ΠΑΝΕΠΙΣΤΗΜΙΟ
ΘΕΣΣΑΛΙΑΣ
ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ Η/Υ,
ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ ΚΑΙ
ΔΙΚΤΥΩΝ

διπλωματική εργασία

ΥΛΟΠΟΙΗΣΗ ΤΟΥ ΑΘΡΟΙΣΜΑΤΟΣ ΑΠΟΛΥΤΩΝ ΔΙΑΦΟΡΩΝ ΓΙΑ ΤΟ Η.264 ΣΕ ΤΕΧΝΟΛΟΓΙΑ 90 ΝΑΝΟΜΕΤΡΩΝ

Ζαχαρής
Δημήτριος

Βόλος,
Σεπτέμβρης 2006

ευχαριστώ θερμά
τους ανθρώπους που υπέμειναν...

Προσφορές

Ο καθηγητής μου κύριος Γιώργος Σταμούλης πρόσφερε την ανεκτίμητη καθοδήγηση και εμπειρία του.

Ο επιβλέπων μου κύριος Νέστορας Ευμορφόπουλος χάρισε το πνεύμα και τις γνώσεις του στη διάθεσή μου.

«Πάνω από την κεφαλή μου» ευρέθησαν συχνά ο Δημήτρης Καραμπατζάκης και ο Αντώνης Δαδαλιάρης, διδακτορικοί φοιτητές, ώστε να εγγυηθούν την απουσία σφαλμάτων και χάους.

Συμπαράσταση και κοινή πορεία στο χρόνο εργασίας μοιραστήκαμε με το φίλτατο Αποστόλη Γιαννέτσο και τα υπόλοιπα παιδιά στο εργαστήριο.

Τις δύσκολες στιγμές άγχους ακολουθούν εικόνες των φίλων μου και της οικογένειάς μου. Πρόσωπα που γνωρίζουν την έννοια της ανιδιοτέλειας. Σας ευχαριστώ για την ανοχή, την υπομονή και το κουράγιο που δείξατε απέναντι στη συμπεριφορά μου.

Περισσότερο όλων όμως για το χρόνο σας.

Με εκτίμηση,

Δημήτρης Ζαχαρής

Περίληψη

Στο σύγχρονο κόσμο, οι κοινωνίες κατανέμονται σε ανεπτυγμένες, αναπτυσσόμενες και υποανάπτυκτες. Η εργασία που ακολουθεί δεν αφορά καθόλου τις τελευταίες, λίγο τις προηγούμενες και εξαιρετικά πολύ τις πρώτες περιπτώσεις πολιτισμού στη σημερινή κατάσταση του πλανήτη. Κυρίαρχη θέση στον πολιτισμό αυτό κατέχει η εικόνα. Κοινός παράγοντας της κινούμενης δυσδιάστατης εικόνας, όπως αυτή εμφανίζεται στο παρόν, είναι η ψηφιοποίηση, μετάδοση και αποθήκευσή της σε μορφές εξαρτώμενες της τεχνολογίας που είναι διαθέσιμη. Όμως, σε ποιο ακριβώς σημείο πρέπει να εστιάσουμε;

Μικροεπεξεργαστές υψηλής απόδοσης σχεδιάζονται επικεντρωμένοι σε γενικού σκοπού εφαρμογές. Τα σχέδια αυτά όμως αποδίδουν τυπικά για διεργασίες με πλειονότητα εντολών διαχείρισης ελέγχου και όχι εντολών διαχείρισης δεδομένων. Στην τελευταία κατηγορία εφαρμογών ανήκει η κωδικοποίηση και η αποκωδικοποίηση video. Το ζητούμενο είναι να έχουμε απόδοση που να καλύπτει τις ανάγκες ενός έργου πραγματικού χρόνου. Για να εστιάσουμε εκεί ακριβώς, επιλέγουμε να μη χρησιμοποιήσουμε διαμορφούμενους επεξεργαστές με επεκτάσεις συνόλου εντολών που δεν καλύπτουν τις ανάγκες μίας real-time εφαρμογής. Έχοντας σχηματίσει το προφίλ της κωδικοποίησης και της αποκωδικοποίησης βίντεο H.264, γνωρίζουμε πως ο σειριακός χαρακτήρας του Αθροίσματος Απολύτων Διαφορών (ΑΑΔ - SAD) είναι η πιο χρονοβόρα ανάμεσα στις λειτουργίες. Σε αυτή την εργασία ερευνούμε διάφορες υλοποιήσεις της λειτουργίας ΑΑΔ σε τεχνολογία 90 νανομέτρων εκτελώντας σύνθεση για να τις βελτιστοποιήσουμε. Κατόπιν χρησιμοποιούμε την πληροφορία από τη σύγκρισή τους για την τοποθέτηση και τη δρομολόγηση στο τσιπ. Τέλος εξάγουμε τα συμπεράσματά μας και καθοδηγούμε το επόμενο στάδιο μελέτης.

Εισαγωγή

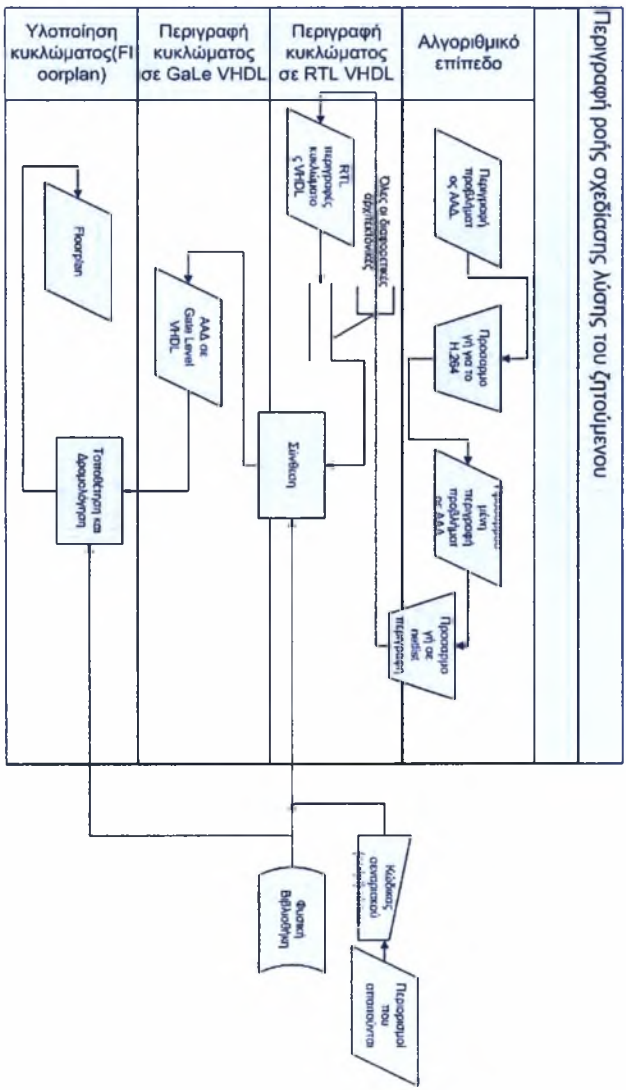
Περιγραφή του προβλήματος.

Διάφορα *standards* έχουν ορισθεί από τη διεθνή κοινότητα επιστημόνων για να τεκμηριώσουν τους αλγορίθμους, τη στρατηγική και τις δομές, ώστε να επιτύχουμε μία επεξεργασία βίντεο όσο το δυνατό κοντύτερα στην απόλυτη ποιότητα, αυτή του ανθρώπινου οφθαλμού. Το τελευταίο και πιο σύγχρονο πρότυπο επεξεργασίας βίντεο είναι το *H.264*. Με την εισαγωγή νέων δομών και αντίστοιχα απαιτήσεων χωρητικότητας και ποιότητας κωδικοποίησης και αποκωδικοποίησης, ο τρόπος αντιμετώπισης του όρου «ποιότητας» έχει αλλάξει άρδην.

Για να μην πολυλογούμε, ποιότητα στο βίντεο συνεπάγεται τη λύση προβλημάτων που εμφανίζονται κατά την επεξεργασία της αρχικής αναλογικής μορφής βίντεο. Ίσως το σημαντικότερο από αυτά είναι η *Εκτίμηση Κίνησης* (Motion Estimation). Κατά την εκτίμηση κίνησης χρειάζεται ένα μέτρο σύγκρισης των διαφόρων πιθανών *διανυσμάτων κίνησης*. Το πιο διαδεδομένο λόγω ισοστάθμισης ποιότητας απόδοσης και ταχύτητας επεξεργασίας είναι το *Άθροισμα των Απολύτων Διαφορών*. Στόχος μας είναι η βέλτιστη υλοποίηση του αλγόριθμου εύρεσης βέλτιστου (ελάχιστου) ΑΑΔ, κατά το H.264, μελετώντας διάφορες αρχιτεκτονικές.

Προσχέδιο λύσης του προβλήματος

Σκοπό έχουμε να μεταφέρουμε αυτό το πολύ κρίσιμο κομμάτι βίντεο, το ΑΑΔ, σε υλικό. Κατά πόσο απαραίτητο είναι αυτό θα αναλυθεί και θα τεκμηριωθεί στο κεφάλαιο βίντεο. Προς το παρόν λέμε «έστω πως ισχύει» το παραπάνω.


Περιγραφή ροής σχεδίασης λύσης του ζητούμενου
Αλγοριθμικό επίπεδο
Περιγραφή προβλήματος σε ΑΛΛ
Προσαρμογή για το H.264
Μελέτη περιγραφή προβλήματος σε ΑΛΛ
Προσαρμογή σε πιστές περιγραφές
Περιγραφή κυκλώματος σε RTL VHDL
Όλες οι διαφορετικές αρχιτεκτονικές
RTL περιγραφές κυκλώματος σε VHDL
Σύνθεση
Περιγραφή κυκλώματος σε GaLe VHDL
ΑΛΛ σε Gate Level VHDL
Υλοποίηση κυκλώματος(Floorplan)
Floorplan
Τοποθέτηση και Δρομολόγηση
Κώδικας σειριακού
Περιφερειακά που απαιτούνται
Φυσική Βιβλιοθήκη

Στο προηγούμενο διάγραμμα ροής περιγράφεται η πορεία προς τη λύση του ζητούμενου προβλήματος (αναλυτικά για το κάθε στάδιο θα μιλήσουμε σε παρακάτω κεφάλαια).

Αρχικά έχουμε τη μαθηματική μορφή του ΑΑΔ:

$$\sum_{i=0}^{15}\sum_{j=0}^{15} |(A_{(x+i,y+j)} - B_{((x+r)+i,(y+s)+j)})|$$

$0 \leq x, y <$ μεγεθος frame
(r, s) motion vector
$A_{(x,y)}$ pixel στη θέση (x, y) του παρόντος frame
$B_{(x,y)}$ pixel frame αναφοράς: (x, y)

(Το πώς και το γιατί των αριθμών θα εξηγηθεί στο αντίστοιχο κεφάλαιο επεξεργασίας βίντεο. Εκεί θα τεκμηριωθεί και η σημασία του υπολογισμού του ΑΑΔ).

Σημασία για μας έχει η μορφή του προβλήματος η οποία μας ζητά την ελαχιστοποίηση του ΑΑΔ. Συνεπώς χρειαζόμαστε την υλοποίηση δύο πράξεων : σύγκριση και πρόσθεση.

Κατόπιν, προσαρμόζουμε το πρόβλημά μας στο πρότυπο στο οποίο μελετούμε, το Η.264. Το τελευταίο προϋποθέτει κάποιες συγκεκριμένες παραμέτρους που δίνουν βέλτιστη - τη στιγμή της συγγραφής - ποιότητα βίντεο. Αυτό σημαίνει συγκεκριμένο μέγεθος, αριθμό και επεξεργασία των εισόδων. Το κύκλωμα που έχουμε δημιουργήσει μπορεί αν έχει πολλές διαφορετικές αρχιτεκτονικές σε *RTL* επίπεδο.

Έστω G : (V,E) κατευθυνόμενο γράφημα, V το σύνολο των κορυφών και E το σύνολο των ακμών. Έστω ,επίσης, σχέση R_1 : (V x E x E x…x E) που προσδιορίζει τη σχέση εισόδων και εξόδων ανά κόμβο. Η σχέση αυτή δεν έχει καθορισμένο αριθμό μελών αφού δεν υπάρχει περιορισμός στον αριθμό εισόδων και εξόδων ενός κυκλώματος. Έστω σχέση R_2 : (V U E) το εμβαδό που αντιστοιχεί σε κάθε κόμβο ή ακμή. Έστω τέλος R_3 : (V x V) η χρονική καθυστέρηση σταθεροποίησης του αποτελέσματος μίας εξόδου κατά τη

μεταβολή τιμής της εισόδου. Υποθέτουμε πως λόγω της φύσης του προβλήματος όλες οι είσοδοι και οι έξοδοι είναι ισοδύναμες (έχουμε συμμετρία στο πρόβλημα σε ό,τι αφορά τις εισόδους μεταξύ τους).

Επιθυμούμε να βρούμε τις βέλτιστες, ελάχιστες τιμές για τα R_2 **ΚΑΙ** R_3 με περιορισμό τη σχέση R_1. Οι σχέσεις αυτές όμως είναι πάντα ανάλογες του γράφου G στον οποίο αναφέρονται και που εξαρτάται από την αρχιτεκτονική του κυκλώματος. Συνεπώς ορίζουμε και ένα σύνολο - διαφορετικών μεταξύ τους - γραφημάτων $GG = \{G_1, G_2 \ldots G_n\}$ που περιγράφει τις διαφορετικές αρχιτεκτονικές που περιγράφουν το ΑΑΔ και που θα δοκιμάσουμε.

(Πριν γίνει η αντιστοίχηση του προβλήματος σε μία φυσική βιβλιοθήκη δεν είναι δυνατό να καταλήξουμε σε τιμές, παρά μόνο σε προσδιορισμό των σχέσεων. Κάτι τέτοιο όμως δεν ανήκει στους στόχους της εργασίας που έχει κατεύθυνση καθαρά πρακτική και όχι θεωρητική. Εξάλλου, χωρίς τιμές η φύση του προβλήματος το καθιστά άλυτο. Μέσω εργαλείων που πραγματοποιούν βελτιστοποίηση που έχουν δοκιμαστεί στη βιομηχανία μικροηλεκτρονικής θα καταλήξουμε απευθείας στο ζητούμενο, χωρίς να μας απασχολήσουν οι ακριβείς αλγόριθμοι βελτιστοποίησης. Με λίγα λόγια προσθέτουμε άλλο ένα σύνολο περιορισμών το οποίο μας το προσφέρει η φυσική βιβλιοθήκη).

Γράφουμε στο επόμενο στάδιο διάφορες αρχιτεκτονικές, είτε χρησιμοποιώντας βιομηχανικά πρότυπα αθροιστών, είτε αρχιτεκτονικές αθροιστών σχεδιασμένες στο χέρι, είτε χρησιμοποιώντας μία πειραματική προσαρμοσμένη αρχιτεκτονική σε επίπεδο πύλης που χρησιμοποιεί *συγκριτές*.

Αυτή η RTL/GaLe μορφή του κάθε γραφήματος θα περαστεί από ένα επίπεδο σύνθεσης για να βελτιστοποιηθεί το κάθε γράφημα σύμφωνα με τους περιορισμούς που περιγράψαμε προηγουμένως. Η έξοδος της σύνθεσης - της οποίας ο ορισμός θα εξηγηθεί μαζί με τους υπόλοιπους παρακάτω (προς το παρόν έχουμε στο μυαλό μας απλά ένα μαύρο κουτί που προσθέτει πληροφορία ανά κομμάτι) – είναι μία Συνδεσμολογία σε Επίπεδο Πύλης

(*GaLe*) και τιμές για τις σχέσεις R_1, R_2 και R_3. Οι τιμές αυτές είναι βέλτιστες μιας και έχει γίνει η αντιστοίχηση στη φυσική βιβλιοθήκη που χρησιμοποιούμε. Επίσης άλλες παράμετροι που καθορίσαμε είναι μερικά όρια στις τιμές των σχέσεων R_1, R_2, R_3, τα οποία έχουν καθοριστεί για αδιάφορους προς την εργασία λόγους.

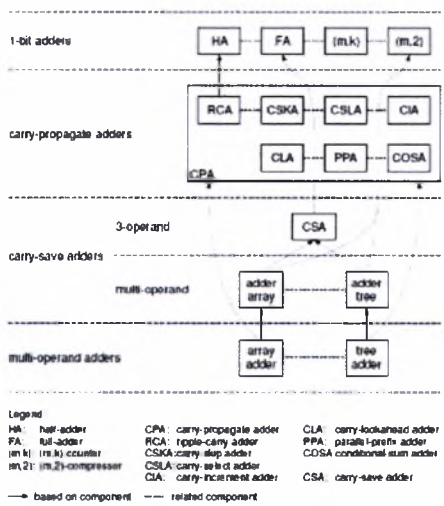
Η βέλτιστη, ως προς το συνδυασμό των σχέσεων R_2 του συνολικού κυκλώματος και R_3 του *critical path*, GaLe συνδεσμολογία σε επόμενο στάδιο θα τοποθετηθεί και θα δρομολογηθεί έτσι ώστε να λάβουμε στην έξοδο ένα έτοιμο προς υλοποίηση κύκλωμα.

Αθροιστές

Οφείλουμε να αναλύσουμε εκτός από την πράξη της πρόσθεσης, τις αρχιτεκτονικές, τις αποδόσεις και τα πλεονεκτήματα και μειονεκτήματα που φέρουν. Γιατί μας αφορά όμως αυτό, αφού τα σύγχρονα εργαλεία επιλέγουν από μόνα τους τις βέλτιστες υλοποιήσεις;

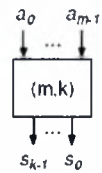
- Για να έχουμε κατανόηση της απόδοσής μας εξαιτίας της δομής.
- Γιατί συχνά εμφανίζονται στο critical path.
- Γιατί είναι βασικό κομμάτι των περισσότερων σχεδιάσεων.

Μελετάμε, λοιπόν, αλγόριθμους υλικού για αριθμητικά modules και εστιάζουμε συγκεκριμένα στους αθροιστές. (Αναλύοντας πολυπλοκότητες χρησιμοποιούμε το μοντέλο unit-gate για το οποίο βλέπε παράρτημα.)



Μονόμπιτοι αθροιστές ή αλλιώς (m,k) μετρητές

Ως βασική συνδυαστική δομή άθροισης, ο μονόμπιτος αθροιστής υπολογίζει το άθροισμα των bits εισόδου της ίδιας τάξης μεγέθους (δηλαδή μονόμπιτους αριθμούς). Λέγεται επίσης (m,k)-μετρητής, επειδή μετρά τον αριθμό των άσσων στις m εισόδους (a_{m-1}, a_{m-2}, … ,a_0) και βγάζει στην έξοδο ένα άθροισμα με k bits (s_{k-1}, s_{k-2}, … , s_0) όπου k = CEIL(log(m+1)) με τη συνάρτηση CEIL να δίνει τον αμέσως μεγαλύτερο ακέραιο από το όρισμά της.

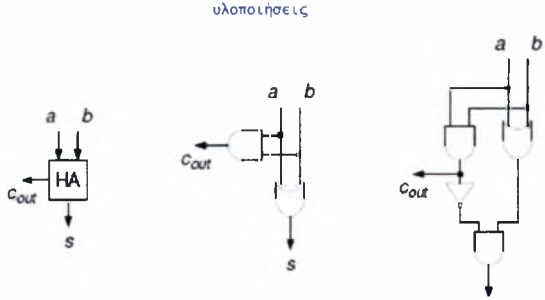


$$\sum_{j=0}^{k-1} 2^j s_j = \sum_{i=0}^{m-1} a_i$$

Ημιαθροιστής

Είναι το συνδυαστικό κύκλωμα που έχει ως εισόδους δύο δυαδικά ψηφία και έξοδο το άθροισμά τους και το πιθανό κρατούμενο. Το πιο σημαντικό ψηφίο ονομάζεται carry-out(c_{out}) επειδή μεταφέρει το γεγονός της υπερχείλισης στο αμέσως υψηλότερο σε θέση bit. Αν X, Y οι μονοψήφιοι αριθμοί προς άθροιση και S,C το άθροισμα και το κρατούμενό τους αντίστοιχα ισχύει ο πίνακας αληθείας και οι εξισώσεις που ακολουθούν.

Σύμβολο και δύο υλοποιήσεις



Πίνακας αληθείας

A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Αριθμητικές εξισώσεις

$$
\begin{aligned}
2c_{out} + s &= a + b \\
s &= (a + b) \bmod 2 \\
c_{out} &= (a + b) \operatorname{div} 2 = \tfrac{1}{2}(a + b - s)
\end{aligned}
$$

Λογικές εξισώσεις

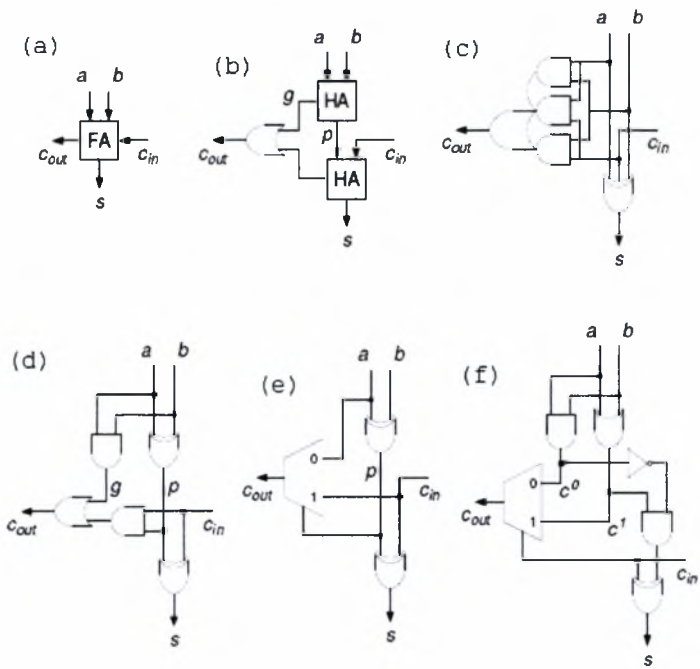
$$
\begin{aligned}
s &= a \oplus b \\
c_{out} &= ab
\end{aligned}
$$

:

Πολυπλοκότητα

$$
\begin{aligned}
T_{HA}(a.b \rightarrow c_{out}) &= 1 \\
T_{HA}(a.b \rightarrow s) &= 2
\end{aligned}
\qquad A_{HA} = 3
$$

πλήρης αθροιστής



Ο πλήρης αθροιστής ενός ψηφίου διαφέρει από τον ημιαθροιστή στο ότι έχει μία παραπάνω είσοδο, αυτή του κρατούμενου εισόδου. Σημαντικά εσωτερικά σήματα του πλήρους αθροιστή είναι τα ***generate*** (δημιουργία)(g) και το ***propagate*** (διάδοση)(p). Το πρώτο υποδεικνύει αν ένα σήμα κρατουμένου δημιουργείται εντός του πλήρους αθροιστή. Το σήμα διάδοσης υποδεικνύει αν ένα κρατούμενο στην είσοδο διαδίδεται στο κρατούμενο εξόδου χωρίς να αλλαχτεί διαμέσου του πλήρους αθροιστή. Εναλλακτικά μπορούμε να υπολογίσουμε δύο ενδιάμεσα σήματα c^0 και c^1 για τις αντίστοιχες τιμές του κρατουμένου εισόδου. Έτσι το κρατούμενο εξόδου

μπορεί να υπολογιστεί συναρτήσει των (g,p) ή των (c^0, c^1) και του κρατουμένου εισόδου και μπορεί να υλοποιηθεί είτε με AND-OR είτε με πολυπλέκτη.

Πίνακας αληθείας

A	B	Ci	Co	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Αριθμητικές εξισώσεις

$$\begin{aligned} 2c_{out} + s &= a + b + c_{in} \\ s &= (a + b + c_{in}) \bmod 2 \\ c_{out} &= (a + b + c_{in}) \operatorname{div} 2 = \tfrac{1}{2}(a + b + c_{in} - s) \end{aligned}$$

Λογικές εξισώσεις

$$\begin{aligned} g &= ab \\ p &= a \oplus b \\ c^0 &= ab \\ c^1 &= a + b \\ s &= a \oplus b \oplus c_{in} \\ &= p \oplus c_{in} \\ c_{out} &= ab + ac_{in} + bc_{in} \\ &= ab + (a + b)c_{in} = ab + (a \oplus b)c_{in} \\ &= g + pc_{in} \\ &= \overline{pg} + pc_{in} = \overline{pa} + pc_{in} \\ &= \overline{c}_{in}c^0 + c_{in}c^1 \end{aligned}$$

Πολυπλοκότητα

$$\begin{aligned} T_{FA}(a, b \to c_{out}) &= 4\,(2) \\ T_{FA}(a, b \to s) &= 4 \\ T_{FA}(c_{in} \to c_{out}) &= 2 \\ T_{FA}(c_{in} \to s) &= 2\,(4) \end{aligned}$$

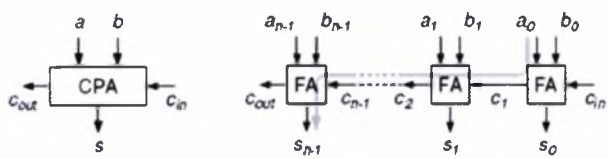
$$A_{FA} = 7\,(9)$$

Ένας πλήρης αθροιστής μπορεί να κατασκευαστεί χρησιμοποιώντας ημιαθροιστές, πύλες 2 εισόδων, πολυπλέκτες ή άλλες πολύπλοκες πύλες. Οι λύσεις (b) και (d) της πρώτης εικόνας και μέχρι κάποιο σημείο η (e) χρησιμοποιούν τα σήματα g και p (generate-propagate scheme - σχήμα δημιουργίας-διάδοσης για το οποίο θα μιλήσουμε στη συνέχεια). Το κύκλωμα (f) βασίζεται στη δημιουργία και των δύο πιθανών σημάτων κρατουμένου c^0 και c^1 και την επιλογή του σωστού χρησιμοποιώντας το

κρατούμενο εισόδου c_{in} (carry-select scheme - σχήμα επιλογής κρατουμένου). Η λύση (c) δημιουργεί το σήμα αθροίσματος s με μία XOR 3 εισόδων και το κρατούμενο εξόδου c_{out} με μία πύλη πλειοψηφίας. Αυτή η σύνθετη πύλη έχει γρήγορη δημιουργία κρατουμένου αλλά είναι μεγαλύτερη, όπως φαίνεται και από την ανάλυση πολυπλοκότητας. Επειδή η πύλη πλεοψηφίας όμως μπορεί να υλοποιηθεί με ικανοποιητική απόδοση σε transistor level (επίπεδο τρανζίστορ), της δίνεται αριθμός πυλών 5 και καθυστέρηση πυλών 2. Ο πολυπλέκτης μετρά 3 πύλες και 2 σε καθυστέρηση. (Για το μοντέλο unit-gate βλέπε παράρτημα.)

Αθροιστές Διάδοσης Κρατουμένου (Carry-Propagate Adders - CPA)

Σύμβολο και αρχιτεκτονική



Αριθμητικές εξισώσεις

Ένας ***Αθροιστής Διάδοσης Κρατουμένου - ΑΔΚ (Carry-Propagate Adder - CPA)*** προσθέτει δύο τελεστέους των n bits A = (a_{n-1}, a_{n-2}, … ,a_0) και B = (b_{n-}

$$2^n c_{out} + S = A + B + c_{in}$$

$$2^n c_{out} + \sum_{i=0}^{n-1} 2^i s_i = \sum_{i=0}^{n-1} 2^i a_i + \sum_{i=0}^{n-1} 2^i b_i + c_{in}$$

$$= \sum_{i=0}^{n-1} 2^i (a_i + b_i) + c_{in}$$

$$2c_{i+1} + s_i = a_i + b_i + c_i \; ; \quad i = 0.1.\ldots.n-1$$

where $c_0 = c_{in}$ and $c_{out} = c_n$

$_1$, b_{n-2}, … ,b_0) μαζί με ένα προαιρετικό κρατούμενο εισόδου c_{in} εκτελώντας διάδοση κρατουμένου. Το αποτέλεσμα είναι ένας αριθμός από n+1 bits που αποτελεί το άθροισμα των n bits S = (s_{n-1}, s_{n-2}, … , s_0) και το κρατούμενο εξόδου c_0:Παρακάτω περιγράφεται η λογική για ανά bit ακολουθιακή πρόσθεση 2 αριθμών από n bit ο καθένας. Μπορεί να υλοποιηθεί με συνδυαστικό κύκλωμα χρησιμοποιώντας n πλήρεις αθροιστές συνδεδεμένους σε σειρά. Ονομάζεται ***ripple-carry adder*** :

Λογικές εξισώσεις

$$
\begin{aligned}
g_i &= a_i\, b_i \\
p_i &= a_i \oplus b_i \\
s_i &= p_i \oplus c_i \\
c_{i+1} &= g_i + p_i\, c_i \quad ; \quad i = 0, 1, \ldots, n-1 \\
&\text{where } c_0 = c_{in} \text{ and } c_{out} = c_n
\end{aligned}
$$

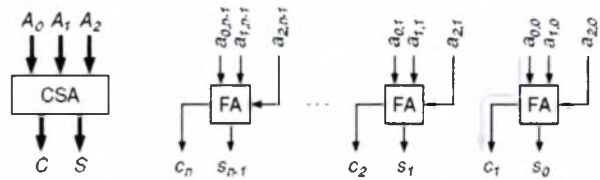
Πολυπλοκότητα

$$
\begin{aligned}
T_{CPA}(a, b \rightarrow c_{out}, s) &= 2n + 2 \\
T_{CPA}(c_{in} \rightarrow c_{out}, s) &= 2n
\end{aligned}
\qquad A_{CPA} = 7n
$$

Σημειώστε πως ο χρόνος υπολογισμού του αθροιστή αυτού αυξάνει γραμμικά με το μήκος τελεστέου εξαιτίας της σειριακής διάδοσης κρατουμένου.

Αθροιστές Συγκράτησης Κρατουμένου (Carry-Save Adders - CSA)

Σύμβολο και αρχιτεκτονική



Ο αθροιστής αυτός αποφεύγει τη διάδοση του κρατουμένου χειριζόμενος τα ενδιάμεσα κρατούμενα ως εξόδους αντί να τα προωθήσει. Το άθροισμα προκύπτει ως ένας ***πλεονάζων carry-save αριθμός*** n ψηφίων, που αποτελείται από δύο δυαδικούς αριθμούς, τα ψηφία αθροίσματος S και τα ψηφία κρατουμένων C . Ένας αθροιστής συγκράτησης κρατουμένου δέχεται τρεις δυαδικές εισόδους ή ,εναλλακτικά, ένα δυαδικό και ένα τελετστέο carry-save. Υλοποιείται από μία γραμμική αναδιάταξη πλήρων αθροιστών και έχει σταθερή καθυστέρηση (ανεξάρτητη του n). Παρατίθενται οι εξισώσεις, η διάταξη και το σύμβολο, αντιστρόφως αντίστοιχα.

Αριθμητικές εξισώσεις

$$2C + S = A_0 + A_1 + A_2$$

$$\sum_{i=1}^{n} 2^i c_i + \sum_{i=0}^{n-1} 2^i s_i = \sum_{j=0}^{2} \sum_{i=0}^{n-1} 2^i a_{j,i}$$

$$2c_{i+1} + s_i = \sum_{j=0}^{2} a_{j,i} \quad ; \quad i = 0, 1, \ldots, n-1$$

Πολυπλοκότητα

$$T_{CSA}(a_0, a_1 \rightarrow c, s) = 4$$
$$T_{CSA}(a_2 \rightarrow c, s) = 2$$
$$A_{CSA} = 7n$$

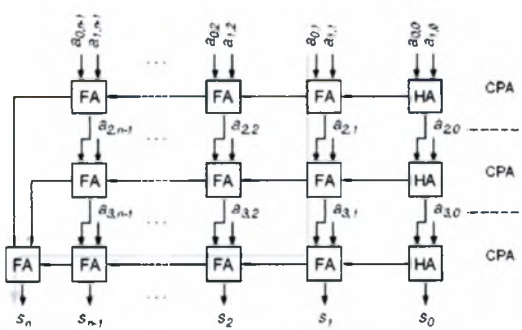
Αθροιστές Πολλών Τελεστέων

Χρησιμοποιούνται για την άθροιση m τελεστέων αριθμών των n bits A_{m-1}, … A_0 όπου m>2 παράγοντας ένα αποτέλεσμα S σε μη πλεονάζουσα αναπαράσταση με n + CEIL(log m) bits. Αναφερθήκαμε σε αυτούς τους αθροιστές μόνο για την πληρότητα του κειμένου και δε θα μας απασχολήσουν γενικά, παρά μόνο μία περίπτωσή τους. Ισχύει:

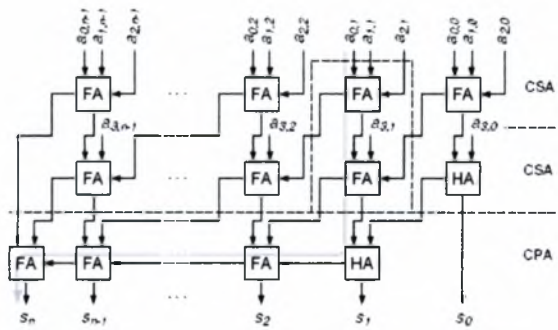
$$S = \sum_{j=0}^{m-1} A_j$$

Πινακοαθροιστές (Array Adders)

Ένας αθροιστής m τελεστέων μπορεί να υλοποιηθεί από σειριακή συνένωση m-1 αθροιστών διάδοσης κρατουμένου,

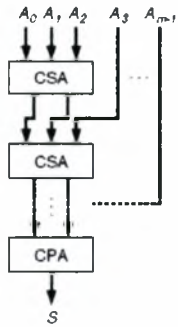


ή από m-2 αθροιστές συγκράτησης ακολουθούμενους από έναν τελικό αθροιστή διάδοσης:



Οι δύο διατάξεις ονομάζονται ***πίνακες αθροιστών(array adders)*** και μοιάζουν εξαιρετικά στη λογική τους δομή και το υλικό από το οποίο αποτελούνται.(προσοχή στη διαφορά *adder arrays* και *array adders'* οι adder arrays - αθροιστές πινάκων - είναι CSA που αποτελούνται από ένα πίνακα κελιών αθροιστών ενώ οι array adders είναι αθροιστές πολλών τελεστέων που χρησιμοποιούν ένα CSA πίνακα και ένα τελικό CPA) Το ίδιο, μάλιστα, ισχύει και με το κρίσιμο μονοπάτι. Όμως οι χρόνοι άφιξης στον τελευταίο αθροιστή διάδοσης είναι διαφορετικοί. Αντίθετα με τον πίνακα CSA , στον πίνακα CPA τα bits υψηλότερης τάξης έρχονται αργότερα από τα πρώτα. Έτσι ακριβώς όμως τα περιμένει ο αθροιστής του τελευταίου επιπέδου.

