && (G_Rdptr >= G_Rdmax-4))
  {
    while (Incnt <= 24)
    {
      if (G_Rdptr >= G_Rdmax)
        Next_Packet();
      G_Bfr |= Get_Byte() << (24 - Incnt);
      Incnt += 8;
    }
  }
  else
  {
    while (Incnt <= 24)
    {
      if (G_Rdptr >= ld->Rdbfr+2048)
        Fill_Buffer();
      G_Bfr |= *G_Rdptr++ << (24 - Incnt);
      Incnt += 8;
    }
  }
  G_Incnt = ld->Incnt = Incnt;

#ifdef VERIFY
  ld->Bitcnt += 32;
#endif /* VERIFY */
}


void Flush_Buffer(N)
int N;
{
  int Incnt;

  G_Bfr <<= N;

  Incnt = G_Incnt -= N;

  if (Incnt <= 24)
  {
    if (System_Stream_Flag && (G_Rdptr >= G_Rdmax-4))
    {
      do
      {
        if (G_Rdptr >= G_Rdmax)
          Next_Packet();
        G_Bfr |= Get_Byte() << (24 - Incnt);
        Incnt += 8;
      }
      while (Incnt <= 24);
    }
    else if (G_Rdptr < ld->Rdbfr+2044)
    {
      do
      {
        G_Bfr |= *G_Rdptr++ << (24 - Incnt);
        Incnt += 8;
      }
      while (Incnt <= 24);
    }
    else
    {
      do
      {
        if (G_Rdptr >= ld->Rdbfr+2048)
          Fill_Buffer();
        G_Bfr |= *G_Rdptr++ << (24 - Incnt);
        Incnt += 8;
      }
      while (Incnt <= 24);
    }
```

```
    G_Incnt = Incnt;
  }

#ifdef VERIFY
  ld->Bitcnt += N;
#endif /* VERIFY */

}



void next_start_code()
{
  /* byte align */
  Flush_Buffer(G_Incnt&7);
  while (Show_Bits(24)!=0x01L)
    Flush_Buffer(8);
}
```

3.7 Clear_block

Η συνάρτηση Clear_block σαν λειτουργία έχει την εξής απλή διαδικασία, να μηδενίζει τις τιμές των block ώστε να μπορούνε να χρησιμοποιηθούνε ξανά, όντας πλέον άδεια. Η συνάρτηση αυτή αν και απλή, καταλήγει να είναι σχετικά δαπανηρή λόγω της συχνότητας με την οποία καλείται και έτσι θα πρέπει να βρούμε ένα τρόπο να τη βελτιώσουμε.

Η αρχική σκέψη είναι να κάνουμε χρήση της memset και να θέσουμε την τιμή 0 σε όλο το block. Ωστόσο, ο χρόνος που απαιτούνταν μετά τη χρήση της ήτανε πανομοιότυπος με το χρόνο πριν και έτσι μου γεννήθηκε η απορία μας πως γίνεται αυτό, αφού εγώ φανταζόμουνα ότι απλά με κάποιον πάρα πολύ γρήγορο τρόπο θα μηδενίσει όλα τα κελιά της μνήμης. Κοιτώντας όμως το disassemply της εντολής με έκπληξη ανακάλυψα ότι η συνάρτηση είναι ένα απλό for loop που το μόνο που κάνει είναι να προχωρά μία-μία θέση στη μνήμη και να τη μηδενίζει, δηλαδή κάτι σαν τον αρχικό κώδικα.

Η δεύτερη σκέψη είναι να γίνει ο μηδενισμός με χρήση Intrinsic SSE2. Η λύση αυτή αποδείχθηκε αρκετά πιο γρήγορη και έγινε απλά με μια ανάθεση στη μεταβλητή mm0, η τιμή 0 και από εκεί και ύστερα απλά γίνεται αποθήκευση της mm0 στην Block_Ptr.

