ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΝΙΚΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Εργαστήριο Μικροεπεξεργαστών και υλικού

Διπλωματική Εργασία

Υλοποίηση για Ολοκληρωμένο Κύκλωμα Ειδικού Σκοπού του Κρυπτογραφικού συν-επεξεργαστή Ccproc

Σίσκος Γεώργιος Αλέξανδρος

Επιβλέπων
Αναπλ. Καθ. Διονύσιος Πνευματικάτος

Εξεταστική Επιτροπή
Αναπλ. Καθ. Διονύσιος Πνευματικάτος
Καθ. Απόστολος Δόλλας
Επίκ. Καθ. Ιωάννης Παπαευσταθίου

Περίληψη

Ο Ccproc είναι ένας κρυπτογράφικός συνεπεξεργαστής VLIW (μεγάλου πλάτους εντολών) για συμμετρικούς αλγόριθμους κρυπτογράφησης. Ο Ccproc έχει το δικό του σύνολο εντολών μειωμένης πολυπλοκότητας, αφοσιωμένο στους συμμετρικούς αλγόριθμους κρυπτογράφησης, ικανό να υποστηρίξει πολλούς από τους σημερινούς συμμετρικούς αλγόριθμους κρυπτογράφησης.

Σε αυτή τη διπλωματική μεταφέραμε και αξιολογήσαμε τον Ccproc σε τεχνολογία Ολοκληρωμένων Κυκλωμάτων Ειδικού Σκοπού(ASIC). Για τη διαδικασία της σύνθεσης σε ASIC χρησιμοποιήσαμε τον Design Compiler του Synopsys. Χρησιμοποιήσαμε μία βιβλιοθήκη 0,13 μm(UMC013).

Η απόδοση του Ccproc καταφέραμε να φτάσει στα 250 Mhz. Ο συνολικός αριθμός κελιών που χρησιμοποιήθηκαν είναι 93.185 τα οποία καταλάμβαναν χώρο 5.343.404 μm^2 .

Περιεχόμενα

1. Εισαγωγή 5
1.1 ASIC 5
1.2 Ccproc 6
1.3 Επιλογή υλοποίηση σε ASIC 6
1.4 Δομή Διπλωματικής εργασίας 7
2. Υπόβαθρο και η Αρχιτεκτονική Ccproc 8
2.1 Κρυπτογραφία 8
2.2 Συμμετρικοί Αλγόριθμοι Κρυπτογράφησης 9
2.3 Υλοποίηση σε Υλικό των Αλγορίθμων και Επιταγχυντές 10
2.4 Η Αρχιτεκτονική του Ccproc 10
3. Σύνθεση 12
3.1 Synopsys 12
3.2 Βασική ροή σύνθεσης 13
4. Υλοποίηση 15
4.1 Ανάπτυξη Αρχείων VHDL για το Synopsys 15
4.1.2 Αλλαγή Μνημών 16
4.2 Προσδιορισμός Βιβλιοθηκών 18
4.3 Προσδιορισμός Περιορισμών 18
4.4 Επιλογή Στρατηγικής Μεταγλώττισης 18
4.5 Βελτιστοποίηση της Σχεδίασης 18
4.6 Επαλήθευση και Αποτελέσματα 19
4.6.1 Λειτουργική και Χρονική Προσομοίωση 19
4.6.2 Αποτελέσματα 20
5. Εμπειρίες και Συμπεράσματα 22
6. Αναφορές 23
Παράρτημα Α : Υπόβαθρο και η Αρχιτεκτονική Ccproc.....26
Παραρτημα Β : Σύνθεση.....46
Παράρτημα Γ : Υλοποίηση.....82

1. Εισαγωγή

Είναι συχνά απαραίτητο να προσαρμοστεί μια υπάρχουσα υλοποίηση σε διαφορετική τεχνολογία . Ο λόγος είναι ότι ψάχνουμε τις λύσεις [I3] που θα παράσχουν το γρηγορότερο χρόνο στην αγορά, το χαμηλότερο κόστος και την υψηλότερη απόδοση για τις διαδοχικές γενεές των προϊόντων. Στην περίπτωσή μας αποφασίσαμε να μετατρέψουμε μια υλοποίηση από FPGA σε ένα ολοκληρωμένο κύκλωμα (ASIC). Ένα ASIC είναι ένα ολοκληρωμένο κύκλωμα που έχει υλοποιηθεί για συγκεκριμένη λειτουργία (χρήση) [I4].