Συνεπώς για να επιταχύνουμε την πρώτη αρχιτεκτονική, τον πίνακα CPA, πρέπει να επιταχύνουμε όλα τα τμήματα, όλους τους αθροιστές διάδοσης με κάτι γρηγορότερο. Συνήθως αρχικά έχουμε ripple carry adders και το ζητούμενο είναι κάτι γρηγορότερο για να αντικαταστήσουμε ολόκληρο τον πίνακα. Από την άλλη πλευρά στον πίνακα CSA, αρκεί να αντικαταστήσουμε τον τελικό αθροιστή διάδοσης με ένα γρήγορο παράλληλης διάδοσης κρατουμένου αθροιστή. Συνεπώς κατασκευάζουμε γρήγορους πίνακες αθροιστών με ένα πίνακα CSA και ένα τελικό στάδιο γρήγορου CPA:



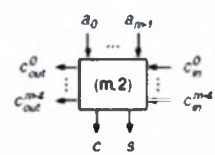
Αρα εχουμε:

$$T_{ARRAY} = (m-2)T_{CSA} + T_{CPA}$$
$$A_{ARRAY} = (m-2)A_{CSA} + A_{CPA}$$

(m,2)-συμπιεστές

Κάθε στήλη (φέτα-slice) ενός CSA πίνακα είναι ένας αθροιστής 1 bit που ονομάζεται και συμπιεστής (m,2) αφού συμπιέζει m bits εισόδου σε 2 bits αποτελέσματος άθροισης, ένα για το άθροισμα και ένα για το κρατούμενο προωθώντας τα m-3 ενδιάμεσα κρατούμενα στο επόμενο υψηλότερο επίπεδο.

Σύμβολο



Δεν έχουμε οριζόντια διάδοση κρατουμένου. Για παράδειγμα στο παραπάνω σχήμα το $c_{in,1}$ επηρεάζει μόνο τα $c_{out,k>1}$. Χτίζεται χρησιμοποιώντας m-2 πλήρεις αθροιστές ή μικρότερους συμπιεστές.

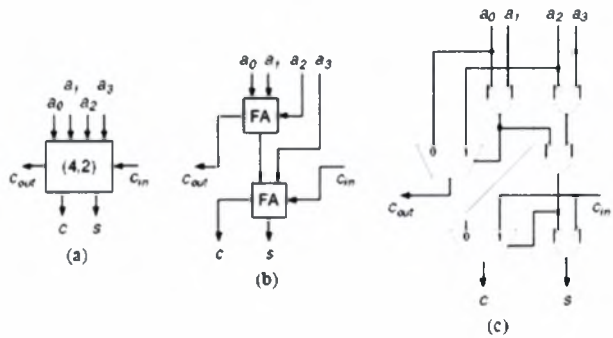
Αριθμητική εξίσωση

$$2(c + \sum_{i=0}^{m-4} c_{out}^i) + s = \sum_{i=0}^{m-1} a_i + \sum_{i=0}^{m-4} c_{in}^i$$

Πολυπλοκότητα

$$T_{(m,2)} = O(m) \qquad A_{(m,2)} = O(\log m)$$

(4,2)-συμπιεστής



Τον (m,k)-συμπιεστή τον αναφέραμε για να μιλήσουμε για τον (4,2)-συμπιεστή. Επιτρέπει με κάποια αναδιάταξη των XOR πυλών του πλήρους αθροιστή να έχουμε κάποια βελτιστοποίηση.

Αριθμητικές εξισώσεις

$$2(c + c_{out}) + s = \sum_{i=0}^{3} a_i + c_{in}$$

Πολυπλοκότητα

$$T_{(4,2)}(a_i \rightarrow c, s) = 6$$
$$T_{(4,2)}(a_i \rightarrow c_{out}) = 4$$
$$T_{(4,2)}(c_{in} \rightarrow c, s) = 2$$

$$A_{(4,2)} = 14$$

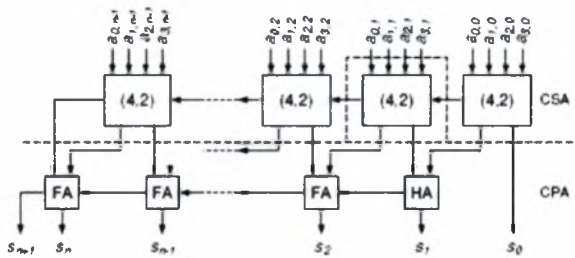
Δεντροαθροιστές(Tree adders)

Δέντρα αθροιστών (Adder trees) ή δέντρα Wallace είναι CSA που αποτελούνται από κυκλώματα συμπιεστών διατεταγμένα σε δέντρο. Οι tree adders είναι αθροιστές πολλών τελεστέων που αποτελούνται από ένα δέντρο CSA και ένα τελικό CPA. Χρησιμοποιώντας ένα γρήγορο τελικό CPA, εκτελούν τη γρηγορότερη άθροιση

Πολυπλοκότητα

$$T_{TREE} = T_{(m,2)} + T_{CPA}$$
$$A_{TREE} = nA_{(m,2)} + A_{CPA}$$

πολλών τελεστέων. Παρατίθεται ένας αθροιστής τεσσάρων τελεστέων με (4,2)-συμπιεστές:



Prefix αλγόριθμοι (Prefix Algorithms)

Η πρόσθεση δύο αριθμών μπορεί να οριστεί και διαφορετικά, ως ένα πρόβλημα προθέματος. Οι αντίστοιχοι αλγόριθμοι παράλληλου προθέματος μπορούν να χρησιμοποιηθούν για να επιταχύνουμε την άθροιση και να κατανοήσουμε τις διάφορες αρχές που τη διέπουν. Το επόμενο υποκεφάλαιο εισάγει μία μαθηματική και οπτική μοντελοποίηση προβλημάτων προθέματος και αλγορίθμων.

Προβλήματα προθέματος

Σε ένα *πρόβλημα προθέματος* n έξοδοι (y_{n-1}, y_{n-2}, …, y_0) υπολογίζονται από n εισόδους (x_{n-1}, x_{n-2}, …, x_0) χρησιμοποιώντας έναν αυθαίρετο προσεταιριστικό δυαδικό τελεστή ($\bullet$) ως εξής:

$$
\begin{aligned}
y_0 &= x_0 \\
y_1 &= x_1 \bullet x_0 \\
y_2 &= x_2 \bullet x_1 \bullet x_0 \\
&\vdots \\
y_{n-1} &= x_{n-1} \bullet x_{n-2} \bullet \cdots \bullet x_1 \bullet x_0
\end{aligned}
$$

Που αναδρομικά μπορεί να γραφεί:

$$
\begin{aligned}
y_0 &= x_0 \\
y_i &= x_i \bullet y_{i-1} \quad ; \quad i = 1.2.\ldots.n-1
\end{aligned}
$$

Συνεπώς, σε ένα πρόβλημα προθέματος κάθε έξοδος εξαρτάται από όλες τις εισόδους μικρότερης ή ίσης τάξης και κάθε είσοδος επηρεάζει κάθε έξοδο μεγαλύτερης ή ίσης τάξης.

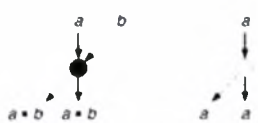
Εξαιτίας της προσεταιριστικότητας του τελεστή-προθέματος ($\bullet$), οι πράξεις μπορούν να εκτελεστούν με οιαδήποτε σειρά. Συγκεκριμένα, σειρές πράξεων μπορούν να ομαδοποιηθούν ώστε να λύσουν το πρόβλημα προθέματος

μερικά και παράλληλα για ομάδες(ακολουθίες) bit εισόδου (x_i, x_{i-1}, …, x_k), δίνοντας ως αποτέλεσμα τις ομαδοποιημένες μεταβλητές $Y_{i:k}$. Σε μεγαλύτερα επίπεδα, ακολουθίες ομαδοποιημένων μεταβλητών μπορούν να αποτιμηθούν, καταλαμβάνοντας m επίπεδα ενδιάμεσων ομαδοποιημένων μεταβλητών, όπου η ομαδοποιημένη μεταβλητή $Y_{i:k,l}$ εμφαίνει το αποτέλεσμα προθέματος των bits (x_i, x_{i-1},…, x_k) στο επίπεδο l. Τοιουτοτρόπως οι μεταβλητές του τελευταίου επιπέδου m πρέπει να καλύπτουν όλα τα bits από το i στο 0 ($Y_{i:0,m}$). Έτσι αναπαριστούν τα αποτελέσματα του προβλήματος προθέματος:

$$
\begin{aligned}
Y^0_{i,i} &= x_i \\
Y^l_{i,k} &= Y^{l-1}_{i,j+1} \bullet Y^{l-1}_{j,k} \,.\; k \le j \le i \;:\; l = 1.2.\ldots.,m \\
y_i &= Y^m_{i,0} \;:\; i = 0.1.\ldots.n-1
\end{aligned}
$$

Σημειώστε πως για j=i η ομαδοποιημένη μεταβλητή $Y_{i:k,l-1}$ παραμένει αμετάβλητη ($Y_{i:k,l-1}$= $Y_{i:k,l}$). Αφού τα προβλήματα προθέματος περιγράφουν μία συνδυαστική σχέση ανάμεσα στην είσοδο και την έξοδο, μπορούν να επιλυθούν με λογικά δίκτυα.

Υπάρχουν διάφοροι σειριακοί και παράλληλοι αλγόριθμοι που λύνουν προβλήματα προθέματος εκμεταλλευόμενοι τις ομαδοποιήσεις που συμβαίνουν στη σχέση που αναφέραμε παραπάνω. Καταλήγουν σε πολύ διαφορετικά αποτελέσματα εμβαδού και καθυστέρησης όταν απεικονίζουν ένα λογικό δίκτυο.



Στους γράφους οι μαύροι κόμβοι (•) συμβολίζουν κόμβους που εκτελούν τη δυαδική προσεταιριστική πράξη (•) στις δύο εισόδους (βλέπε j < i στη σχέση παραπάνω), ενώ οι λευκοί κόμβοι (o) αναπαριστούν ελεύθερη είσοδο χωρίς λογική επεξεργασία (j = i)

Κάθε μία από τις n στήλες θα αντιστοιχεί σε ένα bit. Μαύροι κόμβοι παράλληλα, τίθενται στην ίδια σειρά ενώ όταν οι πράξεις αυτές γίνονται σε σειρά τους τοποθετούμε σε διαδοχικές γραμμές. Έτσι, ο αριθμός m των γραμμών αντιστοιχεί στο μέγιστο αριθμό δυαδικών πράξεων σε σειρά. Οι έξοδοι της σειράς l είναι οι ομαδοποιημένες μεταβλητές $Y_{1:k,l}$.Το κενό ανάμεσα στις γραμμές θα αντικατοπτρίζει την ποσότητα διασυνδέσεων (δηλαδή τον αριθμό των απαραίτητων καλωδιώσεων) ανάμεσα σε διαδοχικές γραμμές. Ταυτόχρονα ο γράφος αναπαριστά και πιθανές υλοποιήσεις σε hardware.

Έστω τώρα τα παρακάτω μέτρα πολυπλοκότητας για τους αλγόριθμους προθέματος, στοχεύοντας στην υλοποίηση κυκλωμάτων:

Χρόνος υπολογισμού $T_\bullet$: μαύροι κόμβοι στο κρίσιμο μονοπάτι(critical path) ή αριθμός γραμμών, $T_\bullet = m$

Εμβαδό μαύρων κόμβων $A_\bullet$: Συνολικός αριθμός μαύρων κόμβων, απαραίτητο για σχεδιάσεις με κελιά, όπου οι λευκοί κόμβοι δε μας απασχολούν.

Εμβαδό μαύρων και λευκών κόμβων $A_{\bullet\circ}$: Συνολικός αριθμός μαύρων και λευκών κόμβων, στο οποίο αναφερόμαστε συνήθως για λόγους κανονικότητας σε custom layout σχεδιάσεις.

Γινόμενο εμβαδού-καθυστέρησης(area-time product) $A_\bullet T_\bullet$

Εμβαδό διασυνδέσεων A_{tracks} : Συνολικός αριθμός οριζόντιων καλωδιώσεων που χρησιμοποιούνται για να διασυνδέσουν την τοπολογία.

Μέγιστο fan-out FO_{max} : fan-out αριθμός του κόμβου με το μεγαλύτερο.

Εξαιτίας του δυαδικού συστήματος, οι εξισώσεις που περιέχουν σύμβολο ισότητας = ισχύουν για κάθε μήκος λέξης(εισόδου) που είναι δύναμη του 2, διαφορετικά ισχύουν προσεγγιστικά.

Τρεις κατηγορίες αλγορίθμων προθέματος μπορούμε να διακρίνουμε: *σειριακού προθέματος(serial-prefix)*, *ομαδικού προθέματος(group-prefix)* και *δεντρικού προθέματος(tree-prefix)*. Στη βιβλιογραφία οι δεντρικού προθέματος αναφέρονται με το κοινό όνομα των παράλληλου προθέματος. Όμως και οι ομαδικού προθέματος χρησιμοποιούν παραλληλία. Για τους ομαδικού προθέματος αλγόριθμους δε θα συζητήσουμε μιας και δε μας είναι χρήσιμοι στη συνέχεια της εργασίας.

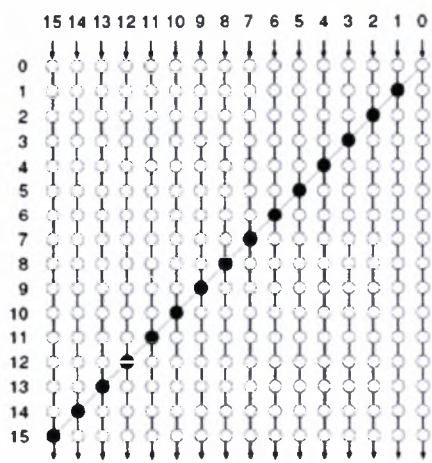
Σειριακού προθέματος αλγόριθμος (Serial-prefix algorithm)

Η εξίσωση που προαναφέραμε και φαίνεται στο διπλανό σχήμα αναπαριστά ένα σειριακό αλγόριθμο

$$y_0 = x_0$$
$$y_i = x_i \bullet y_{i-1} \quad ; \quad i = 1, 2, \ldots, n-1$$

για τη λύση του προβλήματος προθέματος. Ο σειριακού προθέματος αλγόριθμος χρειάζεται έναν ελάχιστο αριθμό δυαδικών πράξεων (•) (O(n)) αλλά είναι εγγενώς αργός (O(n)). Προφανώς οι n-1 μαύροι κόμβοι μπορούν να διαταχθούν σε μία μονή γραμμή για υλοποίηση σε hardware, ξεκαθαρίζοντας όλους τους λευκούς κόμβους (δηλαδή $A_{\bullet+\circ}=A_\bullet$, $A_{tracks}=1$).

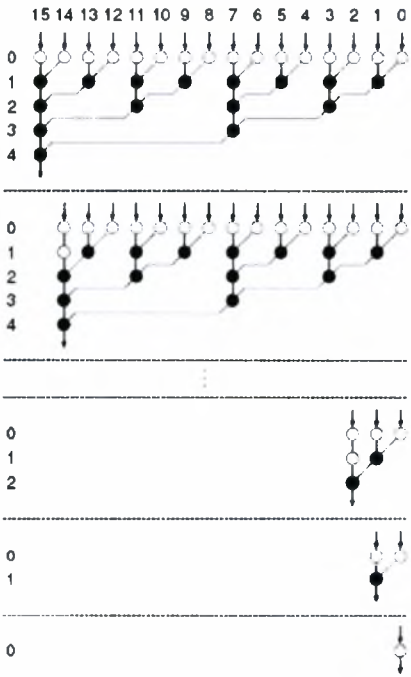
Σειριακός prefix αλγόριθμος 16 εισόδων



$$T_\bullet = n-1$$
$$A_\bullet = n-1$$
$$A_{\bullet+\circ} = n^2$$
$$A_\bullet T_\bullet \approx n^2$$
$$A_{tracks} = 1$$
$$FO_{max} = 2$$

Δεντρικού προθέματος αλγόριθμοι (Tree-Prefix algorithms)

με βελτιστοποιημένος tree-prefix αλγόριθμος για 16 εισόδους.



$$T_\bullet = \log n$$
$$A_\bullet = \tfrac{1}{2}n^2 - \tfrac{1}{2}n$$
$$A_\bullet T_\bullet \approx \tfrac{1}{2}n^2 \log n$$
$$A_{tracks} = \log n$$
$$FO_{max} = \begin{cases} n & \text{at inputs} \\ 1 & \text{internally} \end{cases}$$

Σύμφωνα με τη παρακείμενη εξίσωση όλες οι έξοδοι μπορούν να υπολογιστούν ξεχωριστά και παράλληλα. Διατάσσοντας τις πράξεις των μαύρων κόμβων σε δομή δέντρου, ο υπολογιστικός χρόνος για κάθε έξοδο μπορεί να μειωθεί σε O(logn). Όμως, ο συνολικός αριθμός των πράξεων (•) που πρέπει να

$$\begin{aligned} y_0 &= x_0 \\ y_1 &= x_1 \bullet x_0 \\ y_2 &= x_2 \bullet x_1 \bullet x_0 \\ &\vdots \\ y_{n-1} &= x_{n-1} \bullet x_{n-2} \bullet \cdots \bullet x_1 \bullet x_0 \end{aligned}$$

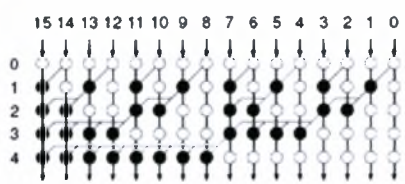
υπολογίσουμε εκτοξεύει το κόστος σε υλικό σε $O(n^2)$, εφόσον χρησιμοποιήσουμε ξεχωριστό δέντρο υπολογισμού για κάθε έξοδο.

Έτσι καταλήγουμε στο ότι μπορούμε να χρησιμοποιήσουμε μία «τράμπα» (trade-off), συνενώνοντας τα ξεχωριστά δέντρα υπολογισμού(που σημαίνει πως οι κοινές υποεκφράσεις θα είναι μερισμένες) σε ένα τέτοιο βαθμό σύμφωνα με τους διάφορους tree-prefix αλγόριθμους, μειώνοντας το εμβαδό σε $O(n\log n)$ ή ακόμα και σε $O(n)$. Παρακάτω παρατίθενται οι πιο γνωστοί αλγόριθμοι που χρησιμοποιούν αυτή τη στρατηγική.

Sklansky tree-prefix αλγόριθμος

Απλά επικαλύπτοντας όλα τα δέντρα υπολογισμού των εξόδων στο μη βελτιστοποιημένο prefix αλγόριθμο, οδηγούμαστε στον αλγόριθμο του Sklansky. Ενδιάμεσα σήματα υπολογίζονται από μία ελαχιστοποιημένη δομή δέντρου και διανέμονται παράλληλα στις υψηλότερες τάξεις bit που χρειάζονται το σήμα. Αυτό οδηγεί σε υψηλό fan-out ορισμένων μαύρων κόμβων ($O(n)$, απεριόριστο fan-out), αλλά οδηγεί στο μικρότερο δυνατό αριθμό καθυστέρησης σε κόμβους (ελάχιστο βάθος δέντρου), μικρό αριθμό σημάτων και πολύ λίγες καλωδιώσεις ($O(\log n)$).

Sklansky tree-prefix αλγόριθμος για 16 εισόδους.

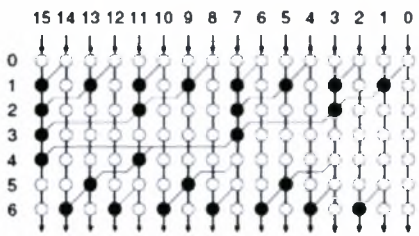


$$
\begin{aligned}
T_{\bullet} &= \log n \\
A_{\bullet} &= \tfrac{1}{2} n \log n \\
A_{\bullet+\circ} &= n \log n \\
A_{\bullet} T_{\bullet} &\approx \tfrac{1}{2} n \log^2 n \\
A_{tracks} &= \log n \\
FO_{max} &= \tfrac{1}{2} n
\end{aligned}
$$

Brent-Kung tree-prefix αλγόριθμος

Αρκετά διαφορετικά χαρακτηριστικά με παρόμοια δομή παρουσιάστηκε από τους Brent και Kung. Η παράλληλη διανομή των ενδιάμεσων σημάτων του Sklansky αντικαθίσταται από δομή όχι ακριβώς δέντρου και μερικά σειριακή διάδοση σήματος. Αυτό σχεδόν διπλασιάζει τον αριθμό των καθυστερήσεων κόμβων αλλά μειώνει τον αριθμό των μαύρων σε O(n) και περιορίζει το fan-out σε logn ή ακόμη και 3, αν μελετούμε μέγιστο fan-out σε κάθε γραμμή ξεχωριστά(έχει νόημα μόνο αν οι λευκοί κόμβοι περιέχουν buffers). Έτσι αυτή η δομή θεωρείται πως έχει περιορισμένο fan-out.

Brent-Kung tree-prefix αλγόριθμος 16 εισόδων.



$$T_\bullet = 2\log n - 2$$

$$A_\bullet = 2n - \log n - 2$$

$$A_{\bullet+\circ} = 2n\log n - 2n$$

$$A_\bullet T_\bullet \approx 4n\log n$$

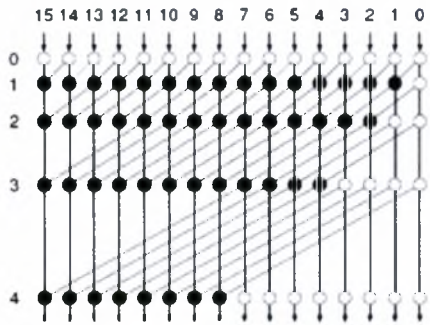
$$A_{tracks} = 2\log n - 1$$

$$FO_{max} = \log n$$

Kogge-Stone tree-prefix αλγόριθμος

Ο αλγόριθμος που προτάθηκε από τους Kogge και Stone έχει ελάχιστο βάθος, όπως αυτός του Sklansky αλλά και περιορισμένο fan-out(FO_{max} = 2) με το κόστος της μαζικής αύξησης τους αριθμού των μαύρων κόμβων και των διασυνδέσεων. Ο λόγος είναι πως χρησιμοποιούν μεγάλο αριθμό ανεξάρτητων δέντρων παράλληλα.

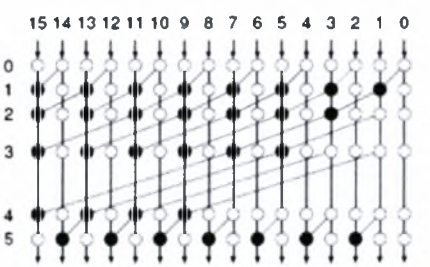
Kogge-Stone tree-prefix αλγόριθμος 16 εισόδων



$$
\begin{aligned}
T_{\bullet} &= \log n \\
A_{\bullet} &= n \log n - n + 1 \\
A_{\bullet+\circ} &= n \log n \\
A_{\bullet} T_{\bullet} &\approx n \log^2 n \\
A_{tracks} &= n - 1 \\
FO_{max} &= 2
\end{aligned}
$$

Han-Carlson tree-prefix αλγόριθμος

Οι Han και Carlson πρότειναν έναν αλγόριθμο που συνδυάζει τα πλεονεκτήματα των Brent-Kung και Kogge-Stone συνενώνοντας. Τα πρώτα και τελευταία k επίπεδα είναι Brent-Kung τύπου, ενώ τα ενδιάμεσα είναι Kogge-Stone(τυπικά ισχύει k=1). Ο αριθμός των παράλληλων δέντρων και άρα αυτός των μαύρων κόμβων και των διασυνδέσεων ελαττώνεται με το κόστος λίγο μεγαλύτερου critical path, σε σύγκριση με τον Kogge-Stone.



$$
\begin{aligned}
T_{\bullet} &= \log n + 1 \\
A_{\bullet} &= \tfrac{1}{2} n \log n \\
A_{\bullet+\circ} &= n \log n + n \\
A_{\bullet} T_{\bullet} &\approx \tfrac{1}{2} n \log^2 n \\
A_{tracks} &= \tfrac{1}{2} n + 1 \\
FO_{max} &= 2
\end{aligned}
$$

Συνεπώς: Ο Sklansky χρειάζεται επιπλέον buffering εξαιτίας του απεριόριστου fan-out. Οι Sklansky και Kogge-Stone είναι οι ταχύτεροι. Αναλόγως της κατάπτωσης της ταχύτητας που προκαλείται από το υψηλό fan-out(Sklansky), όπως επίσης και του μεγέθους του κυκλώματος και της πολυπλοκότητας καλωδίωσης(Kogge-Stone), τα μέτρα επιδόσεων διαφέρουν σε κάποιο βαθμό. Οι Brent-Kung και Han-Carlson αλγόριθμοι προσφέρουν λίγο πιο αργές αλλά και πιο μικρές σε εμβαδό λύσεις.

Ομαδικού προθέματος αλγόριθμοι (Group-prefix Algorithms)

Οι δομές δέντρων τυπικά διαιρούν τους τελεστέους σε σταθερού μεγέθους(στις περισσότερες περιπτώσεις ελαχιστοποιημένα)ομάδες bits και εφαρμόζουν ένα μέγιστο αριθμό από επίπεδα για τον υπολογισμό του προθέματος. Μία άλλη προσέγγιση χρησιμοποιεί μεταβλητού μεγέθους ομάδες από bits σε σταθερό αριθμό από επίπεδα(π.χ. ένα ή δύο επίπεδα). Το αποτέλεσμα είναι οι group-prefix αλγόριθμοι που ανοίγουν μία ευρεία γκάμα διαφορετικών στρατηγικών υπολογισμού του προθέματος.

Οι αλγόριθμοι αυτοί λόγω της ικανότητας που έχουν να είναι εξαιρετικά αποδοτικοί σε μεγάλο αριθμό από εισόδους δεν έχουν εφαρμογή στην αρχιτεκτονική του κυκλώματος την οποία εφαρμόζουμε. Για περισσότερες πληροφορίες ανατρέξατε στις πηγές.

Η δυαδική άθροιση ως ένα πρόβλημα προθέματος

Η δυαδική άθροιση διάδοσης κρατουμένου μπορεί να μοντελοποιηθεί ως ένα πρόβλημα προθέματος χρησιμοποιώντας το generate-propagate σχήμα ή το carry-select σχήμα που περιγράψαμε νωρίτερα.

Generate-Propagate σχήμα

Επειδή το πρόβλημα προθέματος της δυαδικής άθροισης με διάδοση κρατουμένου υπολογίζει τη δημιουργία και τη διάδοση των σημάτων κρατουμένου, οι ενδιάμεσες μεταβλητές προθέματος μπορούν να έχουν τρεις διαφορετικές τιμές: δημιουργία κρατουμένου 0 (ή «σκότωμα»-kill κρατουμένου 1), δημιουργία κρατουμένου 1, διάδοση του κρατουμένου εισόδου. Έτσι απαιτείται να κωδικοποιήσουμε αυτές τις 3 καταστάσεις με 2 bits. Διαφορετικές κωδικοποιήσεις είναι δυνατές, αλλά συνήθως μία ομάδα δημιουργιών - group generate - $G_{1:k,1}$ και μία ομάδα διαδόσεων-group propagate- $P_{1:k,1}$ χρησιμοποιούνται για να σχηματίσουν το δημιουργία/διάδοση - *generate/propagate* ζεύγος σημάτων $Y_{1:k,1}=(G_{1:k,1}, P_{1:k,1})$ στο επίπεδο 1. Τα αρχικά ζεύγη σημάτων προθέματος $(G_{1:1,0}, P_{1:1,0})$, που αντιστοιχούν στο bit generate g_i και στο bit propagate p_i σήμα, πρέπει να υπολογιστούν από τους τελεστέους εισόδου της άθροισης σε ένα προπαρασκευαστικό βήμα. Αυτή η πράξη συμβολίζεται με τον τελεστή ($\square$) .

$$
\begin{aligned}
g_i &= a_i b_i \\
p_i &= a_i \oplus b_i \quad : \quad i = 0, 1, \ldots, n-1
\end{aligned}
$$

Σύμφωνα με τη διπλανή εξίσωση, τα ζεύγη σημάτων προθέματος του επιπέδου 1 υπολογίζονται

$$
\begin{aligned}
g_i &= a_i\, b_i \\
p_i &= a_i \oplus b_i \\
s_i &= p_i \oplus c_i \\
c_{i+1} &= g_i + p_i\, c_i \quad : \quad i = 0, 1, \ldots, n-1 \\
&\text{where } c_0 = c_{in} \text{ and } c_{out} = c_n
\end{aligned}
$$

από αυτά του l-1 με έναν οποιοδήποτε αλγόριθμο προθέματος χρησιμοποιώντας τη δυαδική πράξη:

$$\begin{aligned}(G^l_{i:k},\ P^l_{i:k}) &= (G^{l-1}_{i:j},\ P^{l-1}_{i:j}) \bullet (G^{l-1}_{j:k},\ P^{l-1}_{j:k}) \\ &= (G^{l-1}_{i:j} + P^{l-1}_{i:j}G^{l-1}_{j:k},\ P^{l-1}_{i:j}P^{l-1}_{j:k})\end{aligned}$$

Τα σήματα generate/propagate από το τελευταίο στάδιο προθέματος ($G_{1:0,m}$,$P_{1:0,m}$) χρησιμοποιούνται για να υπολογίσουμε τα σήματα εισόδου c_i. Τα bits αθροίσματος s_i αποκτώνται από ένα μεταπαρασκευαστικό βήμα που συμβολίζεται με τον τελεστή (⋄). Συνδυάζοντας τις δύο εξισώσεις που συναντήσαμε τελευταίες έχουμε τον επίσημο ορισμό του generate-propagate προβλήματος προθέματος άθροισης:

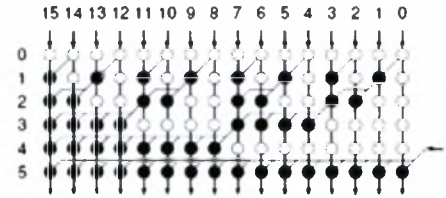
$$\begin{aligned}g_i &= a_i b_i \\ p_i &= a_i \oplus b_i \ : \quad i = 0.1.\ldots.n-1\end{aligned}$$

$$\begin{aligned}(G^0_{i:i},\ P^0_{i:i}) &= (g_i,\ p_i) \\ (G^l_{i:k},\ P^l_{i:k}) &= (G^{l-1}_{i:j} + P^{l-1}_{i:j}G^{l-1}_{j:k},\ P^{l-1}_{i:j}P^{l-1}_{j:k}), \\ &\quad k \le j \le i \ : \quad l = 1.2.\ldots.m\end{aligned}$$

$$c_{i+1} = G^m_{i:0} + P^m_{i:0}c_{in} \ : \quad i = 0.1.\ldots.n-1$$

$$\begin{aligned}s_i &= p_i \oplus c_i \ : \quad i = 0.1.\ldots.n-1 \\ &\text{where } c_0 = c_{in}\end{aligned}$$

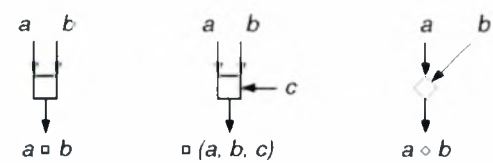
Σημειώσατε πως ένα επιπλέον επίπεδο από τελεστές (•) προστίθεται στο γράφημα για να εξυπηρετήσει το κρατούμενο εισόδου c_{in}. Αυτό κοστίζει επιπλέον hardware αλλά επιτρέπει γρήγορη επεξεργασία του κρατουμένου εισόδου. Για παράδειγμα παρατίθεται ο Sklansky με το επιπλέον επίπεδο επεξεργασίας κρατουμένου:



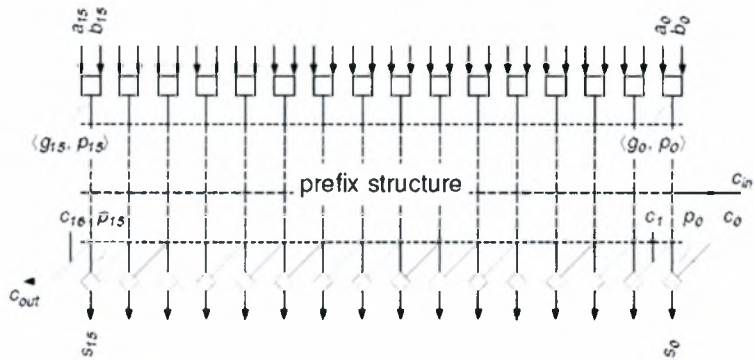
$$
\begin{aligned}
T_{\bullet} &= \log n + 1 \\
A_{\bullet} &= \tfrac{1}{2} n \log n + n \\
A_{\bullet+\circ} &= n \log n + n \\
A_{\bullet} T_{\bullet} &\approx \tfrac{1}{2} n \log^2 n \\
A_{tracks} &= \log n + 1 \\
FO_{max} &= n
\end{aligned}
$$

Ως εναλλακτική λύση, το κρατούμενο εισόδου μπορεί να εισαχθεί στη θέση 0 χρησιμοποιώντας έναν ειδικό τελεστή (◇) 3 εισόδων, αφήνοντας το αρχικό γράφημα παράλληλου προθέματος δίχως αλλαγή (δε χρειαζόμαστε επιπλέον επίπεδο από (•)). Αυτή η λύση επιβάλλει αμελητέα λογική επεξεργασίας κρατουμένου αλλά έχει συγκρίσιμες καθυστερήσεις σημάτων στα σήματα κρατουμένου και αθροίσματος εισόδου. Στη γραφική αναπαράσταση του αλγόριθμου πρέπει να προσθέσουμε μία επιπλέον γραμμή για τον προπαρασκευαστικό τελεστή (□) και το μεταπαρασκευαστικό τελεστή (◇). Στο επόμενο σχήμα φαίνεται ο γράφος ενός γενικού αλγορίθμου άθροισης προθέματος, όπου η δομή προθέματος μπορεί να χρησιμοποιηθεί για την κεντρική μονάδα διάδοσης κρατουμένου. Σημειώστε πως τα σήματα διάδοσης bit p_i πρέπει να δρομολογηθούν μέσω της δομής προθέματος γιατί ξαναχρησιμοποιούνται στο επόμενο βήμα για τον υπολογισμό του bit αθροίσματος. Επίσης σημειώστε την αριστερή ολίσθηση των σημάτων κρατουμένων c_i κατά μία θέση πριν το

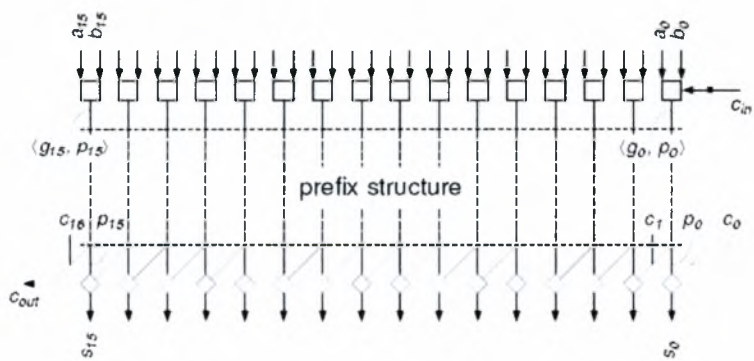
$$
\begin{aligned}
g_0 &= a_0 b_0 + a_0 c_{in} + b_0 c_{in} \\
g_i &= a_i b_i \;:\; i = 1, 2, \ldots, n-1 \\
&\vdots \\
c_{i+1} &= G^m_{i:0} \;:\; i = 0, 1, \ldots, n-1
\end{aligned}
$$



τελικό στάδιο τροποποίησης τάξης μεγέθους. Δύο πιθανότητες υπάρχουν για την επεξεργασία του κρατούμενου εισόδου: μία αργή (b) και μία γρήγορη (a) που απαιτεί ένα επιπλέον επίπεδο προθέματος.