```
/* IMPLEMENTATION: set scratch pad macroblock to zer____o */
__forceinline static void Clear_Block(comp)
int comp;
{
short *Block_Ptr;
  int i;
```

```
  __m128i mm0;

  Block_Ptr = ld->block[comp];

  mm0 = _mm_setzero_si128();

  _mm_store_si128 ((__m128i*)(Block_Ptr) , mm0);
  _mm_store_si128 ((__m128i*)(Block_Ptr+8) , mm0);
  _mm_store_si128 ((__m128i*)(Block_Ptr+16) , mm0);
  _mm_store_si128 ((__m128i*)(Block_Ptr+24) , mm0);
  _mm_store_si128 ((__m128i*)(Block_Ptr+32) , mm0);
  _mm_store_si128 ((__m128i*)(Block_Ptr+40) , mm0);
  _mm_store_si128 ((__m128i*)(Block_Ptr+48) , mm0);
  _mm_store_si128 ((__m128i*)(Block_Ptr+56) , mm0);

  /*for (i=0; i<64; i++)
    *Block_Ptr++ = 0;*/
}
```

3.8 Τελική μέτρηση απόδοσης - Συμπεράσματα

Έχοντας φτάσει πλέον στο τέλος αυτής εδώ της διπλωματικής εργασίας θα πρέπει να κάνουμε τις τελικές μετρήσεις του νέου κώδικα που έχουμε γράψει ώστε τελικά να διαπιστώσουμε πόσο ακριβώς είναι το performance και επίσης να γίνει δοκιμή στο HD video όπως είχαμε υποσχεθεί στην αρχή αυτού του κεφαλαίου και να διαπιστώσουμε εάν καταφέραμε με την προσπάθειά μας να πετύχουμε αποκωδικοποίηση HD video στον επεξεργαστή Atom.

3.8.1 dantes.m2v - Input_Inter.m2v - Vin-4-p.m2v(HD)