1.1 ASIC

Σήμερα, υπάρχουν δύο προσεγγίσεις σε σχεδίαση ASIC:

- Η Full-custom ASIC είναι μη πρακτική για τους περισσότερους χρήστες. Σε μια Full-custom ASIC ο μηχανικός σχεδιάζει μερικά ή όλα τα κύτταρα λογικής, κυκλώματα, συγκεκριμένα για ένα ASIC. Αυτό σημαίνει ότι ο σχεδιαστής εγκαταλείπει την προσέγγιση της χρησιμοποίησης έτοιμων βιβλιοθηκών από λογικά κύτταρα για το σύνολο ή μέρος της σχεδίασης. Έχει νόημα για να υιοθετήσει κανείς αυτή την μέθοδο μόνο εάν δεν υπάρχει καμία κατάλληλη υπάρχουσα βιβλιοθήκη κυττάρων διαθέσιμη που να μπορεί να χρησιμοποιηθεί για ολόκληρη τη σχεδίαση. Αυτή η προσέγγιση (Full-custom) χρησιμοποιείται όταν οι υπάρχουσες βιβλιοθήκες κυττάρων δεν είναι αρκετά αποδοτικές(σε ταχύτητα), ή τα κύτταρα λογικής δεν είναι αρκετά μικρά ή καταναλώνουν πάρα πολλή ενέργεια.

- Η Semi-custom ASIC χρησιμοποίει έτοιμες βιβλιοθήκες λογικών κυττάρων. Η Semi-custom ASIC σχεδίαση είναι πρακτικότερη για τους περισσότερους χρήστες επειδή τα εργαλεία σύνθεσης έχουν ωριμάσει και σε συνδυασμό με την ύπαρξη των γλωσσών περιγραφής υλικού παίρνουμε το αποτέλεσμα των προδιαγραφών μας.

1.2 Ccproc

Η σχεδίαση που θέλουμε να μετατρέψουμε σε ASIC είναι ο Ccproc. Ο CCproc είναι ένας VLIW συνεπεξεργαστής κρυπτογράφος για συμμετρικούς αλγόριθμους. Το datapath του έχει δομή RISC, ικανή για να υποστηρίξει πολλούς από τους σημερινούς αλγόριθμους κρυπτογράφησης, λειτουργώντας σε πολύ ανταγωνιστικές ταχύτητες. Ένας συνεπεξεργαστής είναι μία ειδικής χρήσης μονάδα επεξεργασίας που βοηθά τον κεντρικό επεξεργαστή στην εκτέλεση συγκεκριμένου τύπου διαδικασιών.

1.3 Επιλογή υλοποίηση σε ASIC

Οι FPGAs είναι δημοφιλείς για τις γρήγορες σε χρόνο υλοποίησης σχεδιάσεις με χαμηλό NRE (μη επαναλαμβανόμενες δαπάνες). Εντούτοις, υπάρχουν σημαντικές δαπάνες που συνδέονται με τη χρησιμοποίηση FPGA για μεγάλες σχεδιάσεις. Οι FPGAs πάσχουν από:

- υψηλό κόστος ανά-μονάδων
- χαμηλή απόδοση
- χαμηλά επίπεδα λογικής ολοκλήρωσης
- μεγάλη κατανάλωση ισχύος

Αντίθετα, η πλατφόρμα ASIC έχει:

- χαμηλότερο κόστος μονάδας
- υψηλότερη απόδοση και λογική ολοκλήρωσης
- χαμηλότερη κατανάλωση ισχύος
- και είναι μικρότερη κατασκευαστικά

Αυτοί είναι οι λόγοι που αποφασίσαμε να υλοποιήσουμε τον Ccproc σε ASIC που αρχικά είχε υλοποιηθεί σε FPGA.

1.4 Δομή Διπλωματικής εργασίας

Ο κύριος στόχος αυτής της διπλωματικής εργασίας είναι να υλοποιηθεί ο Ccproc σε ASIC προκειμένου να επιτευχθεί υψηλότερη απόδοση με λιγότερη κατανάλωση ισχύος και να ελαχιστοποιηθεί η περιοχή της σχεδίασης. Το υπόλοιπο αυτού του κειμένου οργανώνεται ως εξής:

• Υπόβαθρο και η αρχιτεκτονική Ccproc
• Διαδικασία σύνθεσης χρησιμοποιώντας το Synopsys
• Διαδικασία που ακολουθήσαμε για να ολοκληρώσουμε το στόχο μας
• Επαλήθευση και αποτελέσματα
• Εμπειρίες και συμπεράσματα

2. Υπόβαθρο και η Αρχιτεκτονική Ccproc

Σε αυτό το κεφάλαιο εστιάζουμε στην περιγραφή του Ccproc. Αρχικά θα δούμε μερικά πολύ σημαντικά πράγματα για την κρυπτογραφία και τους δημοφιλέστερους και ασφαλέστερους αλγόριθμους που σήμερα χρησιμοποιούνται. Κατόπιν εστιάζουμε στην περιγραφή της αρχιτεκτονικής του CCproc πριν αναφερθούμε στη διαδικασία της σύνθεσης με το εργαλείο Synopsys.