(a)



(b)

Προσέξτε πως τα σήματα διάδοσης $P_{i:0,*}$ που υπολογίζονται στο τελευταίο επίπεδο προθέματος δε χρειάζονται

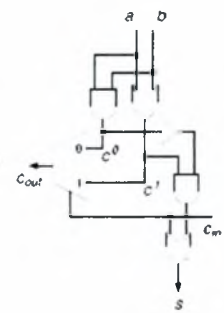
$$g_0 = a_0 b_0 + a_0 c_{in} + b_0 c_{in}$$
$$g_i = a_i b_i \; : \; i = 1, 2, \ldots, n-1$$
$$\vdots$$
$$\vdots$$
$$c_{i+1} = G^m_{i:0} \; : \; i = 0, 1, \ldots, n-1$$

πλέον, αν έχουμε υλοποιήσει τις εξισώσεις:

Έτσι, η AND πύλη του τελεστή (•) της κατώτερης σειρά στο σχήμα για κάθε θέση bit για τον υπολογισμό του $P_{i:0,m}$ δε χρειάζεται και αποσοβείται.

Carry-select σχήμα

Μία εναλλακτική μοντελοποίηση του προβλήματος άθροισης προθέματος βασίζεται στο *carry-select σχήμα* (βλέπε διπλανό σχήμα). Εδώ, η μεταβλητή προθέματος $Y_{i:k,1}$ κωδικοποιείται από τα δύο δυνατά σήματα κρατουμένου $C0_{i:k,1}$,$C1_{i:k,1}$ που υποθέτουν τιμή κρατουμένου 0 και 1 αντίστοιχα.



$$
\begin{aligned}
c_i^0 &= a_i b_i \\
c_i^1 &= a_i + b_i \\
p_i &= a_i \oplus b_i \ : \quad i = 0, 1, \ldots, n-1
\end{aligned}
$$

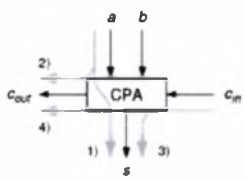
$$
\begin{aligned}
(C0_{i:k}^0,\ C1_{i:k}^0) &= (c_i^0,\ c_i^1) \\
(C0_{i:k}^l,\ C1_{i:k}^l) &= (C0_{i:j+1}^{l-1}\overline{C0}_{j:k}^{l-1} + C1_{i:j+1}^{l-1}C0_{j:k}^{l-1}, \\
&\quad\ C0_{i:j+1}^{l-1}\overline{C1}_{j:k}^{l-1} + C1_{i:j+1}^{l-1}C1_{j:k}^{l-1}), \\
&\quad\ k \le j \le i \ : \quad l = 1, 2, \ldots, m \\
c_{i+1} &= C0_{i:0}^m \overline{c}_{in} + C1_{i:0}^m c_{in} \ : \quad i = 0, 1, \ldots, n-1
\end{aligned}
$$

$$
\begin{aligned}
s_i &= p_i \oplus c_i \ : \quad i = 0, 1, \ldots, n-1 \\
&\text{where } c_0 = c_{in}
\end{aligned}
$$

Βασικά το generate-propagate και το carry-select σχήμα είναι ισοδύναμα και χρησιμοποιούνται οι ίδιοι αλγόριθμοι προθέματος. Το τελευταίο σχήμα όμως, παίζει μικρό ρόλο στη σχεδίαση με χρήση κελιών γιατί οι μαύροι κόμβοι του συντίθεται από δύο πολυπλέκτες αντί για τον πιο επαρκή συνδυασμό AND-OR/END που χρησιμοποιείται στο generate-propagate σχήμα.

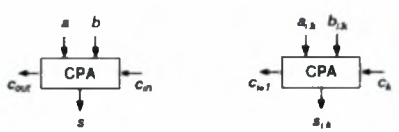
Βασικές τεχνικές επιτάχυνσης της άθροισης

1) $T_{CPA}(a,b \rightarrow s)$
2) $T_{CPA}(a,b \rightarrow c_{out})$
3) $T_{CPA}(c_{in} \rightarrow s)$
4) $T_{CPA}(c_{in} \rightarrow c_{out})$



Οι αθροιστές διάδοσης κρατουμένου που χρησιμοποιούν τον απλό αλγόριθμο ripple-carry είναι πολύ αργοί για τις περισσότερες εφαρμογές. Διάφορες τεχνικές επιτάχυνσης υπάρχουν, οι οποίες μειώνουν το χρόνο υπολογισμού με κάποιο βαθμό παραλληλίας στο κόστος επιπλέον hardware. Οι αρχές αυτές θα συνοψιστούν στο υποκεφάλαιο αυτό.

CPA και μερικός CPA



Ένας CPA υπολογίζει το άθροισμα των δύο τελεστέων εισόδου ενώ ένας *μερικός CPA* αθροίζει μόνο μερικά από τα bits των τελεστέων, που σημειώνονται ως $a_{i:k}$, $b_{i:k}$. Αρχικά πρέπει να διακρίνουμε τα τέσσερα πιθανά μονοπάτια διάδοσης σημάτων στον CPA.

Κρίσιμα μονοπάτια: Κανένα Στις χαμηλής ταχύτητας εφαρμογές, όλα τα μονοπάτια σημάτων είναι μη κρίσιμα.

Κρίσιμα μονοπάτια: Όλα Στις εφαρμογές όπου όλοι οι χρόνοι άφιξης των σημάτων σε όλες τις εισόδους είναι ίσοι και όλες οι έξοδοι του CPA αναμένονται να είναι έτοιμες την ίδια χρονική στιγμή(για παράδειγμα όταν ο CPA είναι το μόνο συνδυαστικό block ανάμεσα σε δύο καταχωρητές ή όταν η περιβάλλουσα λογική έχει ισορροπημένες καθυστερήσεις σήματος), όλα τα μονοπάτια σημάτων είναι υποκείμενα στις ίδιες χρονικές καθυστερήσεις και άρα είναι ομοίως κρίσιμα.

Κρίσιμα μονοπάτια : 2,4 Διάφορες εφαρμογές ζητούν γρήγορο κρατούμενο εξόδου επειδή αυτό το σήμα ελέγχει κάποια ακόλουθη συνδυαστική λογική, για παράδειγμα το carry flag στις ALUs.

Κρίσιμα μονοπάτια : 3,4 Άλλες εφαρμογές απαιτούν γρήγορη διάδοση κρατουμένου εισόδου εξαιτίας ενός καθυστερημένου κρατουμένου εισόδου που ήρθε στον CPA.

Κρίσιμα μονοπάτια : 4 Τέλος, μπορεί να χρειαστεί γρήγορη διάδοση του κρατουμένου εισόδου στο κρατούμενο εξόδου. Οι μερικοί CPAs με αργοπορημένο κρατούμενο εισόδου και γρήγορες ιδιότητες κρατουμένου εξόδου μπορούν να χρησιμοποιηθούν για να επιταχύνουν μεγαλύτερους CPAs.

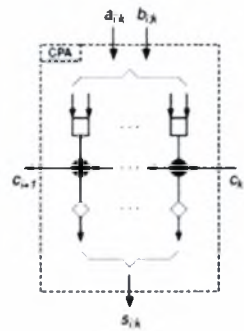
Κρίσιμα μονοπάτια : Ξεχωριστά bits Στις παραπάνω περιπτώσεις υποθέσαμε πως όλα τα bits των τελεστέων καταφθάνουν με ίσους χρόνους άφιξης. Σε μερικές εφαρμογές, όμως, αυτό δε συμβαίνει. Και έτσι τα κρίσιμα μονοπάτια καταλήγουν να διαφέρουν κατά πολύ και να δίνουν πολύπλοκες απαιτήσεις χρονισμού.

Τα βασικά σχήματα για την κατασκευή και επιτάχυνση των CPAs μπορούν να διαιρεθούν σε επιπέδου bit και επιπέδου block σχήματα.

Bit-level ή Ευθεία CPA σχήματα

Οι αθροιστές που χρησιμοποιούν ευθεία CPA σχήματα υλοποιούν τις λογικές εξισώσεις της άθροισης σε επίπεδο bit όπως είναι. Συνεπώς, χτίζονται από φέτες bit που περιέχουν τους τελεστές (•), (□) και (◇) όπου μερικοί αλγόριθμοι προθέματος χρησιμοποιούνται για διάδοση κρατουμένου.

Ripple-carry ή serial-prefix σχήμα

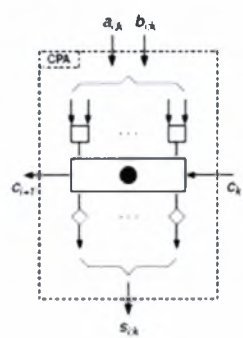


Το *ripple-carry σχήμα* άθροισης χρησιμοποιεί τον serial-prefix αλγόριθμο για τη διάδοση κρατουμένου.

Ιδιότητες

- Ελάχιστη συνδυαστική δομή αθροιστή, ελάχιστο κόστος σε hardware(O(n)).
- Η πιο αργή δομή αθροιστή O(n)
- Χρησιμοποιείται ως βασικός μερικός CPA σε άλλες δομές άθροισης.

Carry-lookahead ή parallel-prefix σχήμα



Ένας parallel-prefix αλγόριθμος μπορεί να χρησιμοποιηθεί για γρήγορη διάδοση κρατουμένου. Έχει ως αποτέλεσμα το σχήμα άθροισης *parallel-prefix ή carry-lookahead*, αφού όλα τα κρατούμενα

προϋπολογίζονται ("looked-ahead") για τον τελικό υπολογισμό των bits αθροίσματος.

Στους παραδοσιακούς αθροιστές carry-lookahead, τα κρατούμενα ομάδων των 4 bits υπολογίζονται παράλληλα σύμφωνα με τις παρακάτω εξισώσεις:

$$
\begin{aligned}
c_{i+1} &= g_i + c_i p_i \\
c_{i+2} &= g_{i+1} + g_i p_{i+1} + c_i p_i p_{i+1} \\
c_{i+3} &= g_{i+2} + g_{i+1} p_{i+2} + g_i p_{i+1} p_{i+2} + c_i p_i p_{i+1} p_{i+2} \\
c_{i+4} &= g_{i+3} + g_{i+2} p_{i+3} + g_{i+1} p_{i+2} p_{i+3} \\
&\quad + g_i p_{i+1} p_{i+2} p_{i+3} + c_i p_i p_{i+1} p_{i+2} p_{i+3}
\end{aligned}
$$

Διάφορες από τις 4μπιτες αυτές δομές μπορούν να διαταχθούν γραμμικά ή ιεραρχικά ώστε να υλοποιήσουν δομές carry-lookahead για μεγαλύτερο μήκος λέξης. Είναι βασικά μία βαριάντα του σχήματος parallel-prefix.

Ιδιότητες

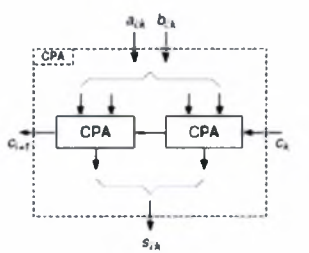
- Αυξημένο κόστος σε hardware O(nlogn)
- Επιτάχυνση όλων των μονοπατιών σήματος O(logn)
- Trade-off μεταξύ επιτάχυνσης και επιπλέον υλικού υπάρχει χρησιμοποιώντας διάφορους αλγόριθμους προθέματος.

Block-level ή Σύνθετα CPA σχήματα

Η άλλη κλάση σχημάτων άθροισης βασίζεται στην επιτάχυνση διάδοσης κρατουμένου ήδη υπαρκτών μερικών CPAs και συνδυάζοντας διάφορους από αυτούς ώστε να σχηματίσει γρηγορότερους και μεγαλύτερους αθροιστές. Αυτοί οι αθροιστές απαρτίζονται από έναν ή περισσότερους CPAs και κάποια επιπλέον λογική. Λειτουργούν σε block-level αφού τα bits επεξεργάζονται σε ομάδες από τους περιεχόμενους CPAs. Μία διάκριση ανάμεσα στα αλυσιδωτά και στα επιταχυντικά σχήματα μπορεί εύκολα να γίνει: τα πρώτα χρησιμοποιούνται για να χτίσουμε μεγαλύτερους αθροιστές από μικρότερους ενώ τα δεύτερα επιταχύνουν την επεξεργασία μιας δεδομένης ομάδας bits.

Ripple-Carry σχήμα

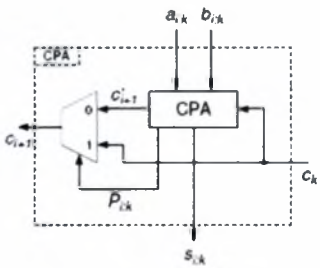
Το ripple-carry σχήμα σε επίπεδο block είναι το βασικό αλυσιδωτό σχήμα κατασκευής μεγαλύτερων CPAs από οποιουσδήποτε μικρότερους CPAs. Αυτό γίνεται συνενώνοντας CPAs στη σειρά έτσι ώστε ένα κρατούμενο να μεταφέρεται μεταξύ της ακολουθίας των μερικών CPAs.



Ιδιότητες

- Συνένωση CPAs

Carry-skip σχήμα



Ο υπολογισμός του κρατουμένου για μία μοναδική θέση bit μπορεί να μοντελοποιηθεί για ένα ολόκληρο CPA(ομάδα bits)(βλέπε δίπλα), όπου $P_{i:k}$ σημαίνει τη διάδοση ομάδας του CPA και λειτουργεί ως σήμα επιλογής στη δομή του πολυπλέκτη. c' είναι το κρατούμενο εξόδου του μερικού αθροιστή (βλέπε δίπλα).

$$c_{i+1} = \overline{p_i} g_i + p c_i$$

$$c_{i+1} = \overline{P_{i:k}} c'_{i+1} + P_{i:k} c_k$$

Δύο περιπτώσεις:

$P_{i:k} = 0$: Το κρατούμενο c'_{i+1} δημιουργείται από το CPA και επιλέγεται από τον πολυπλέκτη ως c_{i+1} .Το κρατούμενο εισόδου c_k **δε** διαδίδεται διαμέσου του CPA στο κρατούμενο εξόδου c_{i+1}.

$P_{i:k} = 1$: το κρατούμενο εισόδου c_k διαδίδεται διαμέσου του CPA στο c'_{i+1} αλλά δεν επιλέγεται από τον πολυπλέκτη. «Πηδά»(skips) τον CPA και απευθείας επιλέγεται ως κρατούμενο εξόδου c_{i+1}. Έτσι, το συνδυαστικό μονοπάτι από το κρατούμενο εισόδου στο κρατούμενο εξόδου διαμέσου του CPA δεν ενεργοποιείται ποτέ.

Με άλλα λόγια, το αργό μονοπάτι από το κρατούμενο εισόδου στο κρατούμενο εξόδου διαμέσου σου CPA διασπάται είτε από τον ίδιο το CPA είτε από τον πολυπλέκτη. Το επακόλουθο carry-skip block άθροισης είναι ένας CPA με μικρό και σταθερό $T_{CPA}(c_k \rightarrow c_{i+1})$, που μπορεί να χρησιμοποιηθεί για να επιταχύνουμε τη διάδοση κρατουμένου. Συντίθεται από έναν οποιοδήποτε CPA με ομαδική διάδοση εξόδου και ένα 2-1 πολυπλέκτη.

Στη βιβλιογραφία, αναφέρεται πως μπορεί να χρησιμοποιηθεί μία OR πύλη αντί του πολυπλέκτη. Αυτό όμως θα επιταχύνει μόνο τις μεταβάσεις

0->1 μεταβάσεις στο μονοπάτι του κρατουμένου. Για τις ανάποδες μεταβάσεις ο αθροιστής λειτουργεί ως ripple-carry. Οι εφαρμογές περιορίζονται σε υλοποιήσεις που χρησιμοποιούν δυναμική λογική.

Δε μελετάμε τις ιδιότητες πλεονασμού πληροφορίας που έχει το παρόν σχήμα.

Ιδιότητες

- Σταθερή καθυστέρηση T_{CPA} (c_k -> c_{i+1})
- Εγγενής λογικός πλεονασμός
- Μικρό επιπλέον κόστος σε hardware: ομαδική λογική διάδοσης και μικρός πολυπλέκτης
- Μέτριο επιπλέον κόστος σε hardware για την έκδοση χωρίς πλεόνασμα: διπλή αλυσίδα διάδοσης κρατουμένου.

Carry-select σχήμα

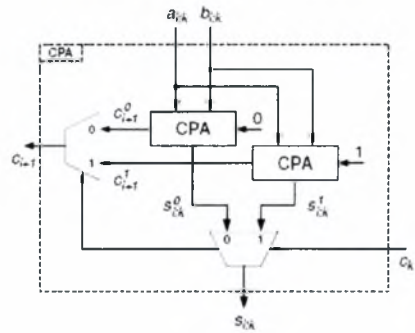
Το βασικό πρόβλημα που αντιμετωπίζεται στην επιτάχυνση της διάδοσης κρατουμένου είναι η γρήγορη επεξεργασία ενός καθυστερημένου κρατουμένου εισόδου. Εφόσον αυτό το κρατούμενο εισόδου μπορεί να έχει δύο μόνο τιμές, τα δύο πιθανά αποτελέσματα άθροισης($s_{i:k,0}$, $c_{i+1,0}$ και $s_{i:k,1}$, $c_{i+1,1}$ αντίστοιχα) μπορούν να προϋπολογιστούν και να επιλεγούν έπειτα από το αργοπορημένο κρατούμενο εισόδου c_k χρησιμοποιώντας μικρό και σταθερό χρόνο. Το επακόλουθο carry-select σχήμα άθροισης χρειάζεται δύο CPAs, ένα για κάθε πιθανό αποτέλεσμα και έναν 2-1 πολυπλέκτη για κάθε bit αθροίσματος και για το κρατούμενο.

$$s_{i:k} = \overline{c_k} s^0_{i:k} + c_k s^1_{i:k}$$

$$c_{i+1} = \overline{c_k} c^0_{i+1} + c_k c^1_{i+1}$$

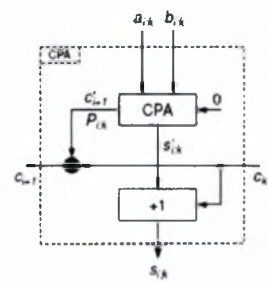
Ιδιότητες

- Σταθερός χρόνος καθυστέρησης $T_{CPA}(c_k \rightarrow c_{i+1})$ και $T_{CPA}(c_k \rightarrow s_{i:k})$.
- Υψηλό επιπλέον κόστος σε hardware: διπλοί CPA και πολυπλέκτες.



Carry-Increment σχήμα

Στο Carry-increment σχήμα άθροισης μόνο ένα αποτέλεσμα με κρατούμενο εισόδου 0 προϋπολογίζεται ($s'_{i:k}$) και αυξάνεται κατά 1 έπειτα, αν c_k=1. Το κρατούμενο εξόδου c_{i+1} υπολογίζεται από το κρατούμενο εξόδου του CPA c'_{i+1} και την ομάδα διάδοσης $P_{i:k}$ χρησιμοποιώντας τον τελεστή (•) της δυαδικής άθροισης. Στις διπλανές εξισώσεις $c'_{i+1} = G_{i:k}$ από τη στιγμή που το κρατούμενο εισόδου στο CPA είναι 0. Το απαιτούμενο κύκλωμα incrementer παρέχει σταθερό χρόνο διάδοσης κρατουμένου και είναι πολύ πιο φθηνό από το επιπλέον υλικό που χρειάζεται στο carry-select σχήμα. Επίσης η λογική του CPA και ο incrementer μπορούν να συνενωθούν ως ένα βαθμό.



$$
\begin{aligned}
s_{i:k} &= s'_{i:k} + c_k \\
c_{i+1} &= G_{i:k} + P_{i:k} c_k \\
&= c'_{i+1} + P_{i:k} c_k
\end{aligned}
$$

Ιδιότητες

- Σταθερός χρόνος καθυστέρησης T_{CPA} (c_k->c_{i+1}) και T_{CPA}(c_k->$s_{i:k}$)
- Μέτριο κόστος επιπλέον hardware: incrementer, λογική διάδοσης ομάδας και τελεστής (•).

Σύνοψη σχημάτων άθροισης σύμφωνα με τα χαρακτηριστικά επιτάχυνσης

	ripple	skip	select	increment	look-ahead
$T_{CPA}(c_{in} \rightarrow c_{out})$		√	√	√	√
$T_{CPA}(c_{in} \rightarrow s_i)$			√	√	√
$T_{CPA}(a_i, b_i \rightarrow c_{out})$					√
$T_{CPA}(a_i, b_i \rightarrow s_i)$					√

Ο παραπάνω πίνακας συνοψίζει τα βασικά σχήματα επιτάχυνσης άθροισης και τα χαρακτηριστικά τους. Το block-level ripple-carry σχήμα είναι ο φυσικός και μόνος τρόπος να συνθέσουμε μεγαλύτερους αθροιστές από μερικούς CPAs διαδίδοντας το κρατούμενο από το χαμηλότερο στο υψηλότερο επίπεδο bit-group(σχήμα συνένωσης). Όλα τα σύνθετα σχήματα επιτάχυνσης (skip, select, increment) παρέχουν μόνο επιταχύνσεις διάδοσης σε μονοπάτια σήματος που ξεκινούν από το κρατούμενο εισόδου. Μπορούν να χρησιμοποιηθούν είτε για εφαρμογές αθροιστών με αργοπορημένα κρατούμενα εισόδου είτε με τον κατάλληλο συνδυασμό για γρήγορους CPAs. Το σχήμα carry-lookahead είναι το μόνο σχήμα άθροισης που παρέχει μία επιτάχυνση σε όλα τα μονοπάτια σημάτων χωρίς να στηρίζεται στη σύνθεση διαφόρων σχημάτων(είναι δηλαδή ευθύ σχήμα επιτάχυνσης).

Αρχιτεκτονικές Αθροιστών

Η επαρκής υλοποίηση κυκλωμάτων αθροιστών δε βασίζεται μόνο στα υπολογιστικά σχήματα αλλά συμπεριλαμβάνει επίσης κα πιθανές κυκλωματικές απλοποιήσεις και βελτιστοποιήσεις.

Η κυκλωματική δομή κάθε αρχιτεκτονικής θα δίνεται από το σύνολο των λογικών εξισώσεων για κάθε επίπεδο bit. Μέγιστα μεγέθη αθροιστών και ομάδων εισόδων για κάθε αθροιστή. Τέλος, ακριβείς πολυπλοκότητες χρόνου και χώρου δίνονται για κάθε αρχιτεκτονική βασισμένες στο μοντέλο μονάδας-πύλης. Δε θα παραθέσουμε όλες τις πιθανές υλοποιήσεις, ειδικά για τα κυκλώματα τα οποία δε χρησιμοποιήσαμε.

Ripple-Carry Αθροιστής (RCA ή RPL)

Ο RCA συντίθεται από μία σειρά από πλήρεις αθροιστές(fa από το full adders), όπου ο αρχικός πλήρης αθροιστής (ifa από το initial full adder) μπορεί να χρησιμοποιήσει μία πύλη πλειοψηφίας για γρήγορο υπολογισμό κρατουμένου. Δίνονται οι αντίστοιχες λογικές εξισώσεις, μεγέθη αθροιστών και μέτρα πολυπλοκότητας. Ο πίνακας για τα μεγέθη αθροιστών δίνει το μέγιστο αριθμό bits n που μπορούμε να υπολογίσουμε σε δεδομένο χρόνο.

ifa	$c_1 = a_0b_0 + a_0c_0 + b_0c_0$ $s_0 = a_0 \oplus b_0 \oplus c_0$	$A = 9$
fa	$g_i = a_ib_i$ $p_i = a_i \oplus b_i$ $c_{i+1} = g_i + p_ic_i$ $s_i = p_i \oplus c_i$	$A = 7$

T	4	4	8	16	32	64	128	256	512
n	1	2	4	8	16	32	64	128	256

$$T_{RCA} = 2n$$
$$A_{RCA} = 7n + 2$$

Carry-Skip αθροιστής (CSKA)

Σύνθεση του σχήματος συνένωσης(concatenation) και του σχήματος carry-skip επιτυγχάνουν τον CSA. Παρατίθεται μία υλοποίησή του για το πλήρες της αναφοράς:

1-level CSA με πλεονασμό (CSKA-1L)

Ο CSKA-1L συντίθεται με σειρές ομάδων skipping και έναν αρχικό πλήρη αθροιστή (ifa από το initial full adder) στο LSB(Least Significant Bit). Κάθε ομάδα skipping αποτελείται από σειρές πλήρων αθροιστών (bfa από το block full adder) με επιπρόσθετη διάδοση ομάδας σήματος δημιουργίας (P_i), έναν αρχικό πλήρη αθροιστή (bifa από το block initial full adder) στο LSB της κάθε ομάδας και τέλος έναν δημιουργό κρατουμένου (carry-generator)(bcg-block carry generator) στο MSB (Most Significant Bit) της ομάδας. Cpb(carry-out of previous block) και ctb(carry-out of this block) σημαίνουν τα κρατούμενα εξόδου του προηγούμενου και του παρόντος block(ομάδας), αντίστοιχα.

Η μέγιστη ταχύτητα επιτυγχάνεται από τη ρύθμιση του μεγέθους κάθε ομάδας bits ξεχωριστά. Επειδή το skipping σχήμα επιταχύνει μόνο το $T_{CPA}(C_k \to C_i)$ και όχι το $T_{CPA}(C_k \to S_i)$, η δημιουργία του κρατουμένου ξεκινά και η αναδιανομή του κρατουμένου τελειώνει σε αργά ripple-carry blocks. Ως φυσική συνέπεια, οι ομάδες χαμηλά και ψηλά είναι μικρότερες αντίθετα με τις ομάδες στη μέση. Αφού η καθυστέρηση

ifa	$c_{tb} = a_0b_0 + a_0c_0 + b_0c_0$ $s_0 = a_0 \oplus b_0 \oplus c_0$	$A = 9$
bifa	$p_i = a_i \oplus b_i$ $P_i = p_i$ $c_{i+1} = a_ib_i + a_ic_{pb} + b_ic_{pb}$ $s_i = p_i \oplus c_{pb}$	$A = 9$
bfa	$g_i = a_ib_i$ $p_i = a_i \oplus b_i$ $P_i = p_iP_{i-1}$ $c_{i+1} = g_i + p_ic_i$ $s_i = p_i \oplus c_i$	$A = 8$
bcg	$c_{tb} = \overline{P_ic_{i+1}} + P_ic_{pb}$	$A = 3$

διαμέσου ενός πλήρους αθροιστή ισούται με την καθυστέρηση ενός πολυπλέκτη ως προς το μέτρο του unit-gate μοντέλου, γειτονικές ομάδες διαφέρουν στο μέγεθος κατά μόνο ένα bit. κ είναι το μέγεθος της μεγαλύτερης ομάδας.

T	4	4	8	10	12	14	16	18	20	22	24	26	28	30	32	···	46
k			2	2	3	3	4	4	5	5	6	6	7	7	8	···	11
n	1	2	5	7	10	13	17	21	26	31	37	43	50	57	65	···	133

Υπάρχουν επίσης τα εξής μοντέλα CSKA:

- 1-level carry-skip αθροιστής χωρίς πλεονασμό (CSKA-1L')
- 2-level carry-skip αθροιστής (CSKA-2L)

$$T_{CSKA-1L} = 4k$$
$$A_{CSKA-1L} = 8n + 6k - 6$$
$$k = \lceil \sqrt{n-1} \rceil$$

Carry-Select αθροιστής (CSLA)

Ένας CSLA είναι ο συνδυασμός των σχημάτων της συνένωσης και της επιλογής. Κάθε θέση bit συμπεριλαμβάνει τη δημιουργία δύο αθροισμάτων ($s_{i,0}$, $s_{i,1}$) και κρατουμένων ($c_{i,0}$, $c_{i,1}$) και τους πολυπλέκτες επιλογής για το σωστό bit αθροίσματος. Το σωστό bit κρατουμένου επιλέγεται στο τέλος της ομάδας (group/block) (bcg block carry generation)

Αφού τα μονοπάτια σήματος $T_{CPA}(c_k \rightarrow c_i)$ και $T_{CPA}(c_k \rightarrow s_i)$ επιταχύνονται από το σχήμα επιλογής, οι ομάδες μπορούν να γίνουν μεγαλύτερες προς το MSB.

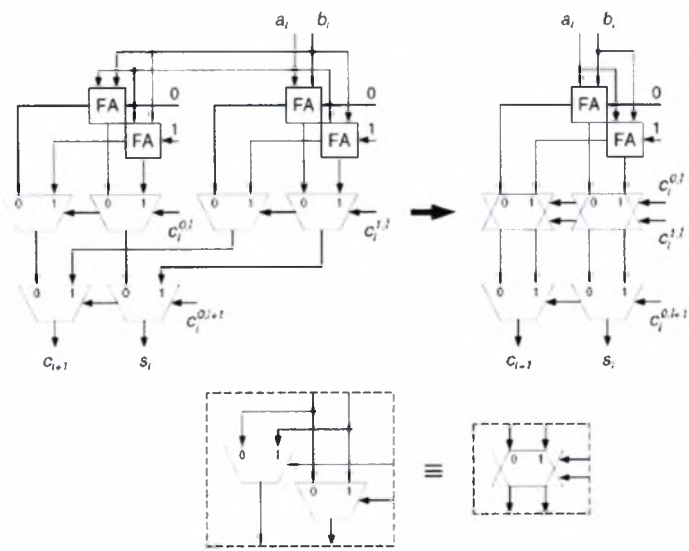
ifa	$c_{th} = a_0 b_0 + a_0 c_0 + b_0 c_0$ $s_0 = a_0 \oplus b_0 \oplus c_0$	$A = 9$
biha	$g_i = a_i b_i$ $p_i = a_i \oplus b_i$ $c_{i+1}^0 = g_i$ $c_{i+1}^1 = g_i + p_i$ $s_i^0 = p_i$ $s_i^1 = \overline{p}_i$ $s_i = \overline{c}_{ph} s_i^0 + c_{ph} s_i^1$	$A = 7$
bfa	$g_i = a_i b_i$ $p_i = a_i \oplus b_i$ $c_{i+1}^0 = g_i + p_i c_i^0$ $c_{i+1}^1 = g_i + p_i c_i^1$ $s_i^0 = p_i \oplus c_i^0$ $s_i^1 = p_i \oplus c_i^1$ $s_i = \overline{c}_{ph} s_i^0 + c_{ph} s_i^1$	$A = 14$
bcg	$c_{th} = c_{i+1}^0 + c_{ph} c_{i+1}^1$	$A = 2$

$$T_{CSLA} = 2k + 2$$
$$A_{CSLA} = 14n - 5k - 5$$
$$k = \lceil \tfrac{1}{2}\sqrt{8n-7} - \tfrac{1}{2} \rceil$$

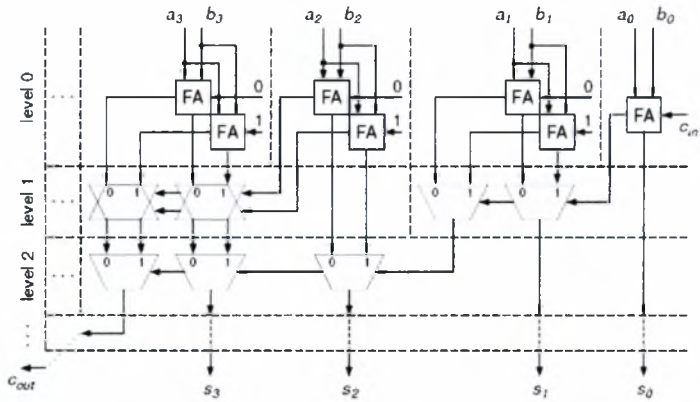
T	4	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34
k		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
n	1	2	4	7	11	16	22	29	37	46	56	67	79	92	106	121	139

Conditional-Sum Αθροιστής (COSA)

Στον CSLA όμως, μπορούμε να έχουμε πολλαπλή εφαρμογή του σχήματος επιλογής. Επειδή το σχήμα επιλογής στηρίζεται στο διπλασιασμό των μερικών CPA, το επιπλέον hardware σε πολυεπίπεδες δομές γίνεται απαγορευτικά μεγάλο εξαιτίας των επαναλαμβανόμενων διπλασιασμών. Όμως, αφού και τα δύο αποτελέσματα κρατουμένων είναι διαθέσιμα σε κάθε επίπεδο, μόνο οι πολυπλέκτες (αντί για όλο το CPA) πρέπει να διπλασιαστούν ώστε να πάρουμε ένα επιπλέον επίπεδο επιλογής.