Εδώ πλέον δεν μας ενδιαφέρουν τόσο τα ποσοστά χρήσης επεξεργαστή κάθε συνάρτησης, αλλά το τελικό performance, δηλαδή πιο πρακτικά πόσα fps έτρεχε ο reference MPEG-2 πριν βάλουμε το χέρι μας και τον πειράξουμε και πόσο τώρα; Κάνοντας διαίρεση τα τελικά fps που αποκωδικοποιούμε με τα αρχικά θα πάρουμε το πόσες φορές πιο γρήγορος είναι ο νέος κώδικας.

Ο παρακάτω πίνακας παρουσιάζει τη νέα κατάσταση όσον αφορά τα παραγόμενα fps.

	Fps atom	Fps core2duo(2,4GHz)
Dantes.m2v	40	127
Input_inter.m2v	36,4	111
Vin.m2v	9,47	29,44

Table 3.1 reference code

	Fps atom	Fps core2duo(2,4GHz)
Dantes.m2v	175	700
Input_inter.m2v	129	492
Vin.m2v	37,2	124

Table 3.9 Βελτιωμένος κώδικας

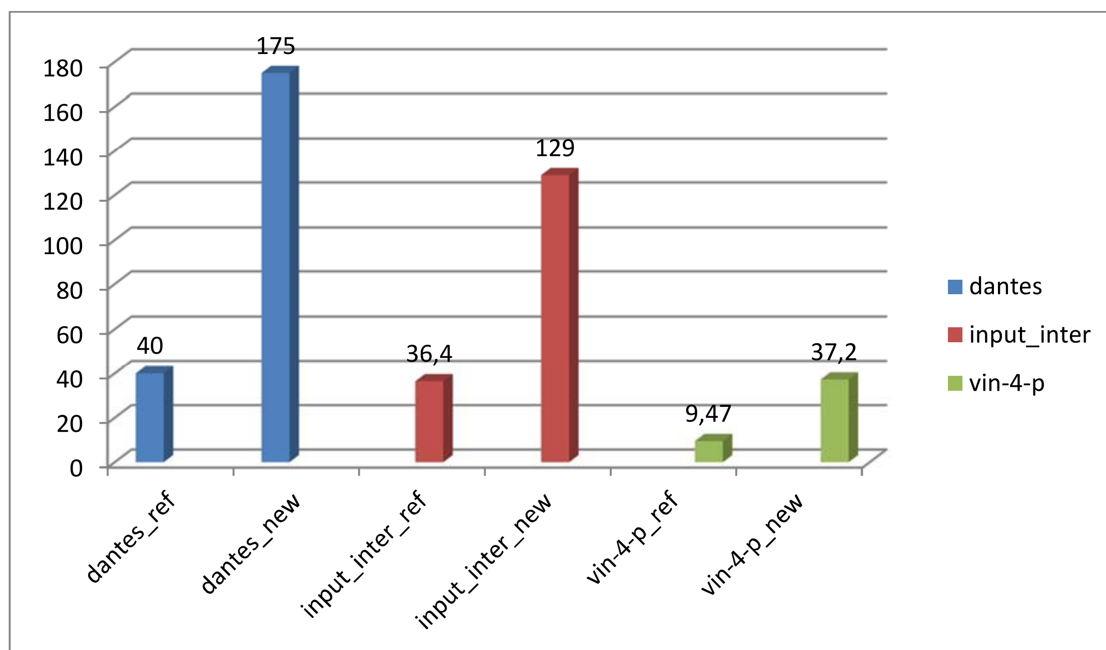


Fig 3.13 Fps of reference code against fps of improved code

Από το παραπάνω γράφημα αλλά και από τον παρακάτω πίνακα φαίνονται οι τελικές αποδόσεις.

	Fps atom	Fps core2duo(2,4GHz)
Dantes.m2v	175/40 = 4,375x	700/127 = 5,5x
Input_inter.m2v	129/36,4 = 3,54x	492/111 = 4,43x
Vin.m2v	37,2/9,47 = 3,92x	124/29,44 = 4,38x

Table3.10

4.Συμπεράσματα

Η διπλωματική αυτή εργασία οδήγησε στην εξαγωγή πολλών συμπερασμάτων τα οποία και θα αναλύσουμε παρακάτω.

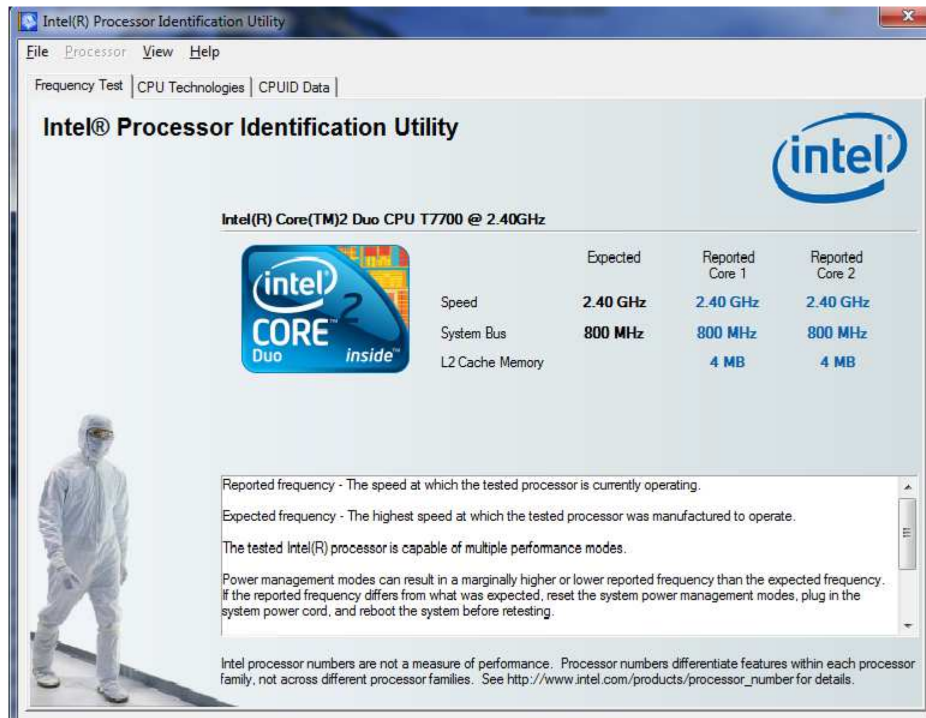
HD decoding

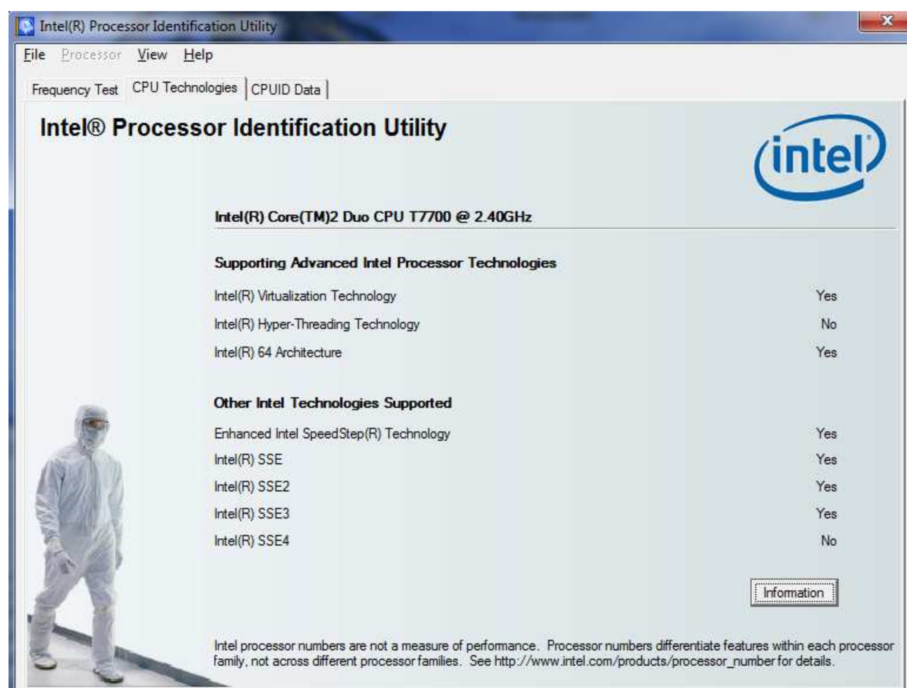
Ίσως η πιο ευχάριστη είδηση σε αυτή τη διπλωματική εργασία είναι η real time αποκωδικοποίηση video HD ανάλυσης 1440x1080. Σίγουρα είναι έκπληξη ένας επεξεργαστής με κόστος 10€ να καταφέρνει με μία software αποκωδικοποίηση να παίζει video με HD