2.1 Κρυπτογραφία

Κρυπτογραφία [I3] είναι ένας τομέας των μαθηματικών ενδιαφερόμενος για την ασφάλεια πληροφοριών και τα σχετικά ζητήματα. Ο σκοπός της είναι να κρύψει την έννοια ενός μηνύματος και όχι την ύπαρξή του. Στις μέρες μας η κρυπτογραφία έχει αρχίσει να εφαρμόζεται και στην πληροφορική. Συστήματα κρυπτογράφησης χρησιμοποιούνται για την ασφάλεια υπολογιστών και δικτύων.

Ένας κρυπτογραφικός αλγόριθμος είναι μία μαθηματική λειτουργία που χρησιμοποιείται στη διαδικασία κρυπτογράφησης και αποκρυπτογράφησης. Ένας κρυπτογραφικός αλγόριθμος λειτουργεί σε σχέση με ένα κλειδί (μία λέξη, ένας αριθμός, ή μια φράση) για να κρυπτογραφήσει το αρχικό κείμενο. Το αρχικό κείμενο κρυπτογραφείται και δημιουργείται στη θέση του ένα κρυπτογράφημα. Η ασφάλεια των κρυπτογραφημένων στοιχείων εξαρτάται εξ' ολοκλήρου από δύο πράγματα: τη δύναμη του κρυπτογραφικού αλγορίθμου και τη μυστικότητα του κλειδιού.

Σήμερα οι αλγόριθμοι κρυπτογράφησης χωρίζονται σε δύο κατηγορίες [I2]:

- Συμμετρικούς κρυπτογραφικούς αλγόριθμους.
- Μη συμμετρικούς κρυπτογραφικούς αλγόριθμους.

2.2 Συμμετρικοί Αλγόριθμοι Κρυπτογράφησης

Για να καταλάβουμε γιατί ο Ccproc υποστηρίζει συγκεκριμένους συμμετρικούς αλγόριθμους σε αυτό το κεφάλαιο εξετάζονται τα γενικά χαρακτηριστικά των συμμετρικών αλγορίθμων. Συγκεκριμένη προσοχή δόθηκε στην επιλογή των αλγορίθμων που τέθηκαν υπό εξέταση, επειδή πολλοί από αυτούς έχουν αδυναμίες. Έτσι, προκειμένου να καταστεί η ανάλυσή μας όσο το δυνατόν πληρέστερη, επιλέχτηκαν οι ακόλουθοι αλγόριθμοι: Rijndael [7] MARS[8], Serpent [9], Twofish [10], Blowfish [12], RC4 [I5], DES [I6], RC5 [13], IDEA [14]. Από αυτή την ομάδα μετά από ανάλυση επιλέχτηκαν να υποστηριχτούν από τον Ccproc οι πέντε τελικοί υποψήφιοι του AES [I7].

Ο Δ.Θεοδορόπουλος [5] συμπεριέλαβε στον Ccproc τις ακόλουθες πράξεις που χρησιμοποιούνται από τους αλγόριθμους που επιλέχτηκαν:

1. Unsigned πρόσθεση και αφαίρεση modulo 2^{32}
2. Πολλαπλασιασμός modulo 2^{32}
3. Αποκλειστικό ή (xor) μεταξύ 32-bit δεδομένων
4. Σταθερές μετατοπίσεις και περιστροφές
5. Μετατοπίσεις και περιστροφές εξαρτημένων δεδομένων
6. Πολυωνυμικός πολλαπλασιασμός πεπερασμένου πεδίου σε 2^8 modulo
7. Επεκτάσεις και μεταλλαγές (Xboxes)
8. Κουτιά αντικατάστασης (Sboxes)
9. Δομές δικτύων Feistel [15].

2.3 Υλοποίηση σε Υλικό των Αλγορίθμων και Επιταγχυντές

Υπάρχουν τρία είδη υλοποιήσεων αλγορίθμων :

• C και εφαρμογές Assebly