Ένας CSLA με μέγιστο αριθμό επιπέδων (=logn) που χρησιμοποιεί ταυτόχρονα το παραπάνω σχήμα απλοποίησης λέγεται conditional-sum αθροιστής(COSA). Τα μεγέθη ομάδων ξεκινούν με ένα bit στο χαμηλότερο επίπεδο και διπλασιάζονται σε κάθε επιπλέον επίπεδο. Ο τύπος της λογικής οργανώνεται σε επίπεδο αντί για ομάδες bits. Στο πρώτο επίπεδο (csg) και τα δύο πιθανά κρατούμενα αλλά και τα αθροίσματα δημιουργούνται για κάθε θέση bit ($c_{1,0,0}$, $c_{1,1,0}$, $s_{1,0,0}$ $s_{1,1,0}$). Τα επόμενα επίπεδα επιλέγουν νέα ζεύγη από bits κρατουμένων και αθροισμάτων ($c_{1,0,0}$, $c_{1,1,0}$, $s_{1,0,0}$ $s_{1,1,0}$) για αυξανόμενα μεγαλύτερες ομάδες bits (ssl, csl). Το τελευταίο επίπεδο εκτελεί τελική επιλογή κρατουμένου και άθροισμα του bit επιλογής (fssl, fcsl).



csg	$c_{i+1}^{0,0} = a_i b_i$ $c_{i+1}^{1,0} = a_i + b_i$ $s_i^{0,0} = a_i \oplus b_i$ $s_i^{1,0} = \overline{a_i \oplus b_i}$	$A = 6$
ssl	$s_i^{0,l+1} = s_i^{0,l} c_j^{0,l} + s_i^{1,l} c_j^{0,l}$ $s_i^{1,l+1} = s_i^{0,l} c_j^{1,l} + s_i^{1,l} c_j^{1,l}$	$A = 6$
csl	$c_i^{0,l+1} = c_i^{0,l} + c_i^{1,l} c_j^{0,l}$ $c_i^{1,l+1} = c_i^{0,l} + c_i^{1,l} c_j^{1,l}$	$A = 4$
fssl	$s_i = s_i^{0,m} \overline{c}_0 + s_i^{1,m} c_0$	$A = 3$
fcsl	$c_{i+1}^{l} = c_i^{0,l} + c_i^{1,l} c_0$	$A = 2$

T	4	4	6	8	10	12	14	16	18
m		1	2	3	4	5	6	7	8
n	1	2	4	8	16	32	64	128	256

$$T_{\mathrm{COSA}} = 2 \log n + 2$$
$$A_{\mathrm{COSA}} = 3n \log n + 7n - 2 \log n - 7$$

Carry-Increment αθροιστής (CIA)

Για τον αθροιστή αυτό δε χρειάστηκε να ασχοληθούμε αναλυτικά. Είναι το αποτέλεσμα συνδυασμού του σχήματος συνένωσης και αυτού της προσαύξησης(concatenation και incrementation schemes). Οι επαναλαμβανόμενες εφαρμογές του σχήματος αυτού δημιουργούν πολυεπίπεδους αθροιστές carry-increment.

Μπορεί να παρατηρηθεί πως οι CIAs έχουν τη βασική αθροιστική δομή των παράλληλου προθέματος ή πρόβλεψης κρατουμένου αθροιστών, στο ότι αποτελούνται από ένα προπαρασκευαστικό, ένα διάδοσης κρατουμένου και ένα μεταπαρασκευαστικό στάδιο. Με μεγαλύτερη μελέτη μπορούμε να αποκαλύψουμε πως οι CIAs ανήκουν στην οικογένεια των αθροιστών παράλληλου προθέματος.

Parallel-Prefix/Carry-Lookahead Αθροιστές (PPA/CLA)

Οι αθροιστές *parallel-prefix* χρησιμοποιούν το ευθύ σχήμα παράλληλου προθέματος για το γρήγορο υπολογισμό του κρατουμένου. Λέγονται επίσης και *αθροιστές πρόβλεψης κρατουμένου* (carry-lookahead adders). Παρά τις διάφορες εκδοχές που υπάρχουν σε αυτή την κατηγορία, έχουν όλοι την αρχική δημιουργία των σημάτων generate και propagate (igpg, gpg) και την τελική δημιουργία του bit αθροίσματος(sg) και διαφέρουν μόνο στη διάταξη των m ενδιάμεσων επιπέδων δημιουργίας κρατουμένων (cg).

Συνήθως, δυαδικές ή αρχιτεκτονικές των 2 bits χρησιμοποιούνται, δηλαδή ο τελεστής προθέματος επεξεργάζεται 2 bits, ή με άλλα λόγια χρησιμοποιείται μέγεθος block 2 bits στο πρώτο επίπεδο.

Γενικές Λογικές εξισώσεις

igpg	$g_0^0 = a_0b_0 + a_0c_0 + b_0c_0$ $p_0^0 = a_0 \oplus b_0$	$A = 7$
gpg	$g_i^0 = a_ib_i$ $p_i^0 = a_i \oplus b_i$	$A = 3$
cg	$g_i^{l+1} = g_i^l + p_i^l g_j^l$ $p_i^{l+1} = p_i^l p_j^l$	$A = 3$
	$c_{i+1} = g_i^m$	
sg	$s_i = p_i^0 \oplus c_i$	$A = 2$

Sklansky parallel-prefix algorithm (PPA-SK)

T	4	6	8	10	12	14	16	18	20
m		1	2	3	4	5	6	7	8
n	1	2	4	8	16	32	64	128	256

$$T_{PPA-SK} = 2\log n + 4$$
$$A_{PPA-SK} = \frac{3}{2} n \log n + 4n + 5$$

Brent-Kung parallel-prefix algorithm (PPA-BK)

T	4	6	8	12	16	20	24	28	32
m		1	2	4	6	8	10	12	14
n	1	2	4	8	16	32	64	128	256

$$T_{PPA-BK} = 4\log n$$
$$A_{PPA-BK} = 10n - 3\log n - 1$$

Kogge-Stone parallel-prefix algorithm (PPA-KS)

T	4	6	8	10	12	14	16	18	20
m		1	2	3	4	5	6	7	8
n	1	2	4	8	16	32	64	128	256

$$T_{PPA-KS} = 2\log n + 4$$
$$A_{PPA-KS} = 3n \log n + n + 8$$

Αθροιστές παράλληλου προθέματος πολλών bit

Ο τελεστής προθέματος για την δυαδική άθροιση μπορεί να προσαρμοστεί ώστε να επεξεργάζεται περισσότερα bits σε μία χρονική στιγμή (δηλαδή μέγεθος block μεγαλύτερο του 2). Η αντίστοιχη λογική γίνεται περισσότερο περίπλοκη, αλλά ο επακόλουθος αλγόριθμός αριθμεί λιγότερα επίπεδα.

Ο κλασικός carry-lookahead αθροιστής (CLA) που περιγράφεται στη βιβλιογραφία είναι στην πραγματικότητα ένας 4-bit Brent-Kung παράλληλου προθέματος αθροιστής. Εδώ, δύο φάσεις για τη διάδοση κρατουμένου μπορούν να διακριθούν: σε πρώτη φάση (cg1) τα bits κρατουμένων για κάθε τέταρτη θέση bit υπολογίζονται. Σε δεύτερη φάση (cg2) υπολογίζονται όλα τα υπόλοιπα κρατούμενα από τα κρατούμενα της φάσης 1.

igpg	$g_0^0 = a_0 b_0 + a_0 c_0 + b_0 c_0$ $p_0^0 = a_0 \oplus b_0$	$A = 7$
gpg	$g_i^0 = a_i b_i$ $p_i^0 = a_i \oplus b_i$	$A = 3$
cg1	$g_{i3}^{l+1} = g_{i3}^l + p_{i3}^l g_{i2}^l + p_{i3}^l p_{i2}^l g_{i1}^l + p_{i3}^l p_{i2}^l p_{i1}^l g_{i0}^l$ $p_{i3}^{l+1} = p_{i3}^l p_{i2}^l p_{i1}^l p_{i0}^l$	$A = 12$
	$c_{i+1} = g_i^{m/4}$	
cg2	$g_{i2}^{l+1} = g_{i2}^l + p_{i2}^l g_{i1}^l + p_{i2}^l p_{i1}^l g_{i0}^l + p_{i2}^l p_{i1}^l p_{i0}^l c_{i0}$ $g_{i1}^{l+1} = g_{i1}^l + p_{i1}^l g_{i0}^l + p_{i1}^l p_{i0}^l c_{i0}$ $g_{i0}^{l+1} = g_{i0}^l + p_{i0}^l c_{i0}$	$A = 16$
	$c_{i+1} = g_i^m$	
sg	$s_i = p_i^0 \oplus g_{i-1}^l$	$A = 2$

T	4	6	8	12	16	20	24	28	32
m		1	1	2	3	4	5	6	7
n	1	2	4	8	16	32	64	128	256

$$T_{CLA} = 4 \log n$$
$$A_{CLA} = 14n - 20$$

Υβριδικές αρχιτεκτονικές αθροιστών

Έως τώρα παρουσιάσαμε ξεκάθαρες αρχιτεκτονικές, δηλαδή δεν είχαμε ανακάτεμα διάφορων σχημάτων. Η φύση των περισσότερων σχημάτων επιτάχυνσης επιτρέπει τον οποιοδήποτε συνδυασμό τους. Εξαιτίας όμως της αντιπροσώπευσης των ιδιαιτεροτήτων κάθε αρχιτεκτονικής από επιπλέον hardware, η ανάμιξή τους οδηγεί σε εξαιρετικά μεγάλο κόστος υλικού.

Τα πλεονεκτήματά τους εντοπίζονται στην επαρκή υλοποίηση συγκεκριμένων τμημάτων τους με υψηλού επιπέδου τεχνικές, όπως pass-transistor λογική ή δυναμική λογική (π.χ. Manchester-chain αθροιστές), που δεν είναι συμβατές με τεχνολογίες κελιών. Γενικά όλες οι τεχνικές επιτάχυνσης δείχνουν τη δυναμική τους όταν εφαρμόζονται σε μεγάλα blocks και όχι όταν αναμειγνύονται.

Η πιο συχνά χρησιμοποιήσιμη υβριδική αρχιτεκτονική χρησιμοποιεί CLA blocks με ένα τελικό στάδιο CSLA. Μετρώντας με το unit-gate μοντέλο, η ταχύτητα είναι ακριβώς η ίδια με έναν καθαρό CLA. Ο αριθμός των πυλών όμως, αυξάνεται δραστικά εξαιτίας του σταδίου πολυπλέκτη, που είναι εξαιρετικά ακριβό στις τεχνολογίες κελιών.

Συγκρίσεις βασισμένες στα unit-gate μοντέλα εμβαδού και καθυστέρησης

Σε ασυμπτωτική πολυπλοκότητα χρόνου και εμβαδού, οι αρχιτεκτονικές δυαδικών αθροιστών μπορούν να χωριστούν σε τέσσερις κλάσεις. Από αυτές έχουμε αναλύσει τις περισσότερες των περιπτώσεων. Έτσι αν n το μήκος της λέξης τελεστέου και l ο αριθμός των επιπέδων (σημειώνεται πως παρακάτω αναφέρονται και αρχιτεκτονικές αθροιστών όπως ο Carry increment με περισσότερα του ενός επίπεδα, για τους οποίους δε μιλήσαμε μιας και δεν εντάσσονται στους στόχους της εργασίας):

Αριθμός πυλών

adder type	word length [bits] 8	16	32	64	128
RCA	**58**	**114**	**226**	**450**	**898**
CSKA-1L	76	146	286	554	1090
CSKA-2L	71	158	323	633	1248
CSLA-1L	87	194	403	836	1707
CIA-1L	78	157	314	631	1266
CIA-2L	79	158	316	635	1273
CIA-3L	80	159	324	639	1280
CLA	92	204	428	876	1772
PPA-SK	73	165	373	837	1861
PPA-BK	70	147	304	621	1258
PPA-KS	88	216	520	1224	2824
COSA	115	289	687	1581	3563

Καθυστέρηση πυλών

adder type	word length [bits] 8	16	32	64	128
RCA	16	32	64	128	256
CSKA-1L	12	16	24	32	48
CSKA-2L	12	16	20	24	32
CSLA-1L	10	12	18	24	34
CIA-1L	10	12	18	24	34
CIA-2L	10	12	16	18	22
CIA-3L	10	12	16	18	20
CLA	12	16	20	24	28
PPA-SK	10	12	14	16	18
PPA-BK	12	16	20	24	28
PPA-KS	10	12	14	16	18
COSA	**8**	**10**	**12**	**14**	**16**

Κανονικοποιημένο γινόμενο αριθμού, καθυστέρησης

adder type	word length [bits] 8	16	32	64	128
RCA	1.17	1.92	2.86	5.04	8.21
CSKA-1L	1.15	1.23	1.36	1.55	1.87
CSKA-2L	1.08	1.33	1.28	1.33	1.43
CSLA-1L	1.10	1.23	1.43	1.76	2.07
CIA-1L	0.99	**0.99**	1.12	1.32	1.54
CIA-2L	1.00	1.00	**1.00**	**1.00**	1.00
CIA-3L	1.01	1.01	1.03	1.01	**0.91**
CLA	1.40	1.72	1.69	1.84	1.77
PPA-SK	**0.92**	1.04	1.03	1.17	1.20
PPA-BK	1.06	1.24	1.20	1.30	1.26
PPA-KS	1.11	1.37	1.44	1.71	1.82
COSA	1.16	1.52	1.63	1.94	2.04

Ασυμπτωτικά:

area	delay	AT-product	adder schemes
$O(n)$	$O(n)$	$O(n^2)$	ripple-carry
$O(n)$	$O(n^{1/2})$	$O(n^{3/2})$	carry-skip, carry-select, carry-increment
$O(n)$	$O(\log n)$	$O(n \log n)$	carry-lookahead, parallel-prefix
$O(n \log n)$	$O(\log n)$	$O(n \log^2 n)$	parallel-prefix, conditional-sum

adder type	gate count	gate delays	gc × gd-product	architecture description
RCA	$7n$	$2n$	$14n^2$	ripple-carry
CSKA-1L	$8n$	$4n^{1/2}$	$32n^{3/2}$	1-level carry-skip
CSKA-1L*	$10n$	$4n^{1/2}$	$40n^{3/2}$	irredundant 1-level carry-skip
CSKA-2L	$8n$	$x\,n^{1/3}$ *	$x\,n^{4/3}$ *	2-level carry-skip
CSLA-1L	$14n$	$2.8n^{1/2}$	$39n^{3/2}$	1-level carry-select
CIA-1L	$10n$	$2.8n^{1/2}$	$28n^{3/2}$	1-level carry-increment
CIA-2L	$10n$	$3.6n^{1/3}$	$36n^{4/3}$	2-level carry-increment
CIA-3L	$10n$	$4.4n^{1/4}$	$44n^{5/4}$	3-level carry-increment
CLA	$14n$	$4 \log n$	$56n \log n$	"standard" carry-lookahead
PPA-BK	$10n$	$4 \log n$	$40n \log n$	parallel-prefix (Brent-Kung)
PPA-SK	$3/2\,n \log n$	$2 \log n$	$3n \log^2 n$	parallel-prefix (Sklansky)
PPA-KS	$3n \log n$	$2 \log n$	$6n \log^2 n$	parallel-prefix (Kogge-Stone)
COSA	$3n \log n$	$2 \log n$	$6n \log^2 n$	conditional-sum (Sklansky)

* The exact factors for CSKA-2L have not been computed due to the highly irregular optimal block sizes.

Video

«Πήρε βίντεο», λέγαμε κάποτε...

Με τη συνεχή πτώση της τιμής του μεταδιδόμενου ή αποθηκευόμενου bit, δε γίνεται αμέσως προφανής η ανάγκη για συμπίεση video.

Αλλά η συμπίεση:

- Δίνει τη δυνατότητα να χρησιμοποιήσουμε ψηφιακό βίντεο στα περιβάλλοντα μετάδοσης και αποθήκευσης τα οποία δεν υποστηρίζουν ασυμπίεστο "raw" βίντεο. Για παράδειγμα παρά τις broadband συνδέσεις, δεν έχουμε ακόμη τη δυνατότητα να παίζουμε real time raw βίντεο. Αν σε ένα DVD προσπαθούσαμε να αποθηκεύσουμε raw βίντεο, διάρκειά του θα ήταν λίγα δευτερόλεπτα.
- Ακόμη και αν είχαμε διαθέσιμες παντού υψηλής μετάδοσης γραμμές, θα ήταν προτιμότερο να περάσουμε ένα υψηλής ευκρίνειας συμπιεσμένο βίντεο ή πολλαπλά κανάλια από ένα μοναδικό, χαμηλής ανάλυσης ασυμπίεστο ρεύμα.

Ένα σήμα που μεταφέρει πληροφορία μπορεί να συμπιεστεί αφαιρώντας το πλεόνασμα από το σήμα. Σε ένα σύστημα συμπίεσης χωρίς απώλειες το αρχικό σήμα μπορεί να συμπιεστεί και να αποσυμπιεστεί χωρίς καμία απώλεια. Δυστυχώς όμως, οι σημερινές μέθοδοι συμπίεσης εικόνας και βίντεο δεν προσφέρουν αρκετό ποσοστό συμπίεσης. Περισσότερο πρακτικές είναι οι μέθοδοι που έχουν απώλειες με μεγαλύτερη συμπίεση στο τίμημα του τελικού σήματος διαφορετικού από το αρχικό.

Στόχος της συμπίεσης βίντεο είναι η επαρκής αφαίρεση του πλεονάσματος, χωρίς να πέφτει η ποιότητα. Τέτοιες μέθοδοι εφαρμόζονται στο χρονικό, χωρικό και/ή χώρο των συχνοτήτων.

Δεν υπάρχουν πρότυπα σήμερα...

Αντιθέτως. Το MPEG-4 Visual και το H.264 (επίσης γνωστό ως Advanced Video Coding - AVC) είναι σχετικά νέα πρότυπα (standards) για την κωδικοποιημένη αναπαράσταση της οπτικής πληροφορίας. Κάθε πρότυπο είναι ένα έγγραφο που πρωταρχικά ορίζει δύο πράγματα, μία κωδικοποιημένη αναπαράσταση (syntax) που περιγράφει τα οπτικά δεδομένα σε συμπιεσμένη μορφή και μία μέθοδο αποκωδικοποίησης της αναπαράστασης αυτής για την ανακατασκευή της πληροφορίας.

Στην πράξη τα πρότυπα δεν ορίζουν τον κωδικοποιητή, παρά μόνο την έξοδο που πρέπει αυτός να έχει. Επίσης, μία μέθοδος αποκωδικοποίησης ορίζεται σε κάθε πρότυπο. Όμως, οι κατασκευαστές είναι ελεύθεροι να δημιουργήσουν εναλλακτικούς αποκωδικοποιητές, εφόσον ικανοποιούν τα αποτελέσματα που ορίζει το πρότυπο.

Το πρότυπο με το οποίο επιθυμούμε να συμμορφωθούμε εμείς είναι το H.264. Η προσπάθεια για τη δημιουργία του, ξεκίνησε από τη Video Coding Experts Group (VCEG), μία ομάδα εργασίας του International Telecommunication Union (ITU-T), που λειτουργεί παρόμοια με την MPEG και έχει εργαστεί και σε άλλα οπτικά τηλεπικοινωνιακά πρότυπα. Το τελικό στάδιο ανάπτυξης του H.264 διεξήχθη από την Joint Video Team, μία συνεργασία των VCEG και MPEG που κατέληξε στη δημοσίευση του προτύπου ως "MPEG-4 Part 10" στο ISO/IEC και H.264 στο ITU-T το 2003.

Σε γενικές γραμμές, αντίθετα με την εξαιρετικά ευέλικτη προσέγγιση του MPEG-4, το H.264 στοχεύει ειδικά στην επαρκή συμπίεση των video frames. Συγκεκριμένα, επαρκής συμπίεση, μετάδοση επιτεύχθηκαν με διάφορες νέες τεχνικές. Επίσης το H.264 στοχεύει στο να μπορεί να χρησιμοποιηθεί σε δημοφιλείς εφαρμογές συμπίεσης βίντεο.

Μέσα στα κουτιά...

Κωδικοποίηση βίντεο είναι η διαδικασία συμπίεσης και αποσυμπίεσης ενός σήματος ψηφιακού βίντεο.

Μία φυσική οπτική σκηνή είναι χωρικά και χρονικά συνεχής. Για να την αναπαραστήσουμε σε ψηφιακή μορφή πρέπει να εκτελέσουμε δειγματοληψία της πραγματικής σκηνής χωρικά και χρονικά.

Το ***ψηφιακό βίντεο*** είναι η αναπαράσταση της δειγματοληφθήσας σκηνής βίντεο σε ψηφιακή μορφή. Κάθε χώρο-χρονικό δείγμα (στοιχείο εικόνας=picture element ή pixel) αναπαρίσταται ως ένας αριθμός ή ένα σύνολο αριθμών που περιγράφουν τη φωτεινότητα και το χρώμα του δείγματος.

Κατόπιν ακολουθεί μία διαδικασία - που δεν είναι σκοπός της εργασίας η επεξήγησή της - που παράγει χρησιμοποιώντας έναν πίνακα από CCDs (Charged Coupled Devices) για να δημιουργήσει ένα πίνακα από ηλεκτρικά σήματα που αναπαριστούν την εικόνα. Αυτά είναι που δειγματοληπτούνται και παράγουν την εικόνα ή πλαίσιο (frame) που έχει καθορισμένες τιμές πλέον στα σημεία που δειγματοληπτήσαμε. Η διάταξη των σημείων αυτών είναι συνήθως ένα ορθογώνιο πλαίσιο (grid). Αυξάνοντας την ανάλυση λοιπόν, αυξάνουμε τον αριθμό των δειγματοληψιών, άρα και των pixels και μειώνουμε τις μεταξύ τους αποστάσεις αν αναφερθούμε στην κλίμακα της αρχικής εικόνας.

Μία κινούμενη εικόνα βίντεο απαθανατίζεται παίρνοντας ένα ορθογώνιο «στιγμιότυπο» του σήματος σε περιοδικά χρονικά διαστήματα. Ρυθμοί μικρότεροι των 10 frames (πλαισίων ή εικόνων) το δευτερόλεπτο χρησιμοποιούνται μερικές φορές στις χαμηλού ρυθμού μετάδοσης επικοινωνίες. Η κίνηση φαίνεται σπασμωδική και αφύσικη. Ανάμεσα στα 10 με 25 frames το δευτερόλεπτο είναι τυπική δειγματοληψία για επικοινωνίες χαμηλού ρυθμού μετάδοσης. Πάλι, στις γρήγορες εναλλαγές

της εικόνας έχουμε σπασμωδική κίνηση. Οι τηλεοπτικές εικόνες έχουν προτυποποιημένο ρυθμό δειγματοληψίας 25 ή 30 frames το δευτερόλεπτο αναλόγως αν το βίντεο είναι PAL(E.U.) ή NTSC(U.S.) αντίστοιχα. Μεγαλύτεροι ρυθμοί της τάξης των 50-60 frames απαιτούν μεγαλύτερο ρυθμό μετάδοσης αλλά παράγουν και καλύτερη εικόνα. Στα PAL και NTSC όμως έχουμε interlacing(διαπλοκή)...

Διαπλοκή...

Ένα σήμα βίντεο μπορεί να δειγματοληφθεί ως μία σειρά ολόκληρων πλαισίων-frames ή ως μία ακολουθία διαπλεκόμενων πεδίων(fields). Σε μία τέτοια ακολουθία λαμβάνουμε διπλάσιο αριθμό δειγμάτων στον ίδιο χρόνο, αλλά σε κάθε δειγματοληψία εναλλάσσουμε : λαμβάνουμε είτε στις περιττές είτε στις άρτιες γραμμές. Αυτό έχει ως αποτέλεσμα με το ίδιο ποσό δεδομένων να βελτιώσουμε αρκετά την πληροφορία μας.

Δεν πιστεύω στα μάτια μου!

Τα χρώματα που αντιλαμβάνεται το μάτι μπορούν όλα να συντεθούν με βεβαρημένους συνδυασμούς των κόκκινο, μπλε και πράσινο. Εξ'ου και ο R-G-B χρωματικός χώρος που χρησιμοποιούσαμε στις παλαιότερες συσκευές απεικόνισης και αποθήκευσης βίντεο. Τα χρώματα αυτά παριστάνονται από τις CRTs(Color Cathode Ray Tubes) και τις LCDs (Liquid Crystal Displays) με ξεχωριστή απεικόνιση του κάθε χρώματος αναλόγως την ένταση του κάθε τμήματος.

Το ανθρώπινο οπτικό σύστημα όμως είναι λιγότερο ευαίσθητο στο χρώμα από ότι στη φωτεινότητα. Έτσι με έναν μετασχηματισμό του **R-G-B χώρου** έχουμε το **Y-C_b-C_r χώρο**. Y είναι η φωτεινότητα:

$$Y = k_r*R + k_g*G + k_b*B \ ,$$

όπου τα k είναι βάρη.

Οι χρωματικές διαφορές μπορούν να αναπαρασταθούν, εξαιτίας του ότι άθροισμα C_b + Cr + Cg είναι σταθερό, με δύο εξισώσεις οι οποίες δεν είναι επί του παρόντος.

Κάθε εικόνα βίντεο, που μπορεί να είναι είτε ***πλαίσιο(frame)***, είτε ***πεδίο(field)***, χωρίζεται σε σταθερού μεγέθους ***macroblocks***(η ελληνική μετάφραση είναι εξαιρετικά άκυρη) που καλύπτουν μία παραλληλόγραμμη περιοχή εικόνας 16X16 δειγμάτων της φωτεινότητας και 8X8 δείγματα κάθε χρωματικού στοιχείου (για την περίπτωση της μορφοποίησης Y:U:V = 4:2:0). Όλα τα δείγματα του macroblock προβλέπονται χωρικά ή χρονικά και τα ***residuals(υπόλοιπα της αφαίρεσης)*** πρόβλεψης μεταδίδονται χρησιμοποιώντας κωδικοποίηση μετασχηματισμών. Εξαιτίας αυτού, κάθε στοιχείο(Y,U,V) των residuals πρόβλεψης χωρίζεται σε block διαφόρων διαστάσεων. Τα macroblocks ομαδοποιούνται σε ***slices(φέτες)***, που γενικά παριστάνουν υποσύνολα μίας δεδομένης εικόνας που μπορούν να αποκωδικοποιηθούν ανεξάρτητα.

Φέτες και προβλέψεις για το μέλλον... (Slices and Predictions)

Το H.264 υποστηρίζει 5 διαφορετικούς τύπους από προς-κωδικοποίηση slices. Η απλούστερη είναι η I, όπου I σημαίνει intra(ενδο). Στις I slices, τα macroblocks κωδικοποιούνται χωρίς αναφορά σε άλλες εικόνες εντός της ακολουθίας βίντεο. Αυτό λέγεται Intra Prediction (Ενδο-Πρόβλεψη). Από την άλλη πλευρά, μπορούν να χρησιμοποιηθούν προ-κωδικοποιημένες εικόνες. Έτσι σχηματίζεται ένα σήμα πρόβλεψης για macroblocks των κωδικοποιημένων με πρόβλεψη P και B slices(φετών)(P σημαίνει predictive ενώ B bi-predictive). Οι δύο αυτού τύποι αποτελούν την Inter Prediction (Δια-Πρόβλεψη). Οι εναπομείναντες τύποι slices

είναι οι SP (switching P) και SI (switching I), που ορίζονται για εναλλαγές ανάμεσα σε bit streams που κωδικοποιούνται σε διαφορετικούς ρυθμούς μετάδοσης.

Intra Prediction

Στο H.264, η Intra-Prediction με μεγέθη block 4x4, 8x8 και 16X16 χρησιμοποιείται για να συμπιέσει τα I-Macroblocks. Η Intra κωδικοποίηση αναφέρεται στην περίπτωση που εκμεταλλευόμαστε μόνο χωρικά πλεονάσματα (συσχετίσεις) σε μία εικόνα βίντεο. Το frame που έχουμε ως αποτέλεσμα της διαδικασίας αυτή αναφέρεται ως I-picture(I-εικόνα). Οι I-εικόνες τυπικά κωδικοποιούνται με άμεση εφαρμογή των μετασχηματισμών στα διάφορα macroblocks του frame. Ως συνέπεια, οι κωδικοποιημένες I-εικόνες είναι μεγάλες σε μέγεθος αφού δε χρησιμοποιούμε χρονική πληροφορία ως μέρος της διαδικασίας κωδικοποίησης. Για να αυξήσουμε την επάρκεια της διαδικασίας κωδικοποίησης intra στο H.264,εκμεταλλευόμαστε τη χωρική συσχέτιση ανάμεσα σε γειτονικά macroblocks. Η ιδέα βασίζεται στην παρατήρηση πως τα γειτονικά macroblocks τείνουν να έχουν παρόμοιες ιδιότητες. Συνεπώς, ως πρώτο βήμα στη διαδικασία κωδικοποίησης ενός δεδομένου macroblock, μπορούμε να προβλέψουμε το υπό μελέτη macroblock συναρτήσει των γύρω του. Τυπικά, χρησιμοποιούμε το πάνω και το αριστερά, αφού αυτά θα έχουν κωδικοποιηθεί νωρίτερα. Υπάρχουν συνολικά 9 επιλογές πρόβλεψης για κάθε 4X4 block φωτεινότητας, 9 για κάθε 8X8 και 4 για κάθε 16X16 block φωτεινότητας και 4 για κάθε χρωματική συνιστώσα.

Inter Prediction

Για όλους τους άλλους τύπους macroblock, το Η.264 χρησιμοποιεί δια-πρόβλεψη για να τα συμπιέσει. Χρησιμοποιείται το χρονικό μοντέλο. Σε αυτή την περίπτωση το προβλεπόμενο frame δημιουργείται από ένα ή περισσότερα παρελθόντα ή μελλοντικά frames τα οποία ονομάζονται ***frames αναφοράς***. Η διαδικασία της εύρεσης της βέλτιστης πρόβλεψης ονομάζεται ***Εκτίμηση Κίνησης (Motion Estimation)***. Η ακρίβεια της πρόβλεψης μπορεί συχνά να βελτιωθεί με τον συμψηφισμό της κίνησης ανάμεσα στα frames αναφοράς και στο παρόν. Η διαδικασία αυτή ονομάζεται ***Motion Compensation (Συμψηφισμός Κίνησης)***. Ένας πρακτικός και ευρέως διαδεδομένος τρόπος motion compensation είναι να συμψηφίσουμε για κίνηση παραλληλόγραμμων τομέων ή block στο παρόν frame. Το Η.264 είναι ένα ***block-based motion-compensated hybrid transform codec***. Υποστηρίζει διάφορα μεγέθη block : 16x16, 8x16, 16x8, 8x8, 4x8, 4x4 pixels. Συνήθως το κριτήριο εύρεσης του block που ταιριάζει καλύτερα είναι η ενέργεια των υπολοίπων που σχηματίζονται από την αφαίρεση του υποψήφιου block από το παρόν MxN block. Η υποψήφια περιοχή που ελαχιστοποιεί την ενέργεια των υπολοίπων επιλέγεται ως η βέλτιστη επιλογή. Για να μειωθεί όμως η υπολογιστική πολυπλοκότητα, οι περισσότερες εφαρμογές, ανάμεσα στις οποίες το Η.264, χρησιμοποιούν το ΑΑΔ.

Ε, και;

Υπολογιστικά, ένα στοιχείο που εμφανίζεται και στις δύο μεθόδους πρόβλεψης, Intra και Inter είναι το Άθροισμα των Απολύτων Διαφορών(Α.Α.Δ.)(SAD-Sum of Absolute Differences ή αλλιώς SAE Sum of Absolute Errors).

ΑΑΔ (SAD)

Ο κωδικοποιητής τυπικά επιλέγει τη μέθοδο πρόβλεψης και την Intra και στην Inter Prediction για κάθε block, που ελαχιστοποιεί τη διαφορά ανάμεσα στο block πρόβλεψης P και στο block που πρόκειται να κωδικοποιηθεί, έστω C. Η επιλογή γίνεται χρησιμοποιώντας το ΑΑΔ, που επίσης είναι γνωστό ως απόσταση L_1 ανάμεσα στη διανυσματική μορφή των δύο blocks, P και C.

Η L_1-απόσταση είναι ένα θετικό ορισμένο μέτρο που ορίζεται για διανύσματα σε k-διάστατους διανυσματικούς χώρους από την αντίστοιχη νόρμα L_1 του διανύσματος διαφορών, ως εξής:

Έστω, $\underline{x}, \underline{y} \in R^k$

Τότε: $$L_1(\underline{x}, \underline{y}) = \sum_{i=1}^{k} | x_i - y_i |$$

Είναι εύκολο να δείξουμε πως ισχύουν οι ακόλουθες ιδιότητες της L_1 νόρμας:

Συμμετρική	$L_1(x,y) = L_1(y,x)$
Θετικά ορισμένη	$L_1(x,y) \geq 0.$ $L_1(x,y) = 0 \Leftrightarrow x=y$
Τριγωνική Ανισότητα	$L_1(x,y) - L_1(y,z) \geq L_1(x,z)$

Υπάρχουν βέβαια και άλλα μέτρα που χρησιμοποιούνται στους διανυσματικούς χώρους. Ίσως η πιο δημοφιλής είναι η Ευκλείδεια απόσταση, γνωστή και ως L_2 νόρμα, για την οποία είναι εύκολο να αποδειχθούν οι ίδιες ιδιότητες με την L_1.

$$L_2(\underline{x},\underline{y}) = \left(\sum_{i=1}^{k} | x_i - y_i |^2 \right)^{1/2}$$

$$L_n(\underline{x},\underline{y}) = \left(\sum_{i=1}^{k} | x_i - y_i |^n \right)^{1/n}$$

Γενικεύοντας η n-οστή νόρμα δύο διανυσμάτων δίνεται από το διπλανό τύπο και οι ιδιότητες της νόρμας αποδεικνύονται το ίδιο εύκολα με της προηγούμενης.