ανάλυση real time. Σίγουρα τα netbook με επεξεργαστές Atom δεν έχουνε φτιαχτεί για αυτό το σκοπό και για αυτό το λόγο η οθόνη τους δεν υποστηρίζει αυτή την ανάλυση. Άλλωστε τα netbook δεν έχουνε καν HD έξοδο για σύνδεσή τους σε εξωτερική οθόνη ήTV υψηλής ευκρίνειας.
Πλέον με την τεκμηρίωση ότι ναι ο επεξεργαστής Atom μπορεί να παίξει video υψηλής ευκρίνειας ανοίγει ο δρόμος για τις εταιρείες κατασκευής netbook να λανσάρουν στην αγορά netbook με δυνατότητες απεικόνισης HD, αλλά και με κάποια έξοδο HDMI ώστε να μπορούνε να χρησιμοποιηθούνε και ως media center σε ένα οικιακό σύστημα ψυχαγωγίας.

Atom vs Core2duo

Από τις αρχικές και τελικές μετρήσεις διαπιστώνει κανείς ότι ο Intel Core2Duo στα 2,4GHz είναι τουλάχιστον 4 φορές γρηγορότερος από τον επεξεργαστή Atom στα 1,6GHz παρότι μιλάμε για επεξεργαστές ίδιας οικογένειας αν και ο ένας είναι 64bit και ο άλλος είναι 32 bit.
Τα 2,4GHz του Core2Duo δικαιολογούν μια διαφορά της τάξης του 1,5x και σε καμία περίπτωση τη χαοτική διαφορά του 4+x. Ας δούμε όμως με το utility της Intel το Intel Processor Identification τα χαρακτηριστικά του κάθε επεξεργαστή.

Intel Core2Duo





Intel(R) Processor Identification Utility

File Processor View Help

Frequency Test | CPU Technologies | CPUID Data

Intel® Processor Identification Utility

intel

Intel(R) Core(TM)2 Duo CPU T7700 @ 2.40GHz

Processor classification	
CPU Type	0
CPU Family	6
CPU Model	F
CPU Stepping	B
CPU Revision	B6

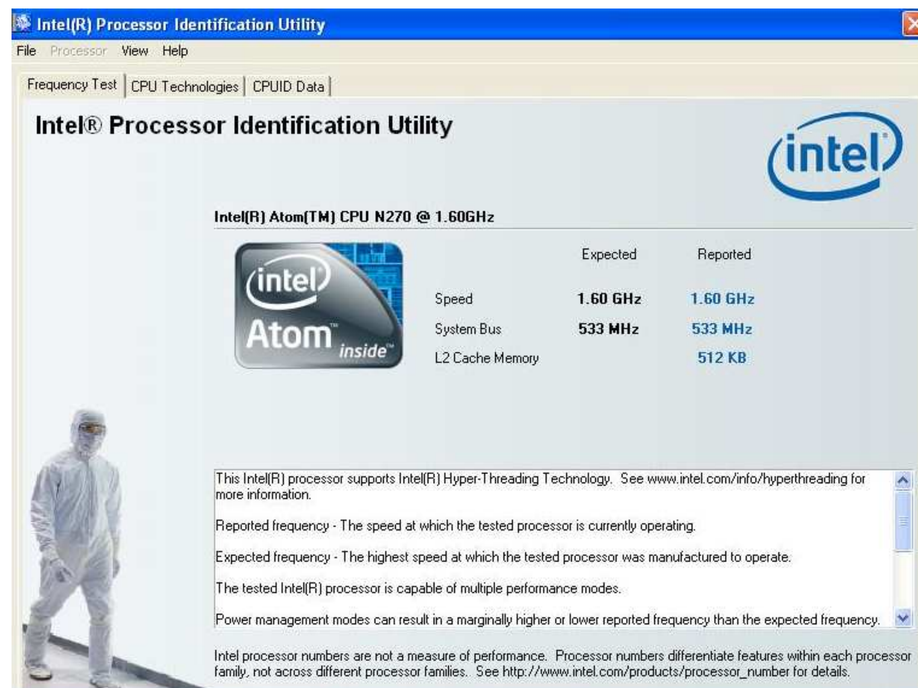
Processor details	
Level 2 cache	4 MB
Level 1 data cache	2 x 32 KB
Level 1 instruction cache	2 x 32 KB
Packaging	µFCPGA/µFCBGA

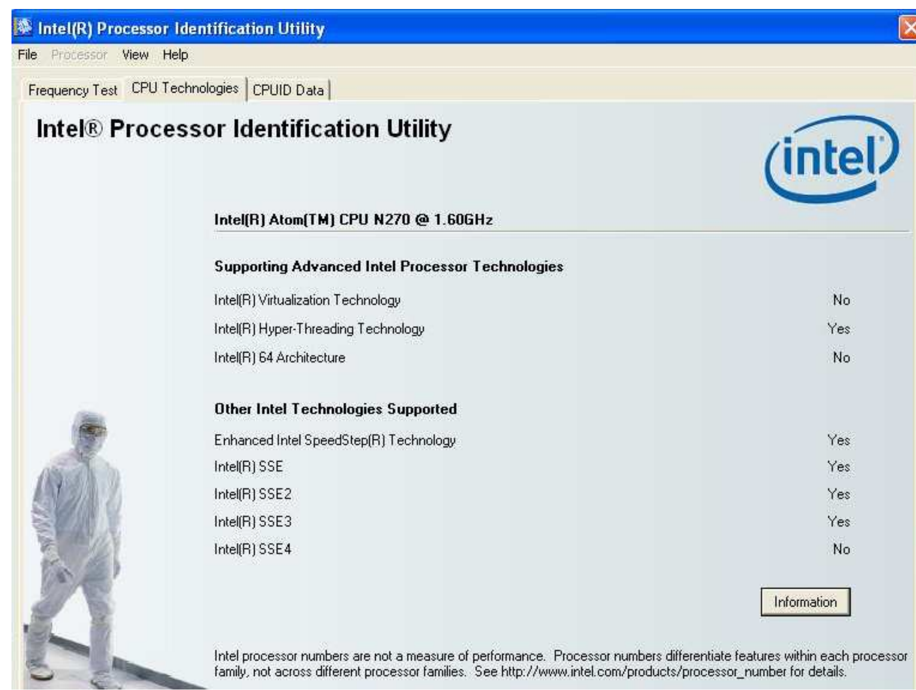
Other Intel processor features	
Execute Disable Bit	Yes
Enhanced Halt State	No

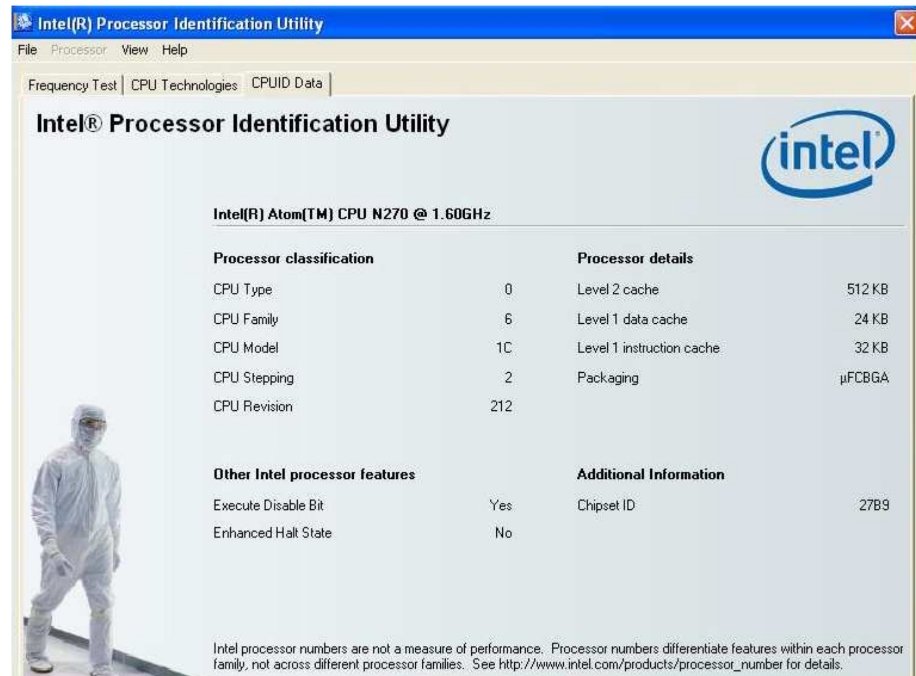
Additional Information	
Chipset ID	2815

Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families. See http://www.intel.com/products/processor_number for details.

Intel Atom




Intel(R) Processor Identification Utility
File Processor View Help
Frequency Test CPU Technologies CPUID Data
Intel® Processor Identification Utility
intel
Intel(R) Atom(TM) CPU N270 @ 1.60GHz
Supporting Advanced Intel Processor Technologies
Intel(R) Virtualization Technology No
Intel(R) Hyper-Threading Technology Yes
Intel(R) 64 Architecture No
Other Intel Technologies Supported
Enhanced Intel SpeedStep(R) Technology Yes
Intel(R) SSE Yes
Intel(R) SSE2 Yes
Intel(R) SSE3 Yes
Intel(R) SSE4 No
Information
Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families. See http://www.intel.com/products/processor_number for details.


Intel(R) Processor Identification Utility
File Processor View Help
Frequency Test CPU Technologies CPUID Data
Intel® Processor Identification Utility