Στην οριακή περίπτωση που n -> ∞, γνωστή και ως L_∞ μπορεί να αποδειχθεί πως :

$$L_\infty(\underline{x},\underline{y}) = \lim_{n\to\infty} L_n(\underline{x},\underline{y}) = \lim_{n\to\infty} \left(\sum_{i=1}^{k} | x_i - y_i |^n \right)^{1/n} \Rightarrow$$

$$L_\infty(\underline{x},\underline{y}) = \max_{i\in\{1,\dots,k\}} | x_i - y_i |$$

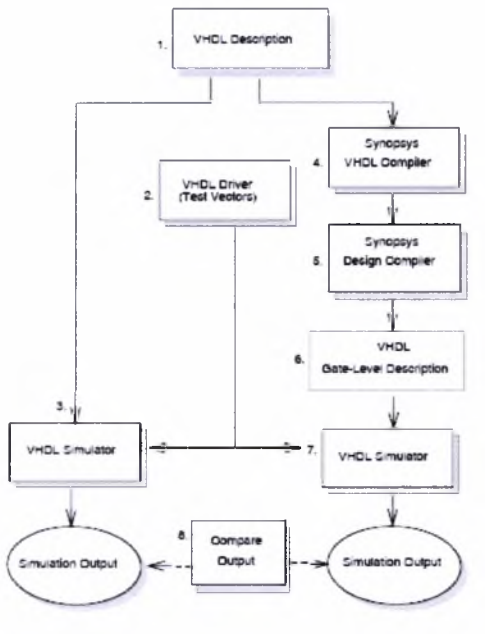
Όλες οι νόρμες προσπαθούν να αντιστοιχίσουν σε ένα, μόνο, αριθμό το ποσό της διαφοράς ανάμεσα σε δύο διανύσματα. Υπάρχει διαφωνία για το ποια νόρμα είναι καλύτερη για να εκφράσει το λάθος στην επεξεργασία σημάτων και ειδικότερα σε ήχο, εικόνα και βίντεο. Παραδοσιακά οι ερευνητές χρησιμοποιούν την L_2 για κριτήριο ελαχιστοποίησης στην επεξεργασία σημάτων και στη συμπίεση. Αυτός είναι ο κύριος λόγος που η ακμή του σηματοθορυβικού λόγου (*peak signal-to-noise ratio PSNR*, που είναι μία λογαριθμική αναπαράσταση της L_2 νόρμας) χρησιμοποιείται στην

αντίστοιχη βιβλιογραφία για να εκφράσει ποιότητα σήματος. Από την άλλη πλευρά, το ΑΑΔ - που είναι η L_1 νόρμα - έχει χρησιμοποιηθεί ευρέως ως το βασικό υπολογιστικό κομμάτι για εύρεση ταιριάσματος blocks στη συμπίεση βίντεο, αφού δεν απαιτεί την επιπρόσθετη πολυπλοκότητα του πολλαπλασιαστή των L_2 νορμών. Αυτός είναι ένας απαραίτητος συμβιβασμός ώστε να έχουμε μία πρακτική υλοποίηση ενός κωδικοποιητή βίντεο.

Ροή σχεδίασης, μεθοδολογία και βασικές έννοιες Computer Aided Design στην εργασία

(Σ.Σ. : Στο κεφάλαιο που ακολουθεί αναφέρονται εργαλεία που χρησιμοποιήθηκαν για να ολοκληρωθεί η εργασία αυτή. Σε καμία περίπτωση δεν αποσκοπούμε στη διαφήμιση και προώθησή τους.)

Για να καταφέρουμε τα επιθυμητά αποτελέσματα είναι απαραίτητος ο σχεδιασμός της κατάλληλης μεθοδολογίας. Οι απαιτήσεις της υλοποίησης του ΑΑΔ, χρησιμοποιώντας τα στοιχεία που έχουμε περιγράψει στα προηγούμενα κεφάλαια, συγκεκριμενοποιούνται στις σχέσεις που περιγράψαμε στην εισαγωγή: Εμβαδό και Χρονισμός. Τα βήματα είναι:



1. Γράψιμο της περιγραφής σχεδίασης σε VHDL. Η περιγραφή αυτή μπορεί να είναι συνδυασμός δομημένων και λειτουργικών στοιχείων. Αυτή η περιγραφή χρησιμοποιείται όχι μόνο από το μεταφραστή του εργαλείου που επιλέξαμε (Synopsys σειρά εργαλείων για μετάφραση και σύνθεση, βλ. παρακάτω), αλλά

και από τον προσομοιωτή μας, που είναι το Modelsim της εταιρίας Mentor Graphics.

2. Για να επαληθεύσουμε τον κώδικα που έχουμε γράψει, χρησιμοποιούμε ακολουθίες δοκιμαστικών εισόδων σε μορφή διανυσμάτων.
3. Κατόπιν επαληθεύουμε την ακρίβεια της περιγραφής χρησιμοποιώντας τις δοκιμαστικές εισόδους που γράψαμε στο προηγούμενο βήμα και το εργαλείο προσομοίωσης. Στην προκειμένη περίπτωση το τελευταίο είναι το Modelsim SE 6.0a.
4. Ο VHDL Compiler (μεταγλωττιστής) συνθέτει την περιγραφή VHDL, εκτελεί αρχιτεκτονικές βελτιστοποιήσεις και τέλος δημιουργεί μία εσωτερική περιγραφή της σχεδίασης. Το εργαλείο που χρησιμοποιήσαμε ήταν ο VHDL Compiler της εταιρίας Synopsys.
5. Η περιγραφή η οποία ορίστηκε στην αρχή ήταν σε επίπεδο RTL. Τη συνθέτουμε χρησιμοποιώντας συγκεκριμένη τεχνολογική βιβλιοθήκη(μέχρι το προηγούμενο βήμα η συνδεσμολογία που είχαμε ήταν ανεξάρτητη τεχνολογίας) και παράγεται GaLe συνδεσμολογία βελτιστοποιημένη. Πάντα σύμφωνα με τους περιορισμούς που έχουμε θέσει και εφόσον έχουμε μεταγλωττίσει προηγουμένως τον κώδικα. Το βήμα αυτό εκτελείται με τον Design Compiler της Synopsys
6. Εξάγουμε τα αποτελέσματά μας χρησιμοποιώντας πάλι τον DeCo παράγοντας μία συνδεσμολογία με τα ίδια ονόματα εισόδων, εξόδων και modules. Υπενθυμίζουμε πως η συνδεσμολογία αυτή είναι GaLe.
7. Εκτελούμε πάλι προσομοίωση.
8. Συγκρίνουμε αρχική και τελική συνδεσμολογία ως προς την απόδοση. Η απόδοση εξαρτάται από τους περιορισμούς που έχουμε ορίσει σε προηγούμενο βήμα.

Γράφοντας τη σχεδίαση σε γλώσσα περιγραφής υλικού(HDL)

Χρησιμοποιήθηκαν ορισμένες αρχές για κώδικα ειδικό για σύνθεση και τεχνολογία της εταιρίας Synopsys που επιτρέπει να δημιουργούμε μακροκελιά αυτόματα. Η τεχνολογία αυτή ονομάζεται Designware IP και ανήκει στην εταιρία Synopsys.

Μία Γλώσσα Περιγραφής Υλικού(hardware description language), σε συντομογραφία ΓΠΥ(HDL) είναι μία οποιαδήποτε γλώσσα από μία κλάση υπολογιστικών γλωσσών που χρησιμοποιείται για επίσημη περιγραφή ηλεκτρονικών κυκλωμάτων. Μπορεί να περιγράψει τη λειτουργία ενός κυκλώματος, τη σχεδίασή του και δοκιμές για την επαλήθευσή του με την έννοια της προσομοίωσης.

VHDL για σύνθεση

Η επιτυχής σύνθεση εξαρτάται σε μεγάλο βαθμό από την κατάλληλη κατάτμηση της σχεδίασης χρησιμοποιώντας «καλό» κώδικα.

Η λογική κατάτμηση είναι το κλειδί για την επιτυχημένη σύνθεση και στο Place and Route (Τοποθέτηση και Δρομολόγηση) εφόσον η διάταξη είναι ιεραρχική. Παραδοσιακά, οι σχεδιαστές έδιναν σημασία στη λειτουργικότητα του κάθε block, χωρίς να συμπεριλαμβάνουν στους υπολογισμούς τους τη διαδικασία της σύνθεσης. Αυτό έχει ως αποτέλεσμα τα όρια μεταξύ των block να μην είναι ευέλικτα και συνεπώς να δυσκολεύεται η βελτιστοποίηση. Το πρόβλημα αυτό λύνεται με τη σωστή κατάτμηση της σχεδίασης. Επίσης, με τη σωστή κατάτμηση επιτυγχάνεται γρήγορη μεταγλώττιση και απλή σεναριακή (script).

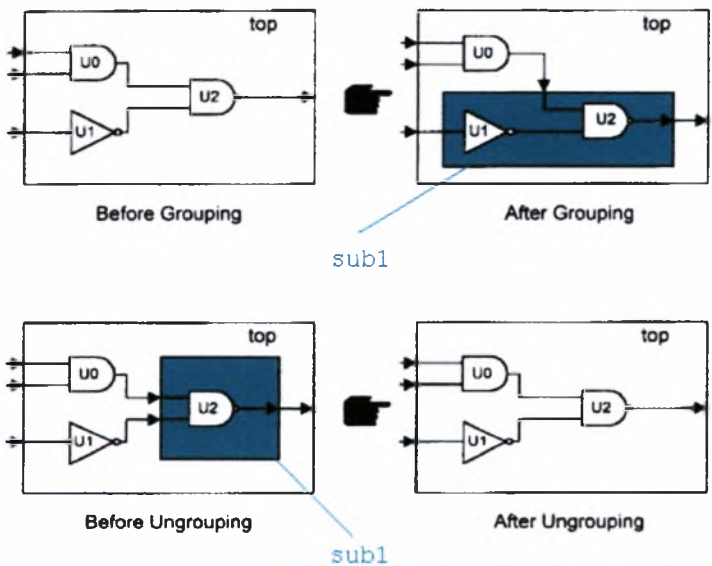
Κατάτμηση για σύνθεση

Η κατάτμηση μπορεί να θεωρηθεί ως μία χρήση της τεχνικής αλγορίθμων «Διαίρει και Βασίλευε». Χρησιμοποιείται για να μειώσει την πολυπλοκότητα των σχεδιάσεων και να τις μετατρέψει σε απλούστερα και περισσότερο διαχειρίσιμα blocks. Ένα από τα σπουδαιότερα πλεονεκτήματα που προσφέρει η κατάτμηση είναι η επαναχρησιμοποίηση των σχεδιάσεων.

Οι ακόλουθες αρχές επιτυγχάνουν καλύτερα αποτελέσματα σύνθεσης και μειώνουν το χρόνο μεταγλώττισης.

- Σχετική συνδυαστική λογική πρέπει να είναι συγκεντρωμένη στο ίδιο module.
- Κατατέμνουμε για επαναχρησιμοποίηση σχεδίασης
- Ξεχωρίζουμε τα modules σύμφωνα με τη λειτουργικότητά τους.
- Ξεχωρίζουμε τη δομημένη λογική από την τυχαία.
- Θέτουμε έναν περιορισμό στο μέγεθος του κάθε block(για παράδειγμα 10 χιλιάδες πύλες ανά block)
- Δεν προσθέτουμε glue-logic στο κορυφαίο επίπεδο
- Απομονώνουμε τις μηχανές καταστάσεων από την υπόλοιπη λογική.
- Αποφεύγουμε τη χρήση πολλαπλών ρολογιών εντός του ίδιο block.
- Απομονώνουμε το block που χρησιμοποιείται για το συγχρονισμό ρολογιών.
- Όταν κατατέμνουμε, σκεφτόμαστε τη μορφή της διάταξης του κυκλώματος.

Για να εκτελέσουμε τα παραπάνω, δηλαδή να ομαδοποιούμε και να κατατέμνουμε σχεδιάσεις χρησιμοποιούμε τον Design Compiler (DeCo). Πρόκειται για ένα πολύ-εργαλείο της Synopsys, το οποίο έχει ως κύρια λειτουργία τη σύνθεση και βελτιστοποίηση σχεδιάσεων. Θα μιλήσουμε αναλυτικά στο υποκεφάλαιο της σύνθεσης. Προς το παρόν θα αναφέρουμε τις εντολές του DeCo με τις οποίες εκτελούμε τις προαναφερθείσες λειτουργίες. Αυτές είναι η group και η ungroup:



Επίσης έχουμε τη δυνατότητα να «ισοπεδώσουμε» την ιεραρχία με *ungroup -flatten -all*.

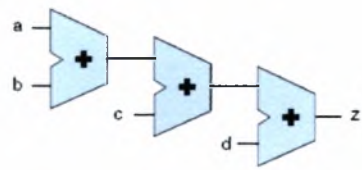
Designware Building Block IP (Foundation Library) (DBBIP)

Είναι μία συλλογή επαναχρησιμοποιήσιμων blocks πνευματικής ιδιοκτησίας της εταιρίας Synopsys που είναι ενσωματωμένα εντός του περιβάλλοντος του Design Compiler(DeCo). Η χρήση **DesignWare Building Block IP(DBBIP)** επιτρέπει διαφανή, υψηλού επιπέδου βελτιστοποίηση επιδόσεων κατά τη διαδικασία της σύνθεσης. Η βιβλιοθήκη αυτή περιέχει υψηλών επιδόσεων υλοποιήσεις της Basic Library IP και επιπλέον πολλά IP που υλοποιούν περισσότερο προχωρημένες αριθμητικές, ακολουθιακές και λογικές συναρτήσεις. Η DesignWare Building Block IP αποτελείται από:

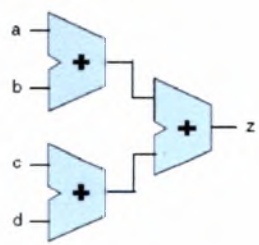
- Basic Library: Ένα σύνολο IP δεσμευμένων με τον HDL Compiler που υλοποιούν διάφορες κοινές αριθμητικές και λογικές συναρτήσεις
- Logic: Συνδυαστικά και Ακολουθιακά IP
- Math: Αριθμητικά και Τριγωνομετρικά IP.
- Digital Signal Processing (DSP) IP: FIR και IIR φίλτρα.
- Memory: Καταχωρητές, FIFOs, ελεγκτές FIFO, Σύγχρονες και Ασύγχρονες RAM και Stack IP.
- Application Specific: Data Integrity, Interface, JTAG IP, και άλλα.

Για παράδειγμα, όταν χρησιμοποιούμε την πράξη της πρόσθεσης στην HDL χρησιμοποιώντας τον αντίστοιχο τελεστή "+" σε μία περιγραφή σχεδίασης, ο HDL Compiler (Verilog ή VHDL) συμπεραίνει(infers) την ανάγκη ενός πόρου αθροιστή. Τοποθετεί, λοιπόν, μία αφηρημένη περιγραφή της πράξης της πρόσθεσης στη συνδεσμολογία του κυκλώματος. Αυτή η αναπαράσταση - που ονομάζεται *Συνθετικός τελεστής* (Synthetic operator) - χειρίζεται από τους υψηλού επιπέδου αλγορίθμους του HDL Compiler και

εφαρμόζεται στη σχεδίαση. Αυτές οι βελτιστοποιήσεις συμπεριλαμβάνουν αριθμητική βελτιστοποίηση, μερισμό πόρων και αντιμετάθεση ακίδων(pin permutation).

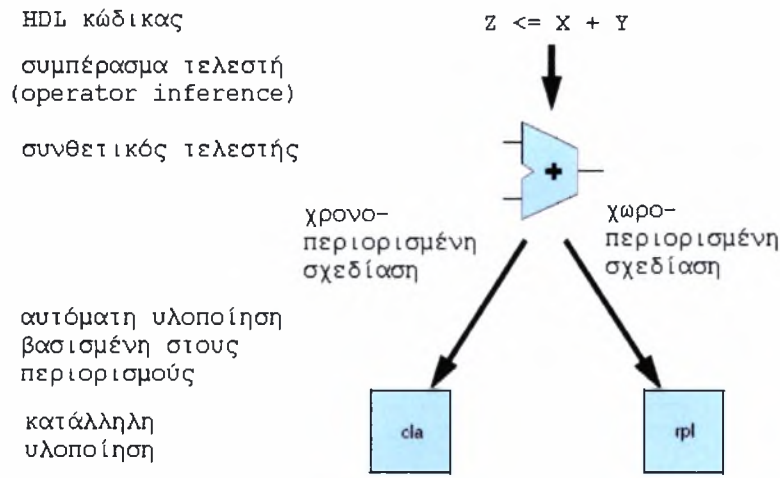


Η αριθμητική βελτιστοποίηση χρησιμοποιεί τους κανόνες της άλγεβρας για να βελτιώσει το εμβαδό του κυκλώματος και την απόδοσή του, αναδιατάσσοντας τις πράξεις. Για παράδειγμα, η έκφραση a + b + c + d περιγράφει τρία επίπεδα από συνεχόμενες αθροίσεις. Η αναδιάταξη της έκφρασης σε (a + b) + (c + d), που μπορεί να οδηγήσει σε γρηγορότερη λογική (έχοντας μόνο δύο επίπεδα από συνεχόμενες αθροίσεις).



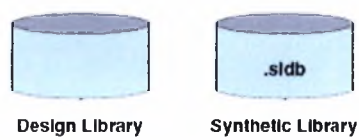
Ο μερισμός πόρων επιτρέπει σε όμοιες πράξεις που δεν επικαλύπτονται στο χρόνο να εκτελεστούν από το ίδιο υλικό. Η αναδιάταξη ακίδων εκμεταλλεύεται το γεγονός πως μερικές πράξεις όπως ο πολλαπλασιασμός και η πρόσθεση δεν επηρεάζονται από την εναλλαγή των πυλών εισόδου.

Με το DBBIP, μία δεδομένη πράξη μπορεί να υλοποιηθεί με διάφορους τρόπους, με την επιλογή της υλοποίησης ενός συγκεκριμένου κυκλώματος να αφήνεται στα εργαλεία σύνθεσης. Για παράδειγμα, η άθροιση μη προσημασμένων αριθμών μπορεί να υλοποιηθεί είτε με RCA αρχιτεκτονική είτε με CLA. Μπορούμε να αφήσουμε το εργαλείο να βασιστεί στους περιορισμούς που έχουμε θέσει και να επιλέξει την κατάλληλη υλοποίηση.



Δομή Βιβλιοθηκών

Το DBBIP έχει δύο μέρη, μία *Βιβλιοθήκη Σχεδιάσεων(Design Library)* και μία *Σχεδιαστική Βιβλιοθήκη(Synthetic Library).*



- Μία *Βιβλιοθήκη Σχεδιάσεων* είναι ένας φάκελος UNIX που περιέχει διάφορες περιγραφές κυκλωμάτων για τις IP αρχιτεκτονικές. Συνήθως είναι παραμετροποιημένες.

- Μία *Συνθετική Βιβλιοθήκη* είναι ένα δυαδικό αρχείο, με κατάληξη .sldb, που συνδέει τα κυκλώματα σε μία Βιβλιοθήκη Σχεδιάσεων με τα εργαλεία σύνθεσης της Synopsys.

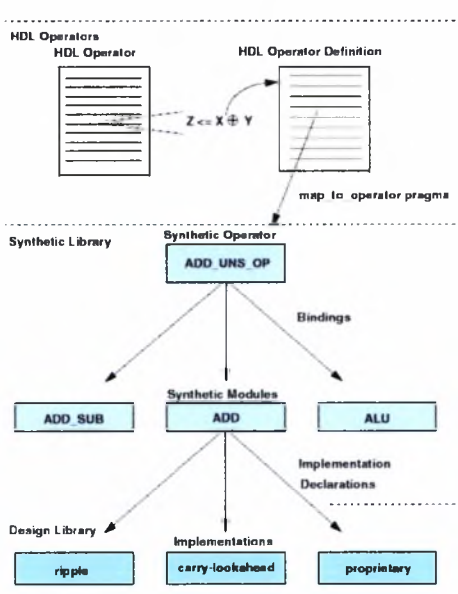
Οι περιγραφές κυκλωμάτων σε μία Βιβλιοθήκη Σχεδιάσεων αποθηκεύονται σε δυαδική μορφή, που μπορούν να χρησιμοποιηθούν απευθείας με τα εργαλεία της Synopsys. Τα κυκλώματα αυτά ποικίλουν: από μία technology-specific συνδεσμολογία που δεν πρόκειται να αλλάξει κατά

τη διαδικασία της σύνθεσης, σε μία πλήρως ιεραρχική περιγραφή μίας παραμετροποιήσιμης, βελτιστοποιήσιμης σχεδίασης.

Η Συνθετική Βιβλιοθήκη περιέχει την πληροφορία που επιτρέπει στα εργαλεία σύνθεσης να εκτελέσουν υψηλού επιπέδου βελτιστοποιήσεις, συμπεριλαμβανομένου επιλογή υλοποίησης.

Οι συνδέσεις ανάμεσα σε κώδικα, Συνθετικές Βιβλιοθήκες και στις Βιβλιοθήκες Σχεδιάσεων με τη χρήση μίας ιεραρχίας αφαιρέσεων. Οι τελεστές HDL αντιστοιχούνται σε Συνθετικούς τελεστές, οι οποίοι με τη σειρά τους συνδέονται με Συνθετικά modules. Κάθε Συνθετικό module μπορεί να έχει πολλαπλές αρχιτεκτονικές υλοποιήσεις που αποκαλούνται *implementations.*

Μία άλλη κλάση του Designware Building Block IP είναι τα subblocks, που έχουν μόνο μία υλοποίηση και μπορούν μόνο να χρησιμοποιηθούν μόνο με τη μέθοδο του instantiation (συγκεκριμενοποίηση) την οποία θα περιγράψουμε παρακάτω. Είναι συνήθως χρήσιμα για μεγάλα τμήματα, όπως ο έλεγχος λαθών και διόρθωση των IPs τα οποία δεν έχουν πολλαπλές υλοποιήσεις. Τα subblocks δε χρησιμοποιούν την παρακάτω ιεραρχία που χρησιμοποιείται στα υπόλοιπα είδη κλάσεων του DBBIP.



HDL τελεστές

*Ενας **HDL τελεστής** είναι μία κατασκευή στη VHDL ή στη Verilog που διαχειρίζεται τιμές εισόδου για να παράγει τιμές εξόδου.*

Μερικοί τελεστές είναι ενσωματωμένοι στη γλώσσα, όπως +,-,*. Τα ορισμένα από το χρήση υποπρογράμματα, όπως οι συναρτήσεις και οι διαδικασίες, μπορούν επίσης να θεωρηθούν τελεστές HDL.

Το DBBIP υλοποιεί πολλούς από τους ενσωματωμένους HDL τελεστές, συμπεριλαμβανομένου τους +, -, *, <, >, <=, >=, / και τις λειτουργίες που ορίζονται από τις if case καταστάσεις. Κάθε τελεστής έχει έναν ορισμό γραμμένο σε HDL. Κάθε ορισμός περιέχει ένα προσομοιώσιμο προσδιορισμό για τη συμπεριφορά του και προαιρετικά ένα map_to_operator pragma που συνδέει τον HDL τελεστή με έναν ισοδύναμο Συνθετικό τελεστή. Ο τελεστής "/" είναι απαραίτητος για την άδεια Designware.

Πολλοί HDL τελεστές αντιστοιχίζονται εξορισμού σε Συνθετικούς Τελεστές στην προκαθορισμένη Synopsys Συνθετική βιβλιοθήκη, standard.sldb .

Συνθετικές βιβλιοθήκες

Μία συνθετική βιβλιοθήκη περιέχει ορισμούς για συνθετικούς τελεστές, συνθετικά modules και συνδέσμους. Επίσης περιέχει δηλώσεις που αντιστοιχούν συνθετικά modules με τις υλοποιήσεις τους. Οι υλοποιήσεις βρίσκονται στην αντίστοιχη Βιβλιοθήκη Σχεδιάσεων.

- *Συνθετικός τελεστής* - Αντιπροσωπεύει την πράξη που αντιστοιχεί στον HDL τελεστή. Τα εργαλεία σύνθεσης εκτελούν την αριθμητική βελτιστοποίηση και το μερισμό πόρων διαχειριζόμενα συνθετικούς τελεστές.

- *Συνθετικό module* - Ορίζει μία κοινή διεπαφή για μία οικογένεια υλοποιήσεων. Όλες οι υλοποιήσεις ενός δεδομένου module έχουν τις ίδιες πύλες και την ίδια συμπεριφορά εισόδου-εξόδου. Δε θα πρέπει να μπερδεύουμε αυτό τον όρο με τον όρο module της Verilog.

- *Σύνδεσμοι(bindings)* - Συσχετίζουν συνθετικούς τελεστές με συνθετικά modules. Για παράδειγμα, ένας σύνδεσμος συσχετίζει το συνθετικό τελεστή της πρόσθεσης με το module πρόσθεσης. Περισσότεροι από ένα τελεστή μπορούν να συνδεθούν με ένα δεδομένο συνθετικό module, και κάθε τελεστής μπορεί να συνδεθεί σε περισσότερα του ενός module.

- *Δηλώσεις υλοποιήσεων* - Συνδέουν τα συνθετικά modules με τις υλοποιήσεις σε μία Βιβλιοθήκη Σχεδιάσεων. Οι δηλώσεις των υλοποιήσεων, λοιπόν, συνδέουν τη Συνθετική Βιβλιοθήκη με τη Βιβλιοθήκη Σχεδιάσεων.

Βιβλιοθήκη Σχεδιάσεων(Design Library)

Περιέχει όλες τις πραγματικές υλοποιήσεις των κυκλωμάτων που εκτελούν τις συναρτήσεις που καλούμε όταν συμπεριλαμβάνουμε DBBIP στη σχεδίασή μας.

Οι έννοιες του DBBIP για το *συνθετικό module* και την *υλοποίηση* αντιστοιχούν πολύ κοντά στις έννοιες της VHDL *entity* και *architecture.* Μία υλοποίηση μπορεί να είναι οτιδήποτε από μία technology-specific συνδεσμολογία ως και μία RTL σχεδίαση που μπορεί να συντεθεί.

Πώς χρησιμοποιείται το DBBIP

Καθορίζουμε ποιες IP βιβλιοθήκες είναι διαθέσιμες στα εργαλεία θέτοντας τις κατάλληλες μεταβλητές στο dc_shell-xg-t που είναι η έκδοση

κελύφους του DeCo και για την οποία θα μιλήσουμε σε επόμενο κεφάλαιο. Συμπεριλαμβάνουμε σε μία σχεδίαση είτε μέσω *operator inference* είτε μέσω *component instantiation.* Στην πρώτη μέθοδο, οι συνθετικοί τελεστές αυτόματα συμπεραίνονται από την παρουσία των συγκεκριμένων τελεστών στον κώδικα HDL. Στην δεύτερη, ο κώδικας HDL επισημαίνει πώς θα χρησιμοποιηθεί ένα Συνθετικό module.

Προσομοίωση

Προσομοίωση είναι η μίμηση ενός στοιχείου του πραγματικού κόσμου, μίας κατάστασης ή μίας διαδικασίας. Στην επιστήμη υπολογιστών είναι ένα υπολογιστικό πρόγραμμα που προσπαθεί να προσομοιώσει ένα αφαιρετικό μοντέλο ενός συγκεκριμένου συστήματος.

Στην προκείμενη εργασία το Modelsim 6.0a χρησιμοποιήθηκε για τον έλεγχο των αρχείων που περιείχαν την περιγραφή των αθροιστών και των αρχείων που αποτελούσαν τα testbenches των τοπολογιών αυτών.

Αναλυτικότερα:

1. File > Change_Directory : μεταφερόμαστε στον φάκελο όπου είναι αποθηκευμένα τα αρχεία VHDL με την περιγραφή του εκάστοτε κυκλώματος αλλά και του testbench που θα χρησιμοποιήσουμε.

2. Δημιουργούμε μια νέα βιβλιοθήκη με τις εντολές File > New > Library. Στο πλαίσιο που εμφανίζεται επιλέγουμε "a map to an existing library" και ακολούθως προσδιορίζουμε την τοποθεσία στην οποία βρίσκεται η βιβλιοθήκη με βάση την οποία θέλουμε να γίνει η προσομοίωση της λειτουργίας του κυκλώματος. Στην περίπτωσή μας η βιβλιοθήκη ήταν η

GSCLIB90 της εταιρίας Cadence και όχι κάποια default επιλογή του προγράμματος.

3. Ακολούθως, επιλέγουμε Compile και παρουσιάζονται σε παραθυρικό περιβάλλον τα αρχεία που αναφέραμε παραπάνω. Αφού ελέγξουμε την ορθότητα των προς προσομοίωση αρχείων αλλά και των προγραμμάτων προσομοίωσης - testbenches - αυτών, είμαστε έτοιμοι για την κύρια φάση αυτού του τμήματος της εργασίας, την προσομοίωση.

4. Επιλέγοντας Simulate προσομοιώνουμε την λειτουργία του κυκλώματος με την βοήθεια του testbench και μας δίνεται η δυνατότητα αναπαράστασης της «λειτουργίας» του κυκλώματός μας με την χρήση κυματομορφών.

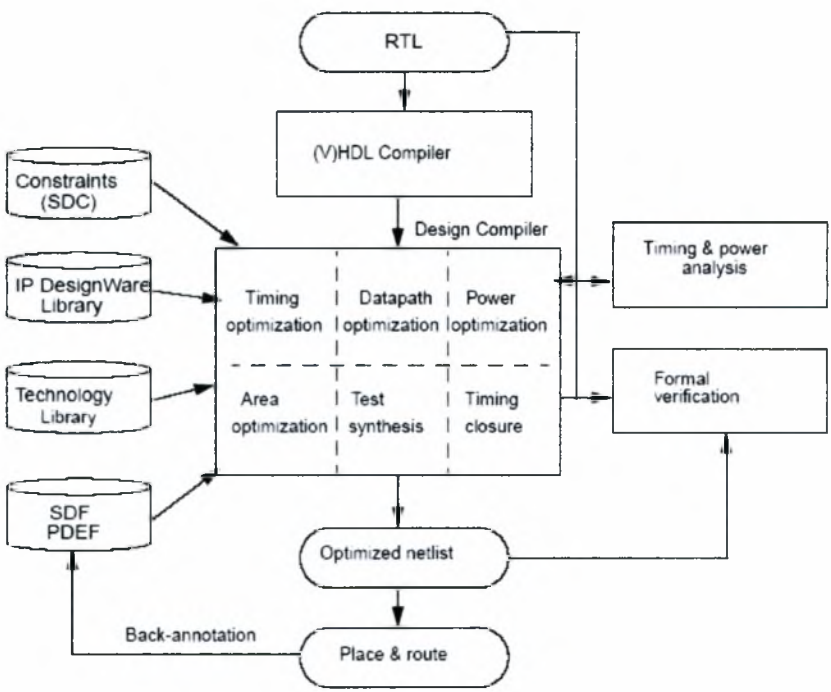
5. Αφού τσεκάρουμε για μερικές ακολουθίες εισόδων και εξόδων χειροκίνητα, δημιουργούμε με μία τυχαία συνάρτηση, σε μας με ομοιόμορφη κατανομή και χρησιμοποιώντας κάποια Χ γλώσσα προγραμματισμού (εμείς Java), διανύσματα τυχαίων ακολουθιών και κατόπιν προσομοιώνουμε όπως στο βήμα 4.

Διαδικασία σύνθεσης

Σύνθεση *ονομάζεται η διαδικασία μετατροπής RTL σχεδίασης RTL HDL κώδικα σε ένα βέλτιστο Gate Netlist πλήρως προσδιορισμένο από μια τεχνολογική βιβλιοθήκη*

Στη διαδικασία αυτή χρησιμοποιήθηκε ο *Design Compiler(DeCo)* που αποτελεί τον πυρήνα του λογισμικού της SYNOPSYS. Διαθέτει εργαλεία που συνθέτουν τον HDL κώδικα σε βελτιστοποιημένα τεχνολογικά εξαρτημένα ή και ανεξάρτητα κυκλώματα σε επίπεδο πυλών (gate level netlist). Υποστηρίζει ένα μεγάλο πλήθος από επίπεδα και ιεραρχικά μοντέλα και μπορεί να βελτιστοποιεί ταυτόχρονα τόσο συνδυαστικά όσο και ακολουθιακά κυκλώματα σε θέματα χρονισμού, μεγέθους ολοκληρωμένου και κατανάλωσης ισχύος.

Στο σχήμα της επόμενης σελίδας παρουσιάζεται ο τρόπος με τον οποίο ο DC συμβάλλει στην συνολική ροή σχεδίασης :



Τα βήματα που ακολουθούμε κατά την διαδικασία της σύνθεσης ενός κυκλώματος, σε άμεση αντιστοιχία με το παραπάνω σχήμα, είναι τα εξής :

1) Δεχόμαστε ως είσοδο, αρχεία τα οποία περιγράφουν το κύκλωμά μας σε επίπεδο καταχωρητών (RTL - Register Transfer Level), δηλαδή αρχεία σε κάποια γλώσσα περιγραφής υλικού (HDL) όπως η Verilog και η VHDL.

2) Κατά την σύνθεση ο Design Compiler μεταφράζει την HDL περιγραφή σε συνθετικά στοιχεία της DesignWare βιβλιοθήκης, όπως είναι για παράδειγμα οι αθροιστές και τα γενικά Boolean συνθετικά στοιχεία (τα οποία στο εξής θα αναφέρονται ως generic Boolean components ή πιο απλά GTECH components). Τα GTECH components δεν έχουν πληροφορίες όσον αφορά τον χρονισμό και την δυνατότητα οδήγησης και δεν προσδιορίζονται - αντιστοιχούν σε κάποια τεχνολογική βιβλιοθήκη. Ο DC χρησιμοποιεί τεχνολογικές (technology), συνθετικές (synthetic) και συμβολικές (symbolic) βιβλιοθήκες από το σημείο αυτό και μετέπειτα για την υλοποίηση της σύνθεσης και την παρουσίαση του αποτελέσματος ως γραφική αναπαράσταση.

3) Αφού γίνει η μετάφραση - μετατροπή του HDL κώδικα σε επίπεδο πυλών, ο DC βελτιστοποιεί την σχεδίαση και αντιστοιχίζει τα στοιχεία που την αποτελούν σε ένα συνδυασμό αποτελούμενο από συγκεκριμένα κελιά βιβλιοθηκών, βασιζόμενος στις επιλογές του χρήστη και τους περιορισμούς που αυτός έχει δώσει. Οι περιορισμοί αποτελούν στην ουσία τις σχεδιαστικές απαιτήσεις του χρήστη για τους στόχους που θέλει να πετύχει με βάση την απόδοση της σχεδίασης, δηλαδή αναφέρονται

στους χωρικούς και χρονικούς περιορισμούς κατά κύριο λόγο υπό τους οποίους καλείται να επιτελεστεί η σύνθεση.