intel
Intel(R) Atom(TM) CPU N270 @ 1.60GHz
Processor classification
CPU Type 0
CPU Family 6
CPU Model 1C
CPU Stepping 2
CPU Revision 212
Processor details
Level 2 cache 512 KB
Level 1 data cache 24 KB
Level 1 instruction cache 32 KB
Packaging µFCBGA
Other Intel processor features
Execute Disable Bit Yes
Enhanced Halt State No
Additional Information
Chipset ID 27B9
Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families. See http://www.intel.com/products/processor_number for details.

Από τα παραπάνω screenshots προκύπτει ο παρακάτω πίνακας που συνοψίζει τις διαφορές των δύο επεξεργαστών οι οποίοι είναι και οι δύο για laptop.

	Atom	Core2Duo
Frequency GHz	1,6	2,4
Cache level 2	512KB	4MB
Cache level 1 data	24	2x32
Cache level 1 instruction	32	2x32
FSB (MHz)	533	800

Το μυστικό της τόσο μεγάλης διαφοράς των δύο επεξεργαστών στην αποκωδικοποίηση video είναι η μεγαλύτερη κατά 8 φορές cache του Core2Duo, που είναι 4MB σε αντίθεση με τα μόλις 512KB του Atom. Επίσης η cache level 1, δηλαδή αυτή που είναι ακόμα πιο κοντά στον επεξεργαστή είναι διπλάσια στην περίπτωση του Core2Duo.

Αυτό που συμβαίνει με τον Core2Duo είναι ότι όχι μόνο είναι 4 φορές πιο γρήγορος στον reference code, αλλά ακόμα και στην περίπτωση του βελτιωμένου MPEG-2 το κέρδος είναι μεγαλύτερο σε σχέση με το κέρδος που έχουμε με τον επεξεργαστή Atom. Πως ερμηνεύεται αυτό;