4) Αφού ολοκληρωθεί η φάση αυτή, ακολουθώντας την ροή σχεδίασης που παρουσιάστηκε στο παραπάνω σχήμα, δοκιμάζουμε το αποτέλεσμα της σύνθεσης για να δούμε αν «πιάνει» τα standards που θέσαμε στην προηγούμενη φάση έτσι ώστε στην περίπτωση που παρουσιαστεί οποιοδήποτε πρόβλημα να το επιλύσουμε σε όσο το δυνατό πιο πρώιμο στάδιο του κύκλου σχεδίασης.

5) Μετά το πέρας του παραπάνω σταδίου, το κύκλωμά μας, όπως έχει πλέον διαμορφωθεί, είναι έτοιμο για την προσαγωγή του στα place & route εργαλεία ,τα οποία σε γενικές γραμμές είναι αρμόδια για την τοποθέτηση και την διασύνδεση των επιμέρους κελιών στην σχεδίαση. Ο σχεδιαστής στο σημείο αυτό, έχει την δυνατότητα να επισημειώσει επιπρόσθετα στοιχεία, όπως οι καθυστερήσεις που παρουσιάζονται στις εσωτερικές διασυνδέσεις (interconnection delays) και να βάλει τον Design Compiler να επανασυνθέσει την σχεδίαση προκειμένου η ανάλυση χρονισμού να είναι πιο ακριβής.

Ο Design Compiler διαβάζει και παράγει αρχεία που προσδιορίζουν σχεδιάσεις σε πολλαπλές μορφές που αποτελούν standards στην ηλεκτρονική σχεδίαση ψηφιακών κυκλωμάτων, συμπεριλαμβανομένων των αρχείων της Synopsys .db και .eqn. Επιπρόσθετα ο DC προσφέρει άμεση σύνδεση με άλλα EDA (Electronic Design Automation) εργαλεία. Αυτό μας δίνει την δυνατότητα να μεταφέρουμε με ευκολία από εργαλείο σε εργαλείο τους περιορισμούς και τα αποτελέσματα που έχουμε ανακτήσει μέχρι την προκείμενη στιγμή.

Επόμενο βήμα η αναλυτικότερη εισχώρηση στη μεθοδολογία, τη στρατηγική και το γιατί χρησιμοποιήθηκαν για την σύνθεση και βελτιστοποίηση των κυκλωμάτων μας.

Application Specific Σχεδιασμός και Αποτελέσματα Πειραμάτων

Επιθυμούμε να χτίσουμε ένα κύκλωμα που να λύνει το πρόβλημα του ΑΑΔ, χρησιμοποιώντας αρχιτεκτονικές αθροιστών τις οποίες μελετήσαμε προηγουμένως. Τις αρχιτεκτονικές αυτές θα τις υλοποιήσουμε χρησιμοποιώντας *Designware components* της Synopsys. Έχουμε ήδη μελετήσει τα χαρακτηριστικά του προβλήματος ΑΑΔ, τα χαρακτηριστικά των διάφορων αλγορίθμων άθροισης και τις δυνατότητες των Designware Components. Όλα αυτά θα συνδυαστούν μέσω ακολουθίας της ροής που ήδη έχουμε μελετήσει στο προηγούμενο κεφάλαιο, κατά τη διαδικασία της σύνθεσης.

Λειτουργία Σχεδίασης

Η σχεδίασή μας αποτελεί ένα υβρίδιο τεχνολογιών υλικού. Ας εξετάσουμε τους λόγους που μας οδήγησαν στην επιλογή αυτή αφού πρώτα περιγράψουμε τη λειτουργικότητα του κυκλώματος. Λαμβάνοντας ως εισόδους τα αθροίσματα απολύτων διαφορών, είτε από διαφορετικά modes (intra prediction), είτε από διαφορετικές θέσεις εκτίμησης κίνησης (inter prediction), το κύκλωμά μας μπορεί να συγκρίνει 9 τέτοια νούμερα και να εξάγει αυτό με το καλύτερο ΑΑΔ, δηλαδή το μικρότερο. Στην intra πρόβλεψη, αυτό συμβαίνει απευθείας για όλα τα modes μιας και δεν έχουμε χρονική πρόβλεψη, ενώ στην inter πρόβλεψη μπορούμε να το εκτελούμε επαναληπτικά. Φθάνοντας στο τέλος των εισόδων, ή βρίσκοντας ένα ΑΑΔ με αρκετή ακρίβεια κοντά στο 0.

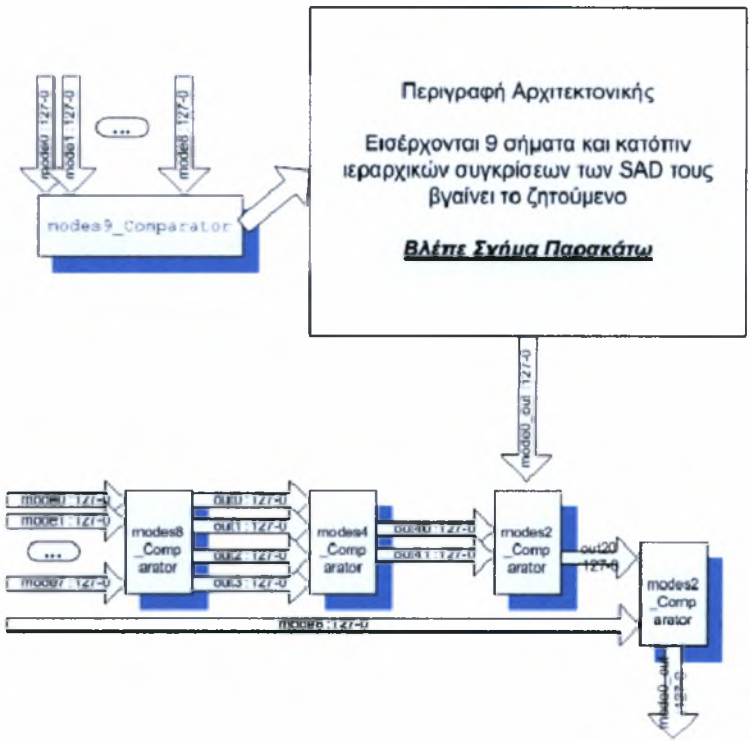
Στην πρόταση των Κοζύρη, Σταμούλη και Κατσαβουνίδη που παρατίθεται στη βιβλιογραφία, το προαναφερθέν πρόβλημα λύνεται με προσαρμοσμένη(custom) σχεδίαση σε χαμηλό επίπεδο (πύλης). Από την

πλευρά μας χρησιμοποιήσαμε ένα απολύτως βιομηχανικό εργαλείο CAD, το DBBIP, για να χτίσουμε μία αντίστοιχη λύση στο ίδιο πρόβλημα, με όμοια αρχιτεκτονική σε υψηλό επίπεδο(RTL) με την παραπάνω πρόταση. Σκοπός μας είναι να αποδείξουμε πως **πολύ δύσκολα** μία αυτοματοποιημένη σχεδίαση μπορεί να εξισωθεί σε μία προσαρμοσμένη εξαρχής στους περιορισμούς σχεδίαση. Κι αυτό έπειτα από την πιο σκληρή βελτιστοποίηση. Από την άλλη πλευρά μία όμοια αρχιτεκτονική γραμμένη με το χέρι, θα είναι σίγουρα **όχι καλύτερη** από την σχεδιασμένη με CAD.

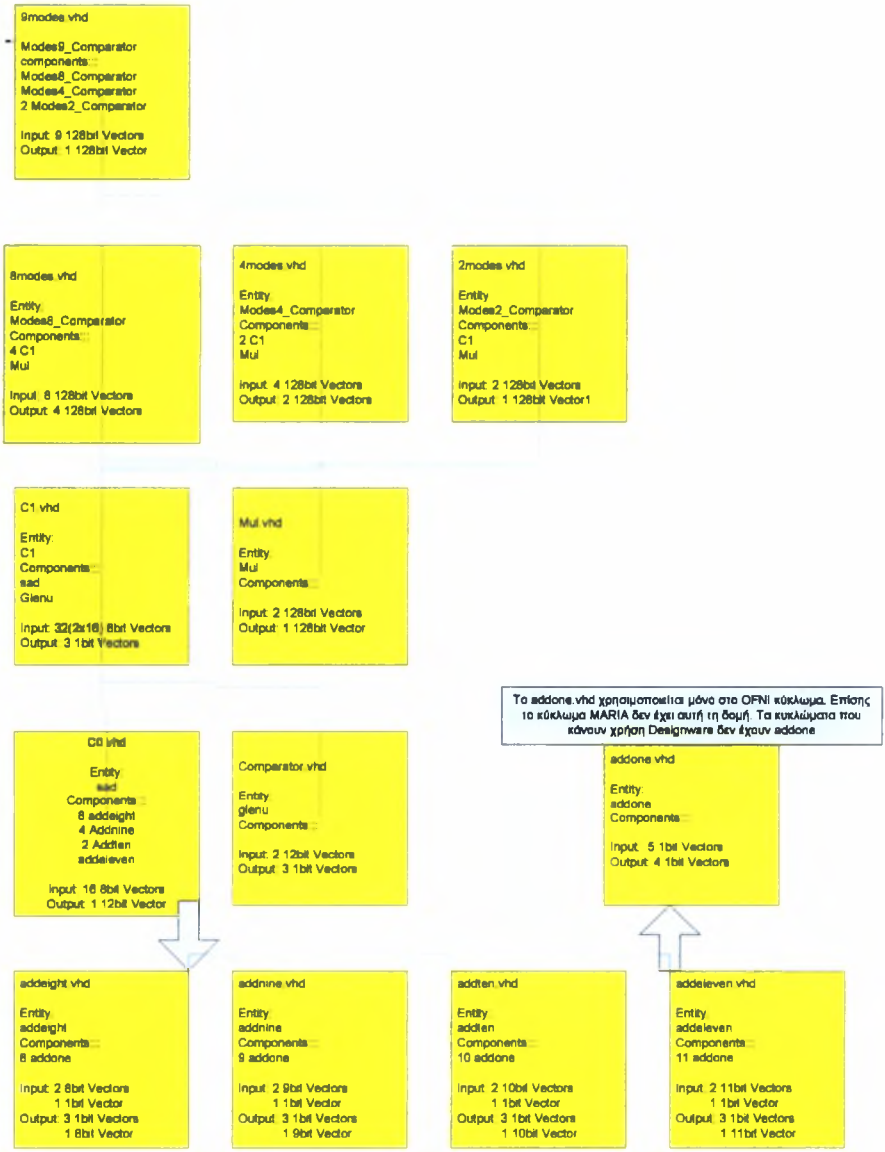
Στο τελευταίο βήμα θα συγκρίνουμε τις αποδόσεις των κυκλωμάτων μας ώστε να επαληθεύσουμε τα θεωρητικά συμπεράσματα των αθροιστικών σχημάτων. Υπενθυμίζουμε πως το κύκλωμά μας έχει ιδιαίτερα χαρακτηριστικά αθροιστών σε ό,τι αφορά το μέγεθος λέξης αλλά και σε ό,τι αφορά τα επιπλέον στοιχεία που βρίσκονται εντός αυτού.

Δομή Σχεδίασης

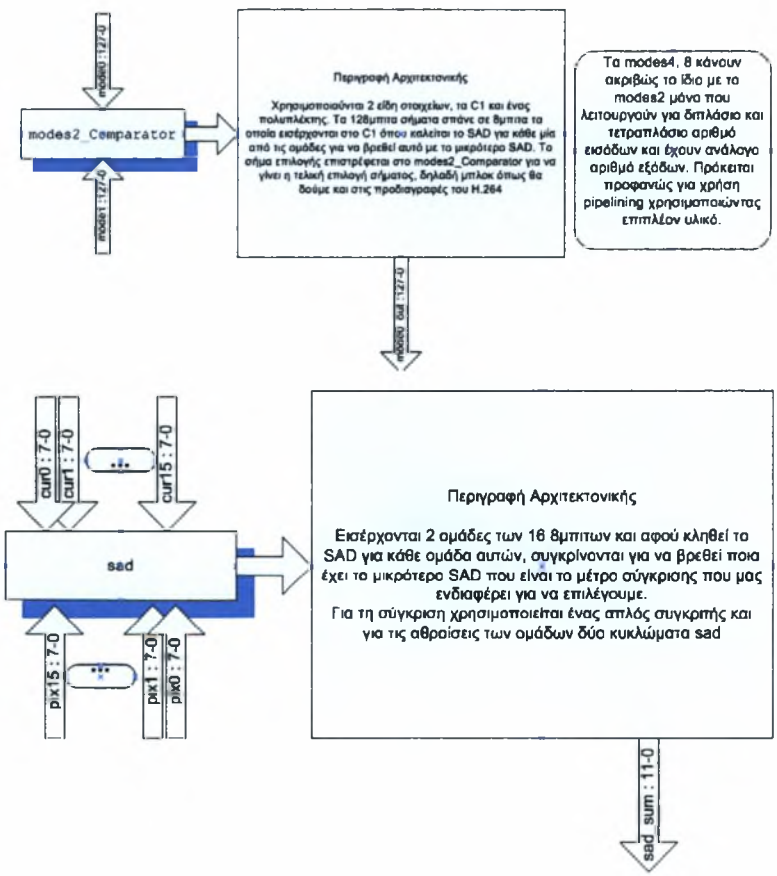
Η δομή του κυκλώματος που χρησιμοποιούμε ακολουθεί στην επόμενη σελίδα.:

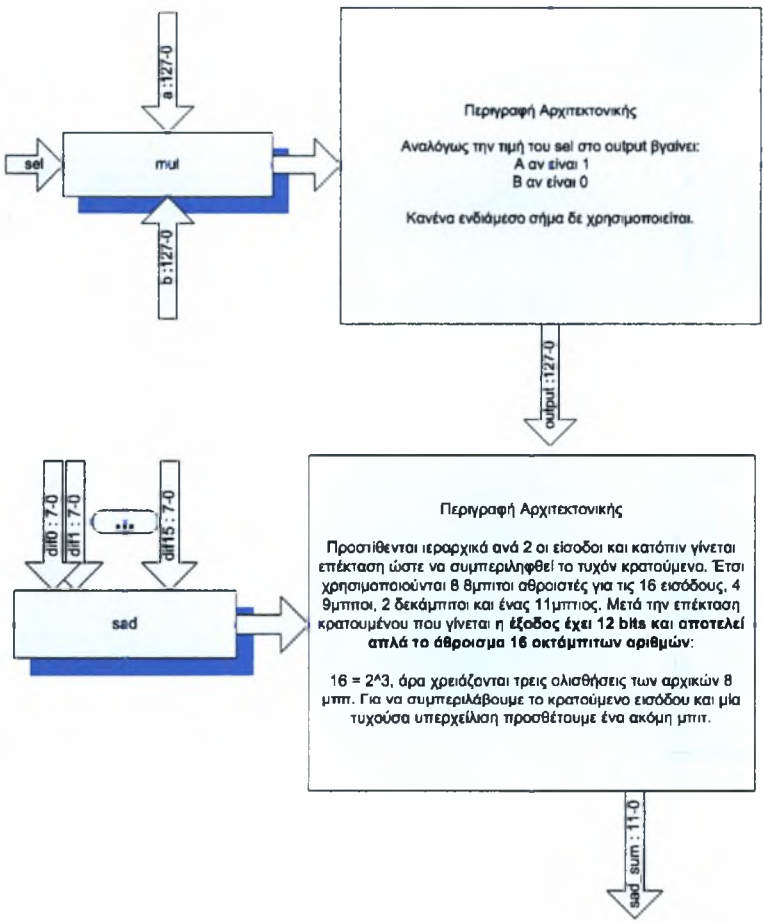


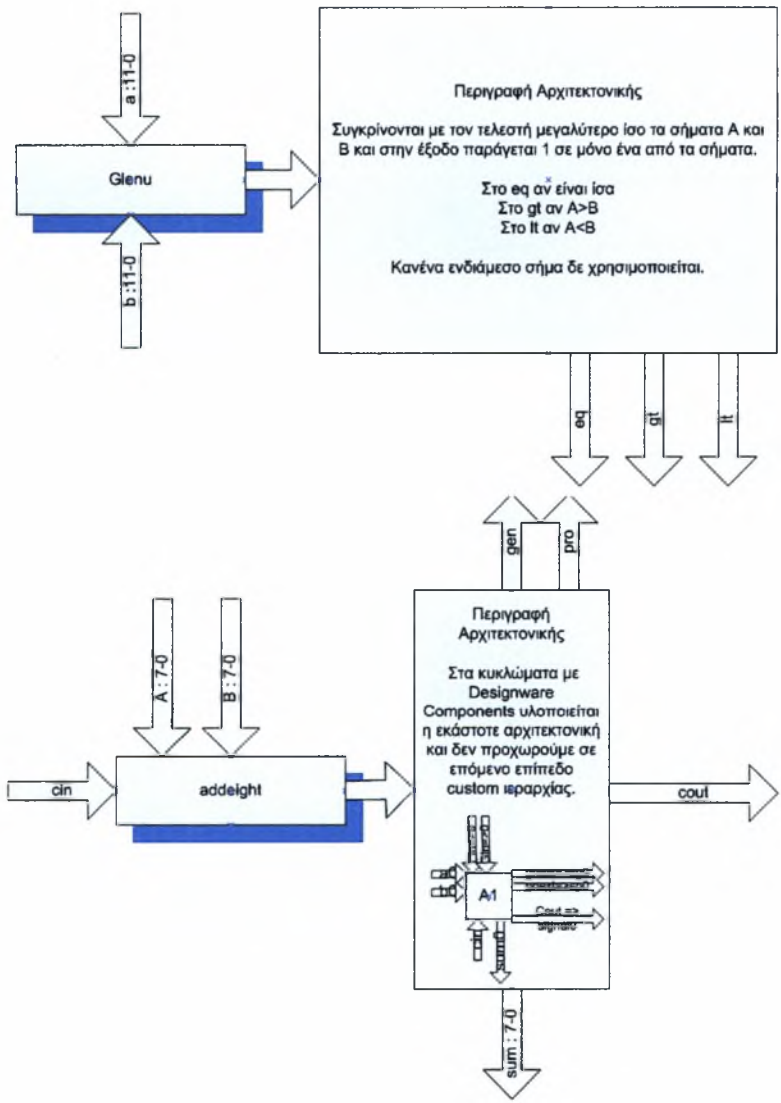
Ας διεισδύσουμε στη δομή του κώδικα:


9modes.vhd
Modes9_Comparator
components:
Modes8_Comparator
Modes4_Comparator
2 Modes2_Comparator
Input: 9 128bit Vectors
Output: 1 128bit Vector
8modes.vhd
Entity:
Modes8_Comparator
Components:
4 C1
Mul
Input: 8 128bit Vectors
Output: 4 128bit Vectors
4modes.vhd
Entity:
Modes4_Comparator
Components:
2 C1
Mul
Input: 4 128bit Vectors
Output: 2 128bit Vectors
2modes.vhd
Entity:
Modes2_Comparator
Components:
C1
Mul
Input: 2 128bit Vectors
Output: 1 128bit Vector1
C1.vhd
Entity:
C1
Components:
sad
Glenu
Input: 32(2x16) 8bit Vectors
Output: 3 1bit Vectors
Mul.vhd
Entity:
Mul
Components:
Input: 2 128bit Vectors
Output: 1 128bit Vector
Το addone.vhd χρησιμοποιείται μόνο στο OFNI κύκλωμα. Επίσης το κύκλωμα MARIA δεν έχει αυτή τη δομή. Τα κυκλώματα που κάνουν χρήση Designware δεν έχουν addone
C0.vhd
Entity:
sad
Components:
8 addeight
4 Addnine
2 Addten
addeleven
Input: 16 8bit Vectors
Output: 1 12bit Vector
Comparator.vhd
Entity:
glenu
Components:
Input: 2 12bit Vectors
Output: 3 1bit Vectors
addone.vhd
Entity:
addone
Components:
Input: 5 1bit Vectors
Output: 4 1bit Vectors
addeight.vhd
Entity:
addeight
Components:
8 addone
Input: 2 8bit Vectors
1 1bit Vector
Output: 3 1bit Vectors
1 8bit Vector
addnine.vhd
Entity:
addnine
Components:
9 addone
Input: 2 9bit Vectors
1 1bit Vector
Output: 3 1bit Vectors
1 9bit Vector
addten.vhd
Entity:
addten
Components:
10 addone
Input: 2 10bit Vectors
1 1bit Vector
Output: 3 1bit Vectors
1 10bit Vector
addeleven.vhd
Entity:
addeleven
Components:
11 addone
Input: 2 11bit Vectors
1 1bit Vector
Output: 3 1bit Vectors
1 11bit Vector

Παρακάτω περιγράφεται η λειτουργία των οντοτήτων και των αρχιτεκτονικών, η αντιστοιχία των οποίων είναι 1 προς 1, όπως επίσης συμβαίνει με τα αρχεία και τις οντότητες.



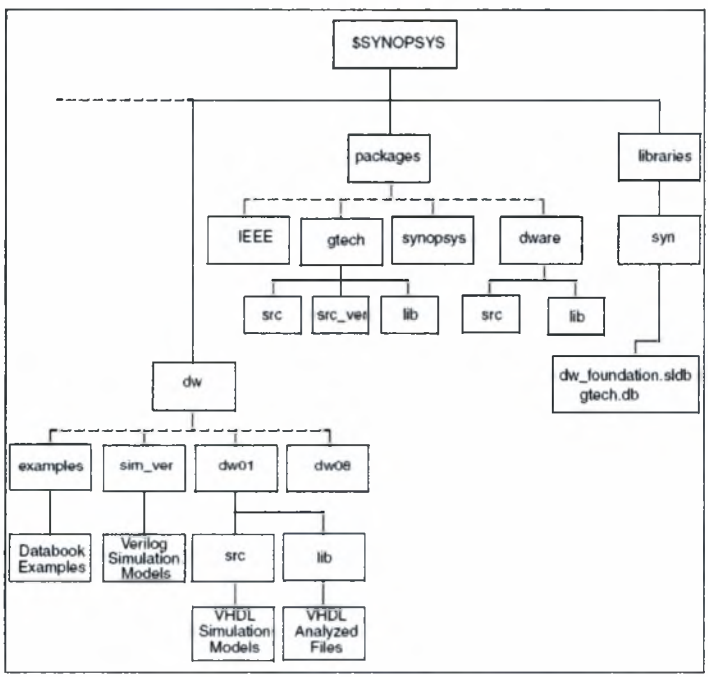

a :127-0
sel
mul
b :127-0
Περιγραφή Αρχιτεκτονικής
Αναλόγως την τιμή του sel στο output βγαίνει:
A αν είναι 1
B αν είναι 0
Κανένα ενδιάμεσο σήμα δε χρησιμοποιείται.
output :127-0
dif0 : 7-0
dif1 : 7-0
dif15 : 7-0
sad
Περιγραφή Αρχιτεκτονικής
Προστίθενται ιεραρχικά ανά 2 οι είσοδοι και κατόπιν γίνεται επέκταση ώστε να συμπεριληφθεί το τυχόν κρατούμενο. Έτσι χρησιμοποιούνται 8 8μπιτοι αθροιστές για τις 16 εισόδους, 4 9μπιτοι, 2 δεκάμπιτοι και ένας 11μπτιος. Μετά την επέκταση κρατουμένου που γίνεται η **έξοδος έχει 12 bits και αποτελεί απλά το άθροισμα 16 οκτάμπιτων αριθμών**:
16 = 2^3, άρα χρειάζονται τρεις ολισθήσεις των αρχικών 8 μπτ. Για να συμπεριλάβουμε το κρατούμενο εισόδου και μία τυχούσα υπερχείλιση προσθέτουμε ένα ακόμη μπιτ.
sad_sum : 11-0



Ομοίως περιγράφονται και τα addnine, addten, addeleven.

Designware Components

Η περιγραφή των κοινότυπων βημάτων περιγραφής του τρόπου συγγραφής και της προσομοίωσης θεωρείται τετριμμένη. Θα αναφερθούμε στο επόμενο στοιχείο που εισάγει η εργασία αυτή, τα Designware Components.



Δομή Φακέλων Synopsys

Για να έχουμε πρόσβαση και να χρησιμοποιήσουμε το DBBIP στη σύνθεση της Synopsys και στα εργαλεία προσομοίωσης, πρέπει να αποκτήσουμε πρόσβαση στις DBBIP συνθετικές βιβλιοθήκες. Αυτό γίνεται θέτοντας τις μεταβλητές του dc_shell-t *synthetic_libary* και

link_library. (Δεν μπορούμε να παραθέσουμε παράδειγμα ενός DBBIP μιας και είναι σε binary μορφή).

Πρόσβαση στις Συνθετικές Βιβλιοθήκες

Οι συνθετικές βιβλιοθήκες της DBBIP περιέχουν τα συνθετικά modules, τους τελεστές και τους συνδέσμους που ενώνουν το DBBIP με τα εργαλεία σύνθεσης της Synopsys. Παρατίθεται ο κώδικας που υποδεικνύει πώς θέτουμε τις απαιτούμενες μεταβλητές για τις ανεξάρτητες βιβλιοθήκες DBBIP DW01, DW02, DW03, DW04, DW06, DW08. Όλα τα DBBIP συμπεριλαμβάνονται όταν χρησιμοποιούμε την *dw_foundation*. Οι μεταβλητές δεν αναθέτονται από μόνες τους! Εκτελούμε λοιπόν το παρακάτω, ή το τοποθετούμε σε ένα script ή και στο .synopsys_dc.setup αρχείο:

```
set synthetic_library [list dw_foundation.sldb]
set link_library [concat $target_library $synthetic_library]
set search_path [concat $search_path [list \
[format "%s%s" $synopsys_root "/dw/sim_ver"]]]
set synlib_wait_for_design_license [list "DesignWare"]
```

Εμείς επιλέξαμε να τοποθετήσουμε τις γραμμές αυτές στο .synopsys_dc.setup :

```
set synthetic_library
usr/synopsys/DeCo/libraries/syn/dw_foundation.sldb
set link_library [concat $target_library $synthetic_library]
set search_path [concat $search_path [list [format "%s%s"
/usr/synopsys/DeCo "/dw/sim_ver" ]]]
set set_synlib_wait_for_design_license "Designware"
```

Μπορούμε να προσπελάσουμε το DBBIP μέσω του HDL κώδικα, Verilog ή VHDL. Οι μηχανισμοί είναι δύο : *inferencing(συμπερασμός)* και *instantiation(συγκεκριμενοποίηση)*.

Χρησιμοποιώντας τον DeCo, οποιοδήποτε DBBIP μπορεί να χρησιμοποιηθεί μέσω instantiation στον HDL κώδικα. Μπορούμε επίσης να συμπεράνουμε(infer) όλα τα Συνδυαστικά IP (Combinational IP) με τους HDL τελεστές και/ή συναρτήσεις. Παρόλα αυτά δεν μπορούμε να χρησιμοποιήσουμε inferring για ακολουθιακά στοιχεία στον DeCo.

Χρησιμοποιώντας τον Behavioral Compiler (BeCo), μπορούμε να συμπεράνουμε(infer) όλο τα Combinational IP, όπως και στον DeCo. Επιπρόσθετα, για πολλά ακολουθιακά DBBIP, υπάρχουν ειδικές συναρτήσεις που μπορούμε να συμπεράνουμε το IP στον κώδικά μας. Παρόλα αυτά, δεν μπορούμε να συγκεκριμενοποιήσουμε(instantiate) DBBIP στον BeCo.

Προς το παρόν δεν υπάρχει γραφικό εργαλείο για τη χρήση των DBBIP..

Συμπερασμός τελεστή(Operator Inferencing)

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity DW01_add_oper is
        generic(wordlength: integer := 8);
        port(in1, in2 : in STD_LOGIC_VECTOR(wordlength-1 downto 0);
                sum : out STD_LOGIC_VECTOR(wordlength-1 downto 0));
end DW01_add_oper;

architecture oper of DW01_add_oper is
                signal in1_signed, in2_signed, sum_signed: SIGNED(wordlength-1 downto
                0);
begin
        in1_signed <= SIGNED(in1);
        in2_signed <= SIGNED(in2);
        -- infer the "+" addition operator
        sum_signed <= in1_signed + in2_signed;
        sum <= STD_LOGIC_VECTOR(sum_signed);
end oper;
```

Συμπερασμός Συνάρτησης(Function Inferencing)

```
library IEEE, DW02;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use DW02.DW02_components.all;

entity DW02_sin_func is
        generic(func_A_width, func_sin_width: INTEGER := 8);
        port(func_A : in std_logic_vector(func_A_width-1 downto 0);
        SIN_func_TC : out std_logic_vector(func_sin_width-1 downto 0);
        SIN_func_UNS : out std_logic_vector(func_sin_width-1 downto 0) );
end DW02_sin_func;

architecture func of DW02_sin_func is
begin
        SIN_func_TC <= std_logic_vector(DWF_sin(SIGNED(func_A), func_sin_width));
        SIN_func_UNS <= std_logic_vector(DWF_sin(UNSIGNED(func_A), func_sin_width));
end func;
```

Συγκεκριμενοποίηση DesignWare (Instantiating DesignWare)

```
library IEEE,DWARE,DW01;
use IEEE.std_logic_1164.all;
use DWARE.DWpackages.all;
use DW01.DW01_components.all;

entity DW01_add_inst is
        generic ( inst_width : NATURAL := 8 );
        port ( inst_A : in std_logic_vector(inst_width-1 downto 0);
               inst_B : in std_logic_vector(inst_width-1 downto 0);
               inst_CI : in std_logic;
               SUM_inst : out std_logic_vector(inst_width-1 downto 0);
               CO_inst : out std_logic );
end DW01_add_inst;

architecture inst of DW01_add_inst is
begin
        -- Instance of DW01_add
        U1 : DW01_add
               generic map ( width => inst_width )
               port map ( A => inst_A, B => inst_B, CI => inst_CI,
               SUM => SUM_inst, CO => CO_inst );
end inst;

-- pragma translate_off
configuration DW01_add_inst_cfg_inst of DW01_add_inst is
        for inst
        end for; -- inst
end DW01_add_inst_cfg_inst;
-- pragma translate_on
```

Εμείς χρησιμοποιήσαμε instantiation. Παραθέτουμε το εξής ενδεικτικό παράδειγμα από τη σχεδίασή μας:

```
library IEEE, DW01, synopsys;
use IEEE.std_logic_1164.all;
use DW01.DW01_components.all;
use synopsys.attributes.all;

entity addeleven is
        generic(wordlength: integer := 11);
        port(a, b: in STD_LOGIC_VECTOR(wordlength-1 downto 0);
                cin: in STD_LOGIC;
                sum: out STD_LOGIC_VECTOR(wordlength-1 downto 0);
        cout: out STD_LOGIC);
end addeleven;

architecture inst of addeleven is
        attribute implementation: STRING;
        attribute implementation of U1 : label is "cla";
begin
        U1: DW01_add
        generic map(width => wordlength)
        port map(A => a, B => a, CI => cin, SUM => sum, CO => cout);
end inst;
```

Θέτοντας τις παραμέτρους που φαίνονται στις προδιαγραφές μπορούμε να επιλέξουμε το ζητούμενο. Για παράδειγμα παρατίθεται τμήμα του specification για αθροιστές το οποίο και χρησιμοποιήσαμε το διπλανό πίνακα οδηγιών.

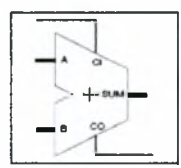
DW01_add

Adder

Last Revised: Release DWF_0212

Features and Benefits

- Parameterized word length
- Carry-in and carry-out signals
- Module Compiler Architectures



Description

DW01_add adds two operands A and B with a carry-in CI to produce the output SUM with a carry-out CO.

Table 1: Pin Description

Pin Name	Width	Direction	Function
A	width bits	Input	Input data
B	width bits	Input	Input data
CI	1 bit	Input	Carry-in
SUM	width bits	Output	Sum of (A + B + CI)
CO	1 bit	Output	Carry-out

Table 2: Parameter Description

Parameter	Values	Description
width	≥1	Word length of A, B, and SUM

Σύνθεση

Προτού προβούμε σε σύνθεση πρέπει να καθορίσουμε τους περιορισμούς, όποιοι και να είναι αυτοί. Υπάρχουν δύο τύποι περιορισμών:

Design rule constraints

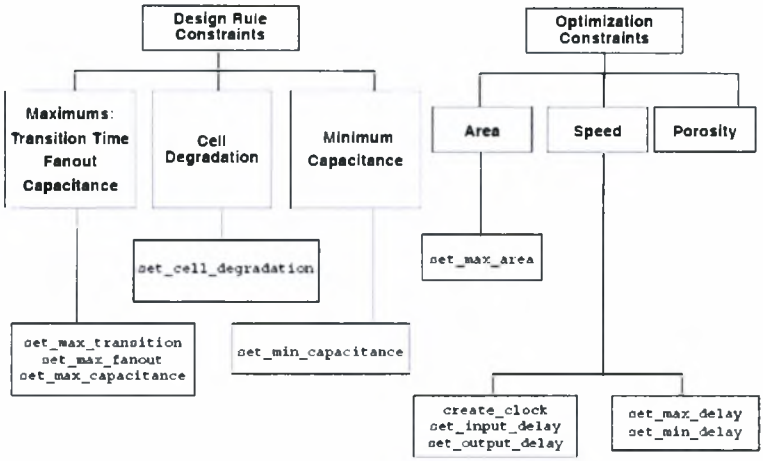
Οι εγγενείς περιορισμοί της σχεδίασης οι οποίοι καθορίζονται από την τεχνολογική βιβλιοθήκη. Είναι απαραίτητοι για να λειτουργεί μία σχεδίαση σωστά και εφαρμόζονται σε κάθε σχεδίαση που χρησιμοποιεί τη συγκεκριμένη τεχνολογική βιβλιοθήκη. Θα έπρεπε να είναι περισσότερο αυστηροί από τους περιορισμούς βελτιστοποίησης.

Optimization Constraints

Είναι περιορισμοί εξωτερικοί, που ορίζονται από το σχεδιαστή. Οι περιορισμοί βελτιστοποίησης εφαρμόζονται στη σχεδίαση που δουλεύουμε κατά τη διάρκεια του dc_shell session και να αναπαριστούν τους στόχους της σχεδίασης. Πρέπει να είναι ρεαλιστικοί.

Έτσι ο DeCo υπολογίζει κάθε φορά δύο συναρτήσεις κόστους: μία για τους περιορισμούς κανόνων σχεδίασης (design rule constraints) και μία για τους περιορισμούς βελτιστοποίησης. Κατά τη βελτιστοποίηση επιπέδου Gale, ο DeCo αναφέρει την τιμή κάθε συνάρτησης κόστους όποτε συμβαίνει μία αλλαγή στη σχεδίαση.

Οι περιορισμοί καθορίζονται διαδραστικά ή σε ένα αρχείο περιορισμών.



Παραπάνω παρατίθενται οι κύριες κατηγορίες περιορισμών, τις οποίες δεν πρόκειται να αναλύσουμε μιας και δε μας απασχολούν στη συνέχεια.

Οι κύριες κατηγορίες περιορισμών που μας αφορούν ανήκουν στην κατηγορία των περιορισμών βελτιστοποίησης. Αυτοί είναι συγκεκριμένα:

Μέγιστη καθυστέρηση

Καθορίζει και το κρίσιμο μονοπάτι στο κύκλωμα το οποίο αναφερόμαστε. Ο DeCo περιέχει έναν ενσωματωμένο στατικό αναλυτή χρονισμού για να υπολογίζει τους περιορισμούς χρονισμού. Ο στατικός αναλυτής χρονισμού υπολογίζει τις καθυστερήσεις μονοπατιών κάνοντας χρήση των τοπικών πυλών και των καθυστερήσεων διασυνδέσεων αλλά δεν προσομοιώνει τη σχεδίαση. Το κρίσιμο μονοπάτι το οποίο βρίσκει ο DeCo δε θα είναι απαραίτητα το μεγαλύτερο συνδυαστικό μονοπάτι σε ένα ακολουθιακό κύκλωμα, αφού τα μονοπάτια μπορούν να είναι σχετικά με διάφορα ρολόγια σε αρχές και τέλη μονοπατιών.

Υπολογισμός Κόστους

Η μέγιστη καθυστέρηση είναι συνήθως ο πιο σημαντικός όρος της συνάρτησης βελτιστοποίησης κόστους. Το κόστος της βελτιστοποίησης μέγιστης καθυστέρησης οδηγεί τον DeCo να παράγει μία σχεδίαση που λειτουργεί στην επιθυμητή ταχύτητα. Το κόστος της μέγιστης καθυστέρησης περιλαμβάνει πολλά τμήματα. Παρατίθεται η εξίσωση κόστους. Οι τιμές μέγιστης καθυστέρησης στις οποίες στοχεύουμε καθορίζονται αυτόματα αφού ληφθούν υπόψη κάποιες τιμές καθυστερήσεων διαφόρων προελεύσεων και τελικά η εντολή **set_max_delay**. Η μέγιστη καθυστέρηση εξαρτάται άμεσα από το ποια μονοπάτια ομαδοποιούνται μεταξύ τους(εντολές *group* και *ungroup*), αλλά και από το πού θεωρούμε πως υπάρχει το ρολόι και πότε αυτό ξεκινά.

$$\sum_{l=1}^{m} v_l \cdot w_l$$

l = Index
v = worst violation
m = number of path groups
w = weight

Μέγιστο εμβαδό

Το μέγιστο εμβαδό αναπαριστά **τον αριθμό πυλών** στη σχεδίαση, όχι το φυσικό εμβαδό που καταλαμβάνει η σχεδίαση.

Συνήθως οι περιορισμοί εμβαδού για τη σχεδίαση δηλώνονται ως η μικρότερη σχεδίαση που ικανοποιεί τις επιδόσεις που έχουμε ως στόχο. Ορίζοντας μέγιστο εμβαδό κατευθύνει τον DeCo να βελτιστοποιήσει τη σχεδίαση για εμβαδό **αφού** τελειώσει η βελτιστοποίηση χρονισμού.

Η εντολή **set_max_area** προδιαγράφει το μέγιστο επιτρεπτό εμβαδό για την παρούσα σχεδίαση. Ο DeCo υπολογίζει το εμβαδό μίας σχεδίασης αθροίζοντας τις περιοχές κάθε στοιχείου στο χαμηλότερο επίπεδο της ιεραρχίας σχεδίασης (και το εμβαδό των δικτύων).

Αγνοεί :

- Άγνωστα στοιχεία

- Στοιχεία με άγνωστα εμβαδά
- Τεχνολογικά ανεξάρτητα γενικευμένα κελιά

Το εμβαδό ενός κελιού(στοιχείου) είναι τεχνολογικά-εξαρτημένο και παρέχεται από την τεχνολογική βιβλιοθήκη.

Υπολογισμός κόστους

Η εξίσωση κόστους μέγιστου εμβαδού είναι :

Cost = max (0, παρόν εμβαδό - μέγιστο εμβαδό)

Προτεραιότητες περιορισμών

Κατά τη βελτιστοποίηση, ο DeCo χρησιμοποιεί ένα διάνυσμα κόστους για να λύσει οποιεσδήποτε διενέξεις ανάμεσα στις ανταγωνιζόμενες προτεραιότητες των περιορισμών. Η προεπιλεγμένη προτεραιότητα είναι :

Priority (descending order)	Notes
connection classes	
multiple_port_net_cost	
min_capacitance	Design Rule Constraint
max_transition	**Design Rule Constraint**
max_fanout	**Design Rule Constraint**

max_capacitance	**Design Rule Constraint**
cell_degradation	**Design Rule Constraint**
max_delay	**Optimization Constraint**
min_delay	**Optimization Constraint**
power	Optimization Constraint
area	Optimization Constraint
cell count	

Ο πίνακας αυτός δείχνει πως εξορισμού οι κανόνες σχεδίασης έχουν προτεραιότητα έναντι των περιορισμών βελτιστοποίησης. Όμως έχουμε τη δυνατότητα να αναδιατάξουμε τις προτεραιότητες των περιορισμών που βλέπουμε παραπάνω με έντονα γράμματα, χρησιμοποιώντας την εντολή **set_cost_priority**.

Θέτοντας τους περιορισμούς

Οι προκαθορισμένες τιμές των παραμέτρων είναι μη ρεαλιστικές κι έτσι τα αποτελέσματα μίας βελτιστοποίησης κυκλωμάτων χωρίς αλλαγμένες παραμέτρους πιθανότατα δε θα είναι βέλτιστες.

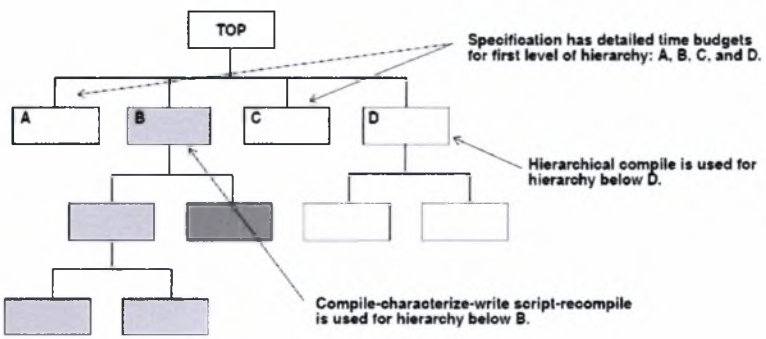
Για να καθορίσουμε ρεαλιστικούς στόχους σχεδίασης και αν η σχεδίαση βρίσκεται ήδη σε Gale συνδεσμολογία μπορούμε να εξάγουμε τους περιορισμούς χρονισμού με την εντολή **derive_timing_constraints**. Αντίστοιχα για να βρούμε ένα δείγμα περιορισμών εμβαδού κυκλώματος θέτουμε set_max_area 0 δίχως άλλο περιορισμό και συνθέτουμε τη σχεδίαση. Αυτή η μέθοδος θα μας αποδώσει τη βέλτιστη σε εμβαδό σχεδίαση που μπορούμε να επιτύχουμε ώστε να μπορούμε να καθορίσουμε εύκολα τον περιορισμό εμβαδού.

Στρατηγική

Πριν εκτελέσουμε τη διαδικασία της σύνθεσης πρέπει να επιλέξουμε τη στρατηγική μας. Μία ιεραρχική σχεδίαση μπορεί να επεξεργαστεί με :

- Top-down ιεραρχική επεξεργασία
- Bottom-up στρατηγική ή αλλιώς compile-characterize-write script-recompile.

Επίσης υπάρχει η δυνατότητα ανάμειξης των δύο στρατηγικών. Δίνουμε ένα παράδειγμα:



Top-down στρατηγική

Ο DeCo αυτόματα εκτελεί compile σε ιεραρχικά κυκλώματα χωρίς να διαλύει την ιεραρχία. Αφού επεξεργαστεί κάθε module στη σχεδίαση, ο DeCo συνεχίζει να βελτιστοποιεί το κύκλωμα μέχρι να ικανοποιήσει τους περιορισμούς. Αυτή η διαδικασία συχνά απαιτεί να επεξεργαστούμε ξανά τις σχεδιάσεις στο κρίσιμο μονοπάτι. Η διαδικασία θα σταματήσει όταν θα έχουμε πετύχει τους περιορισμούς ή δε θα μπορούμε να βελτιστοποιήσουμε άλλο.

Ο DeCo διατηρεί την ιεραρχία μίας σχεδίασης. Η top-down ιεραρχική επεξεργασία είναι ένας εύκολος τρόπος με τρία βήματα:

1. διάβασμα ολόκληρης της σχεδίασης

2. εφαρμογή των περιορισμών και των παραμέτρων στο κορυφαίο επίπεδο. Οι περιορισμοί και οι παράμετροι βασίζονται στις προδιαγραφές της σχεδίασης.

3. Compile

Όλες οι παράμετροι ορίζονται σύμφωνα με την top-level σχεδίαση αλλά ταυτόχρονα λαμβάνονται υπόψη και διάφορες εξαρτήσεις μεταξύ των σχεδιάσεων. Η μέθοδος αυτή δεν συνίσταται για μεγάλες σχεδιάσεις διότι όλες οι σχεδιάσεις πρέπει να διαβαστούν στην μνήμη ταυτόχρονα.

Bottom-up Στρατηγική

Στην άλλη στρατηγική, την bottom-up, όλες οι υπό-σχεδιάσεις δέχονται παραμέτρους και προσομοιώνονται ξεχωριστά. Η κάθε σχεδίαση αφού προσομοιωθεί επιτυχώς κλειδώνεται με την εντολή *dont_touch* για να μην αλλαχθεί στις προσομοιώσεις που θα ακολουθήσουν. Αφού όλες οι υπό-σχεδιάσεις προσομοιωθούν επιτυχώς προσομοιώνονται για να συνθέσουν τις υψηλότερες σε ιεραρχία σχεδιάσεις μέχρι την top-level σχεδίαση. Η μέθοδος αυτή προτιμάται όταν η σχεδίαση μας είναι μεγάλη αφού δεν χρειάζεται ο DC να διαβάζει τις σχεδιάσεις όλες μαζί στην μνήμη.

Έχει τα εξής μειονεκτήματα :

- Απαιτεί επαναλήψεις μέχρι να γίνει σταθερή η διεπαφή.
- Απαιτεί χειροκίνητο έλεγχο επίβλεψης

Διεξάγεται σε 7 βήματα:

1. Κάνουμε compile κάθε υπομπλόκ ξεχωριστά, χρησιμοποιώντας εκτιμήσεις για τις παραμέτρους βελτιστοποίησης.

2. Διαβάζουμε ολόκληρη τη μεταεπεξεργασμένη σχεδίαση.

3. Χαρακτηρίζουμε ένα subblock

4. Χρησιμοποιούμε την εντολή write_script για να σώσουμε την πληροφορία από το χαρακτηρισμό

5. Αδειάζουμε τη μνήμη από παλιά κατάλοιπα και διαβάζουμε τα υπομπλοκ που απαιτούνται.

6. Με τα νέα subblocks ξανατρέξτε όλη τη σχεδίαση μαζί.

7. Επιλέξτε άλλο subblock, μέχρι όλα τα subblocks να έχουν επαναεπεξεργαστεί.

- Αν δεν τροποποιήσετε τον αρχικό RTL κώδικα μπορείτε να το σώσετε σε ένα .ddc αρχείο. Κερδίζεται χρόνος για αργότερα όταν και θα αιτηθεί να ξαναδιαβάσουμε την ίδια σχεδίαση.
- Η εντολή compile πηγαίνει αυτόματα στο υπομπλοκ.
- Το compile είναι bottom up
- Το characterize είναι top down.

Επεξεργαζόμενοι τον DeCo:

Για να μπορούμε να δουλέψουμε με τον DeCo πρέπει να έχουμε ρυθμίσει τις μεταβλητές οι οποίες θα δείχνουν στα αρχεία βιβλιοθηκών. Η τεχνολογική βιβλιοθήκη μας λέγεται gsclib90 και το αντίστοιχο αρχείο slow.db .

Η βιβλιοθήκη αυτή είναι πνευματική ιδιοκτησία της εταιρίας Cadence και χρησιμοποιεί μία διαδικασία 125C και 0.9V στα 90nm.

Πρέπει με κάποιο τρόπο να εισάγουμε στο εργαλείο την πληροφορία της βιβλιοθήκης. Αυτό γίνεται τροποποιώντας το αρχείο που προσπαθεί κάθε φορά να φορτώσει ο DeCo, το .synopsys_dc.setup:

```
set search_path        {. /home/dzaharis/gsclib090}

set target_library { slow.db}

set synthetic_library /usr/synopsys/DeCo/libraries/syn/dw_foundation.sldb

set link_library [concat $target_library $synthetic_library]

set search_path [concat $search_path [list [format "%s%s" /usr/synopsys/DeCo
"/dw/sim_ver" ]]]
set set_synlib_wait_for_design_license "Designware"
set designer "dimItris zacharIs"

sh rm -r ./WORK
```

Αναλυτικά, η μεταβλητή *search_path* δίνει στον DeCo ποιους φακέλους, εκτός από αυτούς του κεντρικού καταλόγου της SYnopsys, πρέπει να διαβάζει. Το φάκελο από όπου καλέσαμε το κέλυφος του DeCo (.) , και το φάκελο όπου βρίσκεται η τεχνολογική βιβλιοθήκη. Κατόπιν θέτουμε τη μεταβλητή που δείχνει στην τεχνολογική βιβλιοθήκη, την *target_library* στο αντίστοιχο αρχείο. Ορίζουμε όπως προαναφέραμε τις μεταβλητές που συνδέουν αρχεία, τελεστές και συνθετικά στοιχεία. Τέλος αφαιρούμε ό,τι έχει μείνει στο φάκελο όπου δουλεύουμε τις σχεδιάσεις μας με τον DeCo. Αυτή την πράξη την εκτελούμε στο αρχείο αυτό ώστε ποτέ να μην ξεχνούμε παλαιότερα δεδομένα μέσα στο φάκελο εργασίας.

Αρχικά θα περιγράψουμε τα κυκλώματα που χρησιμοποιήθηκαν. Αυτά ήταν :

- η προσαρμοσμένη σχεδίαση της πρότασης των Κοζύρη, Σταμούλη και Κατσαβουνίδη (**Maria**)
- η υλοποίηση της ίδιας αρχιτεκτονικής χρησιμοποιώντας για την πράξη της σύγκρισης στο τελευταίο επίπεδο έναν Carry Lookahead αθροιστή, γραμμένο από σχεδιαστή (**Ofni**)
- η υλοποίηση της αρχιτεκτονικής χρησιμοποιώντας αρχιτεκτονικές αθροιστών οι οποίες ήταν:

- Αυτόματη επιλογή από τον DeCo της αρχιτεκτονικής αθροιστή ως υποκύκλωμα του ΑΑΔ.(**Auto**)
- Brent-Kung αρχιτεκτονική(**BK**)
- Carry-lookahead αρχιτεκτονική (**CLA**)
- Fast Carry lookahead αρχιτεκτονική (**CLF**)
- Conditional sum αρχιτεκτονική (**CSM**)
- Parallel-Prefix αρχιτεκτονική (**PPARCH**)
- Ripple-Carry select αρχιτεκτονική (**RPCS**)
- Ripple-Carry αρχιτεκτονική (**RPL**)

Στις προδιαγραφές τις οποίες μας δίνει η Synopsys δε μας περιγράφεται ακριβώς το πώς υλοποιούνται μερικές αρχιτεκτονικές όπως η PPARCH. Παρεμπιπτόντως, τα ονόματα στην παρένθεση θα αποτελούν από εδώ και στο εξής τους κωδικούς για κάθε κύκλωμά μας, μιας και αποτελούν την ειδοποιό διαφορά.

Έχουμε ήδη παραθέσει τον τρόπο με τον οποίο καλούμε στοιχεία του DBBIP. Αυτό που απομένει είναι να θέσουμε τους περιορισμούς και τις στρατηγικές μας χρησιμοποιώντας τις εντολές του dc_shell-xg-t που είναι η τελευταία προκαθορισμένη έκδοση του κελύφους του DeCo. Οι εντολές οι οποίες χρησιμοποιήθηκαν περιγράφονται στο παράρτημα. Αρχικά παραθέτω ένα παράδειγμα script με το οποίο καθορίζουμε τους χρονικούς περιορισμούς. Στην προκείμενη περίπτωση για το κύκλωμα **maria**:

```
####maria timing Constraints####

analyze -library WORK -format vhdl {/home/dzaharis/Desktop/STORE/src/MARIA/mul.vhd
/home/dzaharis/Desktop/STORE/src/MARIA/Third_Stage.vhd
/home/dzaharis/Desktop/STORE/src/MARIA/Second_Stage.vhd
/home/dzaharis/Desktop/STORE/src/MARIA/Or3.vhd
/home/dzaharis/Desktop/STORE/src/MARIA/Or.vhd
/home/dzaharis/Desktop/STORE/src/MARIA/Not.vhd
/home/dzaharis/Desktop/STORE/src/MARIA/MyXor4.vhd
/home/dzaharis/Desktop/STORE/src/MARIA/MyXor.vhd
/home/dzaharis/Desktop/STORE/src/MARIA/MyXNor.vhd
/home/dzaharis/Desktop/STORE/src/MARIA/First_Stage.vhd
/home/dzaharis/Desktop/STORE/src/MARIA/Comparator_new.vhd
/home/dzaharis/Desktop/STORE/src/MARIA/Comp3_new.vhd
/home/dzaharis/Desktop/STORE/src/MARIA/Choose.vhd
/home/dzaharis/Desktop/STORE/src/MARIA/C1_new.vhd
/home/dzaharis/Desktop/STORE/src/MARIA/C0_new.vhd
/home/dzaharis/Desktop/STORE/src/MARIA/And4.vhd
/home/dzaharis/Desktop/STORE/src/MARIA/And3.vhd
/home/dzaharis/Desktop/STORE/src/MARIA/And.vhd
/home/dzaharis/Desktop/STORE/src/MARIA/9modes.vhd
/home/dzaharis/Desktop/STORE/src/MARIA/8modes.vhd
/home/dzaharis/Desktop/STORE/src/MARIA/4modes.vhd
/home/dzaharis/Desktop/STORE/src/MARIA/3_bit_Comparison_new.vhd
/home/dzaharis/Desktop/STORE/src/MARIA/3_bit_Comparison.vhd
/home/dzaharis/Desktop/STORE/src/MARIA/2modes.vhd
/home/dzaharis/Desktop/STORE/src/MARIA/2_bit_Comparison.vhd}

elaborate MODES9_COMPARATOR -architecture STRUCTURAL -library WORK

compile -map_effort medium -area_effort medium -ungroup_all


report_constraints > /home/dzaharis/Desktop/OUTPUT/constraintsMARIA_iasdf.txt
report_timing -path full -delay max -nworst 1 -max_paths 1 -significant_digits 2 -
sort_by group > /home/dzaharis/Desktop/OUTPUT/timingMARIA_iasdf.txt
write_script -output /home/dzaharis/Desktop/OUTPUT/MARIA_timing_iasdf.scr
derive_timing_constraints
report_timing -path full -delay max -nworst 1 -max_paths 1 -significant_digits 2 -
sort_by group > /home/dzaharis/Desktop/OUTPUT/timingMARIA_oasdf.txt
write_script -output /home/dzaharis/Desktop/OUTPUT/MARIA_timing_oasdf.scr
remove_design -designs

       sh rm -r *
```

Ας δώσουμε προσοχή στις εντολές. Μας ενδιαφέρει ιδιαίτερα το ότι χρησιμοποιούμε την *derive_timing_constants* σε συνδυασμό με τη *write_script* οι οποίες μας επιτρέπουν να καταγράψουμε τους περιορισμούς που θα χρησιμοποιήσουμε στη συνέχεια. Φυσικά αυτά τα νούμερα μπορούμε να τα τροποποιήσουμε, πράγμα που θα κάνουμε.

Κατόπιν τρέχουμε με μόνο περιορισμό το εμβαδό ώστε να καθορίσουμε τις μικρότερες δυνατές σχεδιάσεις.

Τώρα λοιπόν γνωρίζουμε πού περίπου πρέπει να κυμανθούν οι περιορισμοί μας.

Στο επόμενο βήμα ξεκινάμε ολοκληρωτική σύνθεση. Παραθέτουμε παραδείγματα script που χρησιμοποιήθηκαν για να καλούν εργαλεία και να αποθηκεύουν κατόπιν τις εξόδους.

Α. Δημιουργία δομής φακέλων για εκτέλεση

prepareAll.script

```
#PROETOIMASIA
############################################################
echo "PROETOIMASIA"
echo "############################################################"

#Edw fortwnw tis metablhtes perivallontos Synopsys
source /home/dzaharis/Source\ scripts/.synopsys_variables


#FILE STRUCTURE
############################################################
echo "FILE STRUCTURE"
echo "############################################################"
#dhmiourgw tous ypofakelous gia na tre3oume ta kyklwmata

#kentrikos fakelos EKTELESHS
md /home/dzaharis/Desktop/ToRun


#kentrikos fakelos gia arxeia E3odou
md /home/dzaharis/Desktop/OUTPUT
#ypofakeloi arxitektonikwn
md /home/dzaharis/Desktop/OUTPUT/OFNI
md /home/dzaharis/Desktop/OUTPUT/MARIA
md /home/dzaharis/Desktop/OUTPUT/AUTO
md /home/dzaharis/Desktop/OUTPUT/RPL
md /home/dzaharis/Desktop/OUTPUT/CLA
md /home/dzaharis/Desktop/OUTPUT/CLF
md /home/dzaharis/Desktop/OUTPUT/CSM
md /home/dzaharis/Desktop/OUTPUT/RPCS
md /home/dzaharis/Desktop/OUTPUT/BK
md /home/dzaharis/Desktop/OUTPUT/PPARCH


#DeCo Tech LIbraries
############################################################
echo "setting Tech Libraries"
echo "############################################################"
cp -R /home/dzaharis/Desktop/STORE/lib/gsclib090 /home/dzaharis

#DeCo SETUP FILE INPUT
############################################################
echo "DeCo SETUP FILE INPUT"
echo "############################################################"
cp /home/dzaharis/Desktop/STORE/.setup/.synopsys_dc.setup
/home/dzaharis/Desktop/ToRun/
```

β. Κλήση εργαλείων με παράμετρο τα scripts που έχουμε έτοιμα ανά κύκλωμα

```
echo "FORTWNW METAVLHTES PERIVALLONTOS SYNOPSYS"
source /home/dzaharis/Source\ scripts/.synopsys_variables
#############################################################################
echo "METAVASH SE FAKELO EKTELESHS"
cd /home/dzaharis/Desktop/ToRun
#############################################################################
echo "EVRESH TIMING CONSTRAINTS"
##dc_shell-xg-t -x "source ../STORE/scripts/find_constaints.scr"
##to parapanw to ekana se 3exwristo bhma-script telika
rm -r *
#############################################################################
echo "MARIA"
echo "#execute compile script and write to OUTPUT folder"
md ./OUT
md ./OUT/zero
dc_shell-xg-t -x "source /home/dzaharis/Desktop/STORE/scripts/maria9.scr"
echo "#move output files to the correct folder"
mv ./OUT /home/dzaharis/Desktop/OUTPUT/MARIA/
echo "#remove resting files from execute folder"
rm -r *
#############################################################################
echo "OFNI"
echo "#execute compile script and write to OUTPUT folder"
md ./OUT
md ./OUT/zero
dc_shell-xg-t -x "source /home/dzaharis/Desktop/STORE/scripts/ofni9.scr"
echo "#move output files to the correct folder"
mv ./OUT /home/dzaharis/Desktop/OUTPUT/OFNI/
echo "#remove resting files from execute folder"
rm -r *
#############################################################################
echo "AUTO"
echo "#execute compile script and write to OUTPUT folder"
md ./OUT
md ./OUT/zero
dc_shell-xg-t -x "source /home/dzaharis/Desktop/STORE/scripts/auto9.scr"
echo "#move output files to the correct folder"
mv ./OUT /home/dzaharis/Desktop/OUTPUT/AUTO/
echo "#remove resting files from execute folder"
rm -r *
#############################################################################
echo "RPL"
echo "#execute compile script and write to OUTPUT folder"
md ./OUT
md ./OUT/zero
dc_shell-xg-t -x "source /home/dzaharis/Desktop/STORE/scripts/rpl9.scr"
echo "#move output files to the correct folder"
mv ./OUT /home/dzaharis/Desktop/OUTPUT/RPL/
echo "#remove resting files from execute folder"
rm -r *
#############################################################################
echo "CLA"
echo "#execute compile script and write to OUTPUT folder"
md ./OUT
md ./OUT/zero
dc_shell-xg-t -x "source /home/dzaharis/Desktop/STORE/scripts/cla9.scr"
echo "#move output files to the correct folder"
mv ./OUT /home/dzaharis/Desktop/OUTPUT/CLA/
echo "#remove resting files from execute folder"
rm -r *
```

```
#############################################################################
echo "CLF"
echo "#execute compile script and write to OUTPUT folder"
md ./OUT
md ./OUT/zero
dc_shell-xg-t -x "source /home/dzaharis/Desktop/STORE/scripts/clf9.scr"
echo "#move output files to the correct folder"
mv ./OUT /home/dzaharis/Desktop/OUTPUT/CLF/
echo "#remove resting files from execute folder"
rm -r *
#############################################################################
echo "CSM"
echo "#execute compile script and write to OUTPUT folder"
md ./OUT
md ./OUT/zero
dc_shell-xg-t -x "source /home/dzaharis/Desktop/STORE/scripts/csm9.scr"
echo "#move output files to the correct folder"
mv ./OUT /home/dzaharis/Desktop/OUTPUT/CSM/
echo "#remove resting files from execute folder"
rm -r *
#############################################################################
echo "RPCS"
echo "#execute compile script and write to OUTPUT folder"
md ./OUT
md ./OUT/zero
dc_shell-xg-t -x "source /home/dzaharis/Desktop/STORE/scripts/rpcs9.scr"
echo "#move output files to the correct folder"
mv ./OUT /home/dzaharis/Desktop/OUTPUT/RPCS/
echo "#remove resting files from execute folder"
rm -r *
#############################################################################
echo "BK"
echo "#execute compile script and write to OUTPUT folder"
md ./OUT
md ./OUT/zero
dc_shell-xg-t -x "source /home/dzaharis/Desktop/STORE/scripts/bk9.scr"
echo "#move output files to the correct folder"
mv ./OUT /home/dzaharis/Desktop/OUTPUT/BK/
echo "#remove resting files from execute folder"
rm -r *
#############################################################################
echo "PPARCH"
echo "#execute compile script and write to OUTPUT folder"
md ./OUT
md ./OUT/zero
dc_shell-xg-t -x "source /home/dzaharis/Desktop/STORE/scripts/pparch9.scr"
echo "#move output files to the correct folder"
mv ./OUT /home/dzaharis/Desktop/OUTPUT/PPARCH/
echo "#remove resting files from execute folder"
rm -r *
#############################################################################
echo "TELOSSSSSSSS"
```

Παρατηρήστε πως ανά περιόδους καλούμε :

```
dc_shell-xg-t -x "source /home/dzaharis/Desktop/STORE/scripts/XXX9.scr"
```

όπου XXX είναι κάποια από τις αρχιτεκτονικές που χρησιμοποιούμε. Τι κάνει όμως το script αρχείο αυτό στον DeCo;

```
analyze -library WORK -format vhdl {/home/dzaharis/Desktop/STORE/src/RPL/mul.vhd
/home/dzaharis/Desktop/STORE/src/RPL/addten.vhd
/home/dzaharis/Desktop/STORE/src/RPL/addnine.vhd
/home/dzaharis/Desktop/STORE/src/RPL/addeleven.vhd
/home/dzaharis/Desktop/STORE/src/RPL/addeight.vhd
/home/dzaharis/Desktop/STORE/src/RPL/Comparator.vhd
/home/dzaharis/Desktop/STORE/src/RPL/C1.vhd
/home/dzaharis/Desktop/STORE/src/RPL/C0.vhd
/home/dzaharis/Desktop/STORE/src/RPL/9modes.vhd
/home/dzaharis/Desktop/STORE/src/RPL/8modes.vhd
/home/dzaharis/Desktop/STORE/src/RPL/4modes.vhd
/home/dzaharis/Desktop/STORE/src/RPL/2modes.vhd}


elaborate MODES9_COMPARATOR -architecture STRUCTURAL -library WORK

set_max_area 40100

set_max_delay 6 -from [all_inputs] -to [all_outputs]
compile -map_effort medium -area_effort medium -ungroup_all

report_design > ./OUT/zero/ReportDesign.txt
report_hierarchy > ./OUT/zero/DesignHierarchy.txt
report_resources > ./OUT/zero/DesignResources.txt
report_constraint -significant_digits 2 >> ./OUT/zero/Constraints.txt
report_reference -nosplit > ./OUT/zero/Reference.txt
report_port > ./OUT/zero/Ports.txt
report_port > ./OUT/zero/Ports.txt
report_cell > ./OUT/zero/Cells.txt
report_net -connections -transition_times -min -cell_degradation -max_toggle_rate >
./OUT/zero/Nets.txt
report_clock -nosplit > ./OUT/zero/Clocks.txt
report_area -nosplit > ./OUT/zero/Area.txt
report_compile_options -nosplit > ./OUT/zero/CompileOptions.txt
report_power -net -cell -cumulative -analysis_effort medium -sort_mode dynamic_power -
histogram > ./OUT/zero/PowerEffort.txt
report_timing > ./OUT/zero/TimingPath.txt
report_timing_requirements > ./OUT/zero/TimingReqs.txt



write -hierarchy -format vhdl -output ./OUT/zero/9modesCLA.vhdl
write -hierarchy -format verilog -output ./OUT/zero/9modesCLA.v

remove_design -designs

exit
```

Αναλυτικά η επεξήγηση του τι κάνουμε παραπάνω:

1. *analyze* : αναλύει τα HDL αρχεία και αποθηκεύει την ενδιάμεσα μορφή στην υποδεικνυόμενη βιβλιοθήκη. Σε αντιπαράθεση η read_file που εκτελεί την ίδια πράξη δίχως να αποθηκεύει ενδιάμεσα αποτελέσματα και απαιτώντας να γίνει link με τις αναφορές

σχεδιάσεων στη βιβλιοθήκη. Συνεπώς προτιμούμε την analyze αν και δεν επεξεργαζόμαστε τις ενδιάμεσες μορφές.

2. *elaborate* : χτίζει μία σχεδίαση από την ενδιάμεση μορφή ενός Verilog module, μίας VHDL entity και architecture, ή ενός VHDL configuration.

3. *define_design_lib*: μπορούμε να τη χρησιμοποιήσουμε σε συνδυασμό με την read_file για να ορίσουμε όνομα βιβλιοθήκης διαφορετικό της WORK. Δε χρησιμοποιήθηκε.

4. *list_files* : παραθέτει τα αρχεία που φορτώνονται στο dc_shell. Για χρήση σε TCL dc_shell μόνο.

5. *uniquify* : αφαιρεί την πολλαπλά instantiated ιεραρχία στην παρούσα σχεδίαση δημιουργώντας μοναδική σχεδίαση για κάθε στιγμιότυπο κελιού. Δε χρησιμοποιήθηκε.

6. *derive_timing_constraints:* συμπεραίνει περιορισμούς χρονισμού και τοποθετεί την πληροφορία αυτή στην παρούσα σχεδίαση. Χρησιμοποιήθηκε σε προηγούμενο επεισόδιο.

7. *set_max_area*: θέτει την παράμετρο max_area σε μία συγκεκριμένη τιμή στην παρούσα σχεδίαση

8. *set_max_delay*: θέτει μέγιστη καθυστέρηση στα μονοπάτια που καθορίζουμε στην παρούσα σχεδίαση

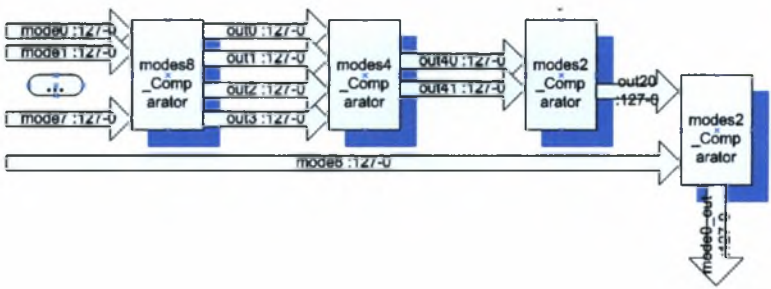
9. *compile* :Εκτελεί λογικού επιπέδου και Gale σύνθεση και βελτιστοποίηση στην παρούσα σχεδίαση

10. *write_script*:Γράφει dc_shell εντολές ώστε να σώσει τις παρούσες ρυθμίσεις.

Κατόπιν έχουμε την αναφορά όλων των reports τα οποία είναι ευνόητα. Το ίδιο ισχύει και για την εντολή write.

Μετρήσεις

Παραθέτουμε τα αποτελέσματα των μετρήσεων ανά κομμάτι σύμφωνα με το παρακάτω κύκλωμα:



Ο χρονισμός εκφράζει το κρίσιμο μονοπάτι σε nanosecond και το εμβαδό σε αριθμό πυλών. Στις σχεδιάσεις των υποκυκλωμάτων modes8Comparator, modes4Comparator και modes2Comparator δε μετρήσαμε όλα τα κυκλώματα μιας και απλά θέλαμε να αποκτήσουμε μία ενδεικτική εικόνα των μεγεθών.

Στα σχήματα που ακολουθούν παρατίθενται τα μεγέθη σε αριθμό πυλών ανά αρχιτεκτονική και τα μέγιστα μονοπάτια σε nanosecond. Τα κυκλώματα αντιστοιχούν φυσικά στο παραπάνω σχήμα.

Υπενθυμίζουμε πως:

(Maria) η προσαρμοσμένη σχεδίαση της πρότασης των Κοζύρη, Σταμούλη και Κατσαβουνίδη

(Ofni) η υλοποίηση της ίδιας αρχιτεκτονικής χρησιμοποιώντας για την πράξη της σύγκρισης στο τελευταίο επίπεδο έναν Carry Lookahead αθροιστή, γραμμένο από σχεδιαστή

(Auto) Αυτόματη επιλογή από τον DeCo της αρχιτεκτονικής αθροιστή ως υποκύκλωμα του ΑΑΔ.

(BK) Brent-Kung αρχιτεκτονική

(CLA) Carry-lookahead αρχιτεκτονική

(CLF) Fast Carry lookahead αρχιτεκτονική

(CSM) Conditional sum αρχιτεκτονική

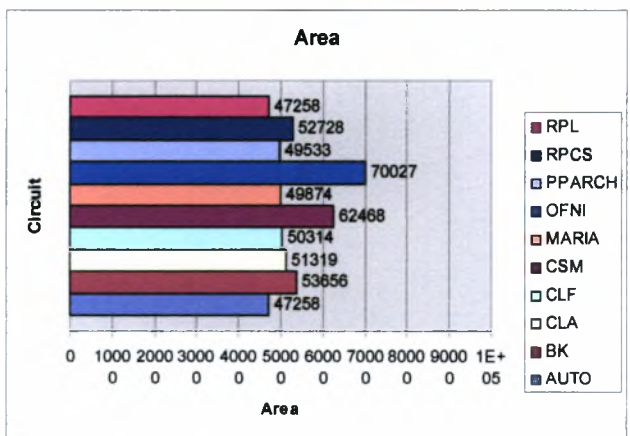
(PPARCH) Parallel-Prefix αρχιτεκτονική

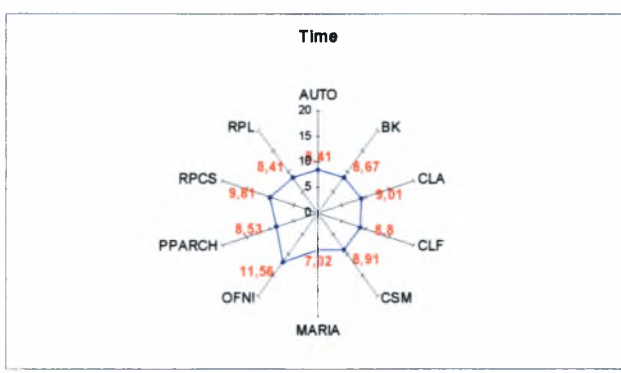
(RPCS) Ripple-Carry select αρχιτεκτονική

(RPL) Ripple-Carry αρχιτεκτονική

Modes9Comparator

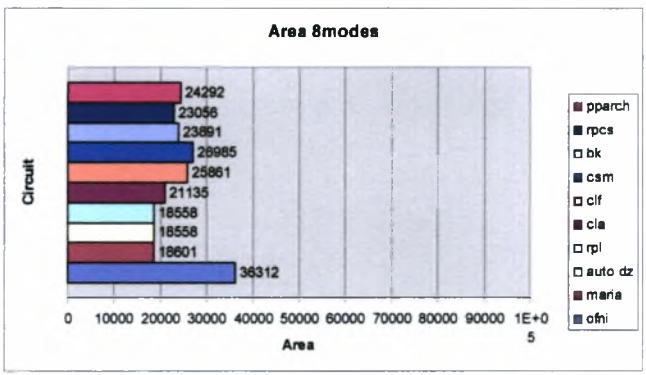
Design	Area	Time
auto	47258	8,41
bk	53656	8,67
cla	51319	9,01
clf	50314	8,8
csm	62468	8,91
maria	49874	7,02
ofni	70027	11,56
pparch	49533	8,53
rpcs	52728	9,81
rpl	47258	8,41

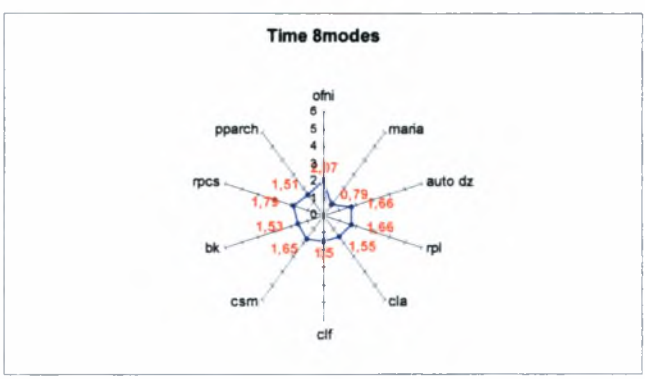




Modes8Comparator

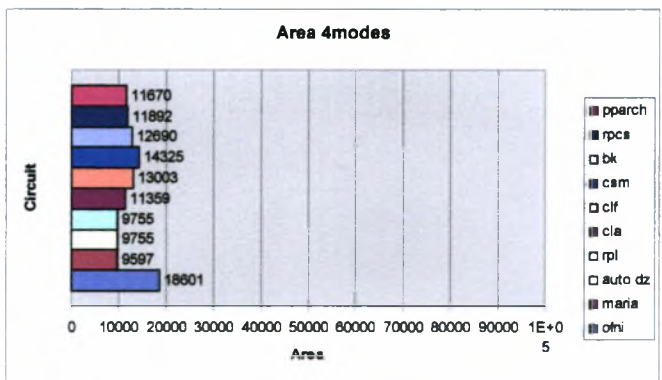
Design	Area	Time
ofni	36312	2,07
maria	18601	0,79
auto	18558	1,66
rpl	18558	1,66
cla	21135	1,55
clf	25861	1,5
csm	26985	1,65
bk	23891	1,53
rpcs	23056	1,79
pparch	24292	1,51

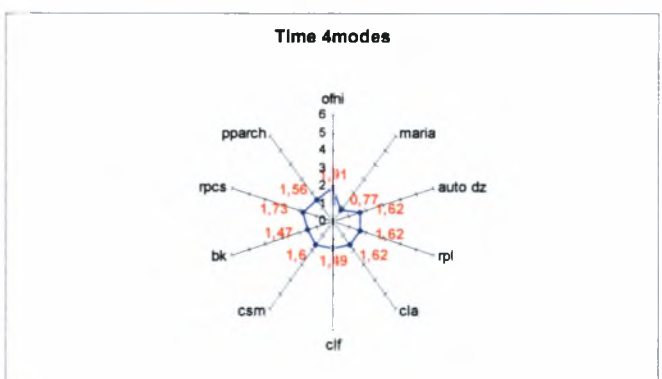




Modes4Comparator

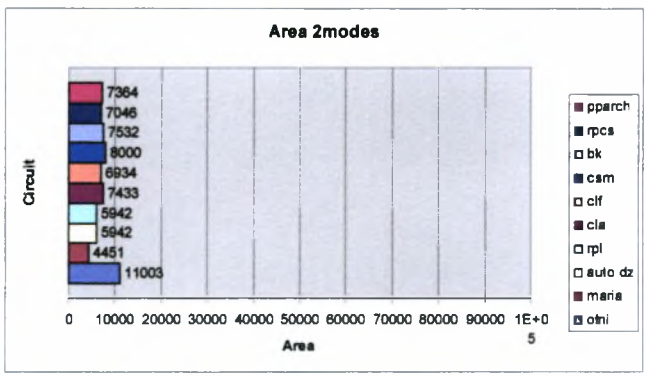
Design	Area	Time
ofni	18601	1,91
maria	9597	0,77
auto	9755	1,62
rpl	9755	1,62
cla	11359	1,62
clf	13003	1,49
csm	14325	1,6
bk	12690	1,47
rpcs	11892	1,73
pparch	11670	1,56

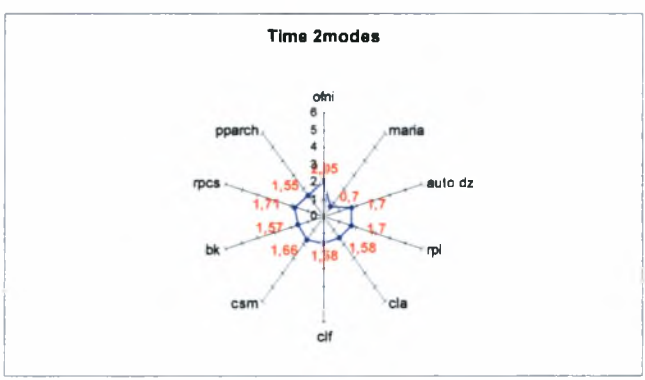




Modes2Comparator

Design	Area	Time
ofni	11002,29883	2,05
maria	4450,571777	0,7
auto	5941,665039	1,7
rpl	5941,665039	1,7
cla	7432,000977	1,58
clf	6933,204102	1,58
csm	7999,67627	1,66
bk	7531,154785	1,57
rpcs	7045,225098	1,71
pparch	7363,123047	1,55





Συμπεράσματα

Παρατηρώντας τα προηγούμενα αποτελέσματα είναι ξεκάθαρο πως υπάρχουν κάποια - κατά πρώτη προσέγγιση - παράδοξα.

Αυτά είναι τα εξής:

- Σύμφωνα με τη θεωρία των αθροιστικών σχημάτων που έχουμε μελετήσει σε προηγούμενο κεφάλαιο, οι συνδεσμολογίες conditional-sum και carry-select θα έπρεπε να είναι αρκετά πιο γρήγορες, σχεδόν από όλες τις συνδεσμολογίες.
- Πέρα από το προηγούμενο παράδοξο ενώ στα υποκυκλώματα modes(8,4,2)Comparator, εμφανίζονται τα αναμενόμενα, στην παράθεση των υποκυκλωμάτων κατά την προσχεδιασμένη αρχιτεκτονική έχουμε την ripple-carry σχεδίαση να είναι γρηγορότερη όλων πράγμα που δε θα έπρεπε να συμβαίνει.

Οι απαντήσεις όμως είναι άμεσες:

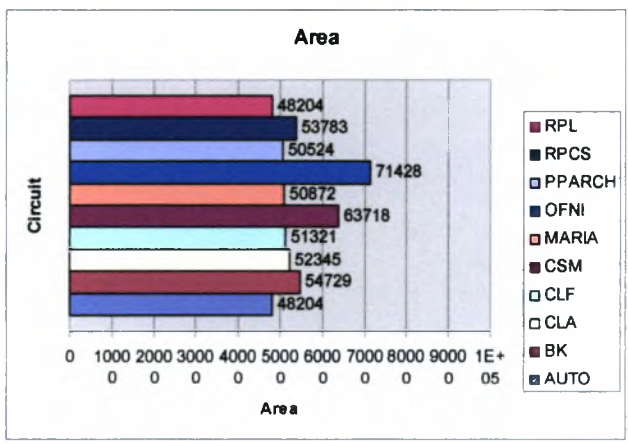
- Οι πολυπλέκτες τους οποίους χρησιμοποιούν οι συνδεσμολογίες carry-select και conditional sum αντιστοιχίζονται κατά τη ροή σχεδίασης και τη σύνθεση στη φυσική βιβλιοθήκη. Η βιβλιοθήκη μας έχει πολύ αργούς πολυπλέκτες γεγονός που δεν επιτρέπει στις αρχιτεκτονικές αυτές να αποδείξουν την αξία τους.
- Η ripple carry αρχιτεκτονική εμφανίζεται λογικά πιο γρήγορη, λόγω της μεθόδου βελτιστοποίησης την οποία ακολουθούμε. Όταν συνενώνουμε τα επιμέρους υποκυκλώματα στο modes9Comparator κάνουμε ungroup_all. Αυτό έχει ως αποτέλεσμα να μην αφήνουμε το εργαλείο σύνθεσης να εκτελέσει αναδιάταξη των εσωτερικών pins βελτιστοποιώντας τα κρίσιμα μονοπάτια. Γιατί όμως αυτό εμφαίνει τόσο πολύ μόνο στη ripple-carry σχεδίαση; Προφανώς και όταν συνενώνονται τα υποκυκλώματα εμείς έχουμε διατάξει τις εξόδους τους να πηγαίνουν αντίστοιχα στις εισόδους του επόμενου βήματος. Κι ενώ οι έξοδοι όλων των υπολοίπων

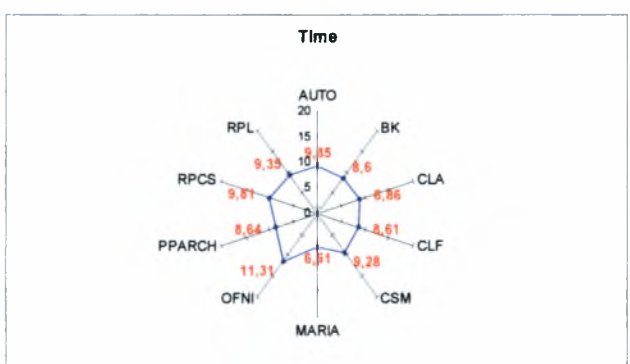
συνδεσμολογιών όπως είδαμε προηγουμένως έρχονται με ομοιόμορφες καθυστερήσεις πάνω κάτω, αυτές της ripple-carry φέρουν μεγαλύτερες καθυστερήσεις στα μεγαλύτερης τάξης bits. Όμως! Αυτό φαίνεται λόγω του φαινομένου του ορθού Pipelining και τελικά βγαίνει το ελάχιστο δυνατό μονοπάτι όπως θα έπρεπε. Το ίδιο δε συμβαίνει σε άλλες συνδεσμολογίες που λειτουργούν με ισοσκελισμένες ανά ομάδες αθροίσεις, ή με τους υπόλοιπους τρόπους επιτάχυνσης άθροισης. Εκεί το αργότερο μονοπάτι συνδέεται με ένα άλλο αργότερο, κ.ο.κ με αποτέλεσμα το συνολικά αργότερο μονοπάτι να είναι πιο αργό και από τη ripple-carry! Παρατηρώντας το γεγονός αυτό εκτελούμε ξανά σύνθεση στο υψηλό επίπεδο nodes9Comparator χρησιμοποιώντας τα ungrouped αποτελέσματα, αλλά χρησιμοποιώντας bottom-up στρατηγική:

Modes9Comparator

second pass

Design	Area	Time
auto	48204	9,35
bk	54729	8,6
cla	52345	8,86
clf	51321	8,61
csm	63718	9,28
maria	50872	6,51
ofni	71428	11,31
pparch	50524	8,64
rpcs	53783	9,81
rpl	48204	9,35





Τα οποία είναι και τα αναμενόμενα αποτελέσματα κατά το θεωρητικό υπόβαθρο.(Πέραν τούτου μία άλλη υπόθεση που θα μπορούσαμε να κάνουμε είναι πως έχουμε τοποθετήσει πολύ χαμηλό constraint παραβλέποντας την έξοδο του derive_timing_constraint και ο DeCo προσπαθώντας να ικανοποιήσει τους περιορισμούς πρόσθεσε επιπλέον buffering και συνδυαστικά στοιχεία πολύ περισσότερο στις μεγαλύτερες σχεδιάσεις μιας και αυτές παραβαίνουν τον περιορισμό του εμβαδού πολύ περισσότερο από την ripple carry. Προσέχουμε λοιπόν να μη βάζουμε υπερβολικά constraints, και ακολουθούμε κατά γράμμα την derive_timing_constraint).

Έχοντας μελετήσει μία σχεδίαση κυκλώματος σε επίπεδο πυλών, ένα σύνολο αυτοματοποιημένων σχεδιάσεων και μία σχεδίαση σε επίπεδο RTL καταλήξαμε :

- Η αυτοματοποιημένη σχεδίαση είναι η μικρότερη σε εμβαδό. Είναι κατά 5% μικρότερη από τη σχεδίαση σε επίπεδο πυλών και κατά 33% μικρότερη από τη σχεδίαση σε επίπεδο RTL. Είναι όμως κατά 25% πιο αργή από την custom GaLe σχεδίαση και κατά 24% πιο γρήγορη από την custom RTL. Όλα αυτά κατόπιν σύνθεσης και βελτιστοποίησης με μία σχετικά «κακή» βιβλιοθήκη σε ό,τι αφορά στην ύπαρξη γρήγορων σύνθετων πυλών. Η αιτία είναι η βελτιστοποιημένη δομή που έχουν τα ενσωματωμένα στα συγκεκριμένα εργαλεία CAD templates.

- Ανάμεσα στις αρχιτεκτονικές αθροιστών που εφαρμόστηκαν σε τμήματα της σχεδίασης, λόγω του συγκεκριμένου αριθμού εισόδων (8, 9, 10, 11) επικρατούν κατά τμήματα οι σχεδιάσεις Fast Carry Lookahead, Brent Kung όπως αναμενόταν σύμφωνα με τη μελέτη των αθροιστών αλλά και την παρατιθέμενη βιβλιογραφία. Επίσης παρατηρούμε τα χαρακτηριστικά της αφαιρετικά περιγραφόμενης ως parallel-prefix architecture (PPARCH) να ταιριάζουν σχεδόν απόλυτα με την αρχιτεκτονική Sklansky. Η σύγκριση του χρονισμού και του εμβαδού ταιριάζουν

αποτελέσματα του unit-gate μοντέλου από το κεφάλαιο των αθροιστών και την αρχιτεκτονική Brent-Kung.

Η συγγραφή των κυκλωμάτων με VHDL για σύνθεση είναι μία αρκετά επίπονη και εξειδικευμένη διαδικασία η οποία μπορεί να αποφευχθεί χρησιμοποιώντας templates όπως το DBBIP. Παρόλα αυτά μία προσαρμοσμένη σχεδίαση σε χαμηλότερο επίπεδο έχει μεγαλύτερη απόδοση κυρίως στο χρονισμό. Συνεπώς μπορούμε να βελτιώσουμε μεγάλο τμήμα της σχεδίασης από κακή διαχείριση πόρων χρησιμοποιώντας templates, και να βελτιστοποιήσουμε το χρονισμό χρησιμοποιώντας προσαρμοσμένη σχεδίαση στο κρίσιμο μονοπάτι, ενώ δεν έχουμε ξοδέψει τόσο χρόνο γράφοντας κώδικα για τμήματα του κυκλώματος που δεν είναι κρίσιμα.

Σε ό,τι αφορά το βίντεο και το H.264, σε μεγαλύτερες σχεδιάσεις σε υλικό, όπως αυτή ενός αποκωδικοποιητή, θα απαιτηθεί να διακρίνουμε το κρίσιμο μονοπάτι και να εφαρμόσουμε τις αρχές που συμπεράναμε στην εργασία μας. Εκτός από το Άθροισμα Απολύτων Διαφορών υπάρχουν και άλλα σημαντικά και χρονοβόρα κομμάτια, όπως η Αριθμητική κωδικοποίηση.

Επόμενο βήμα μας, λοιπόν, είναι η μελέτη τοποθέτησης και δρομολόγησης τέτοιων σχεδιάσεων και η βελτιστοποίηση κατά το κρίσιμο μονοπάτι χρησιμοποιώντας αυτόματη σχεδίαση, η μέτρηση κατανάλωσης σε επίπεδο τρανζίστορ και η φυσική σχεδίαση. Ίσως μάλιστα και η δημιουργία καλύτερων βιβλιοθηκών για χρήση σε χαμηλότερο επίπεδο, σε συνδυασμό με το υψηλό ώστε να καταλήξουμε σε ολοκληρωμένα αυτοματοποιημένη σχεδίαση και να απαιτείται να προσαρμόσουμε μόνο ορισμένα τμήματα του critical path και της κατανάλωσης ισχύος.

Θεμελιώδεις έννοιες

Μοντελοποίηση Εμβαδού και Χρονισμού

Μοντελοποίηση εμβαδού

(θεωρώντας το σύμβολο της αναλογίας το ∓)

- Η συνολική *πολυπλοκότητα κυκλώματος* (GE_{total}) μπορεί να μετρηθεί με τον αριθμό των *ισοδύναμων πύλης*(1 GE ≡ 1 πύλη NAND 2 εισόδων ≡ 4 MOSFET)

- Το *Εμβαδό κυκλώματος*($A_{circuit}$) καταλαμβάνεται από τα λογικά κελιά και τις διασυνδέσεις. Σε τεχνολογίες με τρία και περισσότερα μέταλλα, η δρομολόγηση over-the-cell επιτρέπει την επικάλυψη κελιών και καλωδιώσεων, αντίθετα με τις τεχνολογίες 2 μετάλλων. Αυτό σημαίνει πως το μεγαλύτερο μέρος του εμβαδού των κελιών μπορεί να χρησιμοποιηθεί για καλωδίωση, με αποτέλεσμα πολύ χαμηλούς παράγοντες καλωδίωσης. ($A_{circuit} = A_{cells} + A_{wiring}$)

- Το συνολικό *εμβαδό κελιού* (A_{cells}) είναι σχεδόν ανάλογο του αριθμού των τρανζίστορ ή των ισοδύναμων πύλης (GE_{total}) που περιέχονται στο κύκλωμα. Αυτός ο αριθμός εξαρτάται από την τεχνολογία απεικόνισης, αλλά όχι από το φυσικό layout. Άρα, το εμβαδό κελιού μπορεί να υπολογιστεί χοντρικά από μία γενική περιγραφή κυκλώματος. ($A_{cells} \mp GE_{total}$)

- Το *εμβαδό καλωδίωσης* είναι ανάλογο του συνολικού μήκους καλωδίωσης. Το ακριβές μήκος καλωδίωσης, όμως, δεν είναι γνωστό πριν το φυσικό layout. ($A_{wiring} \mp L_{total}$)

- Το συνολικό *μήκος καλωδίωσης* (L_{total}) μπορεί να εκτιμηθεί από των αριθμών των κόμβων και το μέσο μήκος καλωδίου σε έναν

κόμβο, ή ακριβέστερα, από το άθροισμα του fan-out κελιών και το μέσο μήκος συνδέσεων κελιού προς κελί. Το μήκος καλωδίωσης εξαρτάται από το μέγεθος του κυκλώματος, τη συνδεσιμότητα του γράφου του κυκλώματος κ.ά., που δεν είναι γνωστά πριν την τοποθέτηση και δρομολόγηση.

- Το *fan-out* κελιού είναι ο αριθμός των εισόδων κελιών που οδηγεί ένα κελί, που για πολλά συνδυαστικά κυκλώματα είναι ανάλογο του μεγέθους του κελιού. Από τη στιγμή που το άθροισμα των fan-out κελιών ενός κυκλώματος είναι ίσο με το άθροισμα των fan-in των κελιών, είναι επίσης ανάλογο του μεγέθους του κυκλώματος
- Συνεπώς σε πρώτη προσέγγιση, το εμβαδό κελιού και το εμβαδό καλωδίωσης είναι ανάλογα των ισοδύναμων πύλης. Περισσότερο ακριβείς εκτιμήσεις εμβαδού πριν εκτελέσουμε απεικόνιση σε τεχνολογία (technology mapping) και τοποθέτηση, είναι εξαιρετικά δύσκολες. Για σύγκριση κυκλωμάτων, ο παράγοντας αναλογίας μας είναι αδιάφορος ($A_{circuit} \mp GE_{total} \mp FO_{total}$)

Το μοντέλο εκτίμησης εμβαδού που θα υιοθετήσουμε πρέπει να είναι απλό στον υπολογισμό και ταυτόχρονα όσο ακριβές γίνεται και πρέπει να εφαρμόζεται είτε σε λογικές εξισώσεις είτε σε γενικευμένες συνδεσμολογίες από μόνες τους. Πιθανά υποψήφια είναι:

Μοντέλο εμβαδού unit-gate

Είναι το απλούστερο και πιο αφηρημένο μοντέλο που συναντάμε στη βιβλιογραφία. Μία μονάδα πύλης (unit gate) είναι μία βασική πύλη 2 εισόδων (ή λογική πράξη, αν αναφερόμαστε σε αυτές) όπως AND, OR, NAND και NOR. Οι βασικές πύλες XOR και XNOR μετράνε για δύο πύλες, αντικατοπτρίζοντας την υψηλότερη κυκλωματική πολυπλοκότητα που εμφανίζουν. Οι σύνθετες πύλες όπως και οι πολλών εισόδων χτίζονται από πύλες 2 εισόδων και ο αριθμός των πυλών τους ισούται με τον αριθμό πυλών των επιμέρους κελιών.

Μοντέλο εμβαδού fan-in

Στο μοντέλο αυτό, το μέγεθος των 2 και περισσοτέρων βασικών κελιών μετριέται από τον αριθμό των εισόδων(fan-in). Οι σύνθετες πύλες συντίθενται από βασικά κελιά με τους αριθμούς fan-in να αθροίζονται, ενώ οι πύλες XOR και XNOR να έχουν ειδική μεταχείριση. Τα αποτελέσματα διαφέρουν από αυτά του Unit-gate μοντέλου μόνο κατά μία σταθερά 1(για παράδειγμα η πύλη AND μετρά ως 1 στο unit-gate ενώ έχει fan-in 2)

Άλλα μοντέλα

Για βελτιστοποίηση επιπέδου τρανζίστορ σε πολύπλοκες πύλες τα προηγούμενα μοντέλα δε μας βοηθούν. Τέτοια είναι οι πλήρεις αθροιστές και οι πολυπλέκτες. Παρόλα αυτά κάποια θυσία αφαίρεσης έχει ως αποτέλεσμα να μην έχουμε εφαρμογή σε οποιαδήποτε λογική συνάρτηση. Το ίδιο ισχύει και για μοντέλα που παίρνουν υπόψη τους και την καλωδίωση. Ένα παράδειγμα περισσότερο ακριβούς μοντέλου είναι το gate-equivalent (GE ισοδύναμο πυλών), που βασίζεται στον αριθμό των τρανζίστορ πυλών και άρα είναι εφαρμόσιμο μόνο μετά τη σύνθεση και την τεχνολογική αντιστοίχηση.

Οι αντιστροφείς και οι απομονωτές δε συγκαταλέγονται στα προηγούμενα μοντέλα εμβαδού, το οποίο έχει νόημα για κυκλωματικές περιγραφές πριν τη σύνθεση. Σημειώστε πως οι μεγαλύτερες διαφορές στα κόστη buffering(απομόνωσης) βρίσκονται ανάμεσα στα χαμηλού και υψηλού fan-out. Σε ό,τι αφορά την κατάληψη εμβαδού όμως, τα θέματα αυτά αντισταθμίζονται μερικώς από το εξής γεγονός: τα υψηλού fan-out κυκλώματα χρειάζονται επιπρόσθετο buffering ενώ τα χαμηλού περισσότερη καλωδίωση.

Έρευνες έδειξαν πως η unit-gate προσέγγιση για την εκτίμηση εμβαδού πολύπλοκων πυλών όπως πολυπλέκτες και πλήρεις αθροιστές, δεν εισάγουν περισσότερες ανακρίβειες από την παράλειψη της συνδεσιμότητας του κυκλώματος για την εκτίμηση της καλωδίωσης. Με τις XOR και XNOR να

τυγχάνουν ειδικής αντιμετώπισης, το unit-gate μοντέλο καταφέρει αποδεκτή ακρίβεια στο δεδομένο επίπεδο αφαίρεσης. Επίσης αντικατοπτρίζει τέλεια τη δομή των λογικών εξισώσεων μοντελοποιώντας τους λογικούς τελεστές ξεχωριστά και θεωρώντας πως οι πολύπλοκες συναρτήσεις απαρτίζονται από απλούστερες. Παρόμοια απόδοση με το unit-gate έχει και το μοντέλο fan-in λόγω της ομοιότητας τους. Όταν έχουμε να κάνουμε με εκτιμήσεις και συγκρίσεις εμβαδού από προδιαγραφές λογικών κυκλωμάτων στη βιβλιογραφία συναντούμε συνήθως το unit-gate.

Λόγω των παραπάνω χρησιμοποιούμε το unit-gate μοντέλο.

Μοντελοποίηση καθυστέρησης

Η καθυστέρηση διάδοσης καθορίζεται από τις καθυστερήσεις εντός των κελιών και αυτές των διασυνδέσεων στο κρίσιμο μονοπάτι(μακρύτερο μονοπάτι διάδοσης σήματος σε ένα συνδυαστικό κύκλωμα). Αντίθετα με την εκτίμηση εμβαδού, μέσα και συνολικά νούμερα δε μας ενδιαφέρουν, αλλά ξεχωριστά ανά κελί και ανά κόμβο για την καθυστέρηση μονοπατιού, η εκτίμηση του κρίσιμου μονοπατιού γίνεται από στατική ανάλυση χρονισμού που γίνεται με χρήση αλγορίθμων αναζήτησης σε γράφους. Ο χρονισμός, βέβαια, εξαρτάται και από τη θερμοκρασία, την τάση και τις παραμέτρους του process της τεχνολογίας, γεγονός όμως που δε μας αφορά στο παρόν.

- *Μέγιστη καθυστέρηση* (t_{crit_path}) ενός κυκλώματος είναι το άθροισμα των εσωτερικών καθυστερήσεων ενός κελιού, των καθυστερήσεων ράμπας εξόδου του κελιού και των καθυστερήσεων καλωδιώσεων στο κρίσιμο μονοπάτι.

$$(t_{crit\_path} = \Sigma_{ecrit\_path}((t_{cell} + t_{ramp}) + \Sigma_{ecrit\_path} t_{wire}))$$

- Η *καθυστέρηση κελιού* (t_{cell}) εξαρτάται από την υλοποίηση του κυκλώματος σε επίπεδο τρανζίστορ και την πολυπλοκότητα ενός κελιού. Όλες οι απλές πύλες έχουν συγκρίσιμες καθυστερήσεις. Οι πολύπλοκες πύλες συνήθως περιέχουν κύκλωμα και διατάξεις τρανζίστορ σε μορφή δέντρου, που έχουν ως αποτέλεσμα

λογαριθμικές εξαρτήσεις καθυστερήσεις προς εμβαδό. ($t_{cell} \propto \log(A_{cell})$)

- Η *καθυστέρηση ράμπας* (t_{ramp}) είναι ο χρόνος που παίρνει σε μία έξοδο κελιού να οδηγήσει την προσκολλημένη σε αυτήν χωρητικότητα που αποτελείται από φορτία κελιού και διασύνδεσης. Η καθυστέρηση ράμπας εξαρτάται γραμμικά από το προσκολλημένο χωρητικό φορτίο, που με τη σειρά του εξαρτάται γραμμικά από το fan-out του κελιού.($t_{ramp} \propto FO_{cell}$).
- Η *καθυστέρηση καλωδίωσης* ή *καθυστέρηση διασύνδεσης* (t_{wire})είναι η καθυστέρηση RC ενός καλωδίου, που εξαρτάται από το μήκος του. Οι καθυστερήσεις RC, όμως, είναι αμελητέες σε σύγκριση με τις καθυστερήσεις κελιών και ράμπας για μικρά κυκλώματα όπως οι αθροιστές που μελετούμε εδώ. ($t_{wire} = 0$)
- Άρα, μία χοντρική εκτίμηση καθυστέρησης γίνεται δυνατή θεωρώντας μεγέθη και, με έναν μικρότερο παράγοντα βάρους, το fan-out των κελιών στο κρίσιμο μονοπάτι.

Πιθανά μοντέλα εκτίμησης καθυστέρησης είναι:

Μοντέλο καθυστέρησης Unit-gate

Είναι παρόμοιο με το unit-gate μοντέλο εμβαδού. Ξανά, οι βασικές πύλες 2 εισόδων (AND, OR, NAND, NOR) μετρούν ως καθυστέρηση μίας πύλης με την εξαίρεση των XOR, XNOR πυλών που μετρούν ως δύο. Οι σύνθετες πύλες αποτελούνται από βασικά κελιά χρησιμοποιώντας την ταχύτερη δυνατή διάταξη με τη συνολική καθυστέρηση σε πύλες να καθορίζεται αντίστοιχα.

Μοντέλο καθυστέρησης fan-in

Όπως και στη μοντελοποίηση εμβαδού, μπορούμε να βγάλουμε fan-in αριθμούς αντί για unit-gate αριθμούς. Ξανά, δεν παρατηρούνται πλεονεκτήματα έναντι του unit-gate μοντέλου.

Μοντέλο καθυστέρησης fan-out

Το μοντέλο καθυστέρησης fan-out βασίζεται στο unit-gate μοντέλο αλλά ενσωματώνει fan-out αριθμούς, υπολογίζοντας έτσι αριθμούς fan-out πυλών και καθυστερήσεις διασυνδέσεων. Ξεχωριστά μπορούμε να λάβουμε τους fan-out αριθμούς από την κυκλωματική περιγραφή. Ένας παράγοντας αναλογίας πρέπει να καθοριστεί για την κατάλληλη βαρύτητα των fan-out σε ό,τι αφορά τους αριθμούς του μοντέλου unit-gate.

Άλλα μοντέλα καθυστέρησης

Διάφορα μοντέλα καθυστέρησης υπάρχουν σε άλλα επίπεδα αφαίρεσης. Στο επίπεδο τρανζίστορ, τα τρανζίστορ μπορούν να μοντελοποιηθούν ώστε να συνεισφέρουν το καθένα μία μονάδα καθυστέρησης (τ-μοντέλο). Σε υψηλότερο επίπεδο, οι σύνθετες πύλες όπως οι πλήρεις αθροιστές και οι πολυπλέκτες μπορούν ξανά να μοντελοποιηθούν ξεχωριστά για μεγαλύτερη ακρίβεια.

Χρησιμοποιούμε το unit-gate μοντέλο και εδώ για τους ίδιους λόγους όπως και παραπάνω αλλά και κατά αντιστοιχία.

Βιβλιογραφία

[1] "Design Compiler Reference Manual"
SYNOPSYS
[2] "Binary Adder Architectures for Cell-Based VLSI and their Synthesis",
RETO ZIMMERMANN
[3] "ASIC Design with SYNOPSYS"
Himanshu Bhatnagar
[4] "Design Compiler User Guide"
SYNOPSYS
[5] "HDL Compiler for VHDL Reference Manual",
SYNOPSYS
[6] "Power Reduction in an H.264 Encoder ThroughAlgorithmic and Logic
Transformations",
Maria G. Koziri, George I. Stamoulis, Ioannis X. Katsavounidis
[7] "DesignWare Building Block IP User Guide",
SYNOPSYS
[8] "Ανάλυση καθυστέρησης και κατανάλωσης ισχύος εναλλακτικών διατάξεων αθροιστών",
Δαδαλιάρης Αντώνης
[9] "Ανάλυση και βελτιστοποίηση κατανάλωσης ισχύος συνδυαστικών και ακολουθιακών ψηφιακών
κυκλωμάτων",
Καραμπατζάκης Δημήτριος