Αρκεί να σκεφτούμε ότι με τα intrinsic φορτώνω κάθε φορά ολόκληρα block ή και ολόκληρα macroblock και η πιθανότητα να έχουμε cache miss είναι αυξημένη λόγω των περισσότερων δεδομένων που φορτώνονται. Έτσι είναι λογικό να βλέπουμε τον Core2Duo να έχει καλύτερο κέρδος ακόμη και στον βελτιωμένο κώδικα. Επίσης ακόμα και αν έχουμε cache miss στην περίπτωση του Core2Duo ο δίαυλος επικοινωνίας μνήμης είναι χρονισμένος στα 800MHz ενώ του Atom είναι στα 533MHz, κάτι που αυξάνει τη διαφορά τους.

Καλύτερος atom

Όπως είδαμε στην προηγούμενη παράγραφο ο επεξεργαστής Atom χάνει σε απόδοση σε σχέση με τον επεξεργαστή Core2Duo λόγω της αρκετά μικρότερης cache. Τι θα γινότανε αν αντί για 512KB μπορούσαμε να έχουμε στον Atom 4MB cache; Αυτή η μετατροπή δεν θα κόστιζε καθόλου σε κατανάλωση μπαταρίας ενώ θα εκτόξευε τις HD επιδόσεις.

Παράρτημα

1.http://en.wikipedia.org/wiki/VTune

2.http://www.it.uom.gr/project/MultimediaTechnologyNotes/chap2b_5.htm

3.http://www.intel.com/itcenter/products/atom/index.htm

4.T-REC-H.262-200002-I!!PDF-E (standard MPEG-2)

5. http://en.wikipedia.org/wiki/Intel_Atom

6. http://www.bbc.co.uk/rd/pubs/papers/paper_14/paper_14.shtml

7. http://www.mpeg.org/MPEG/video/mssg-free-mpeg-software.html (download reference code)

8.http://msdn.microsoft.com/el-gr/default.aspx

9. http://www.it.uom.gr/project/MultimediaTechnologyNotes/chap2b_5.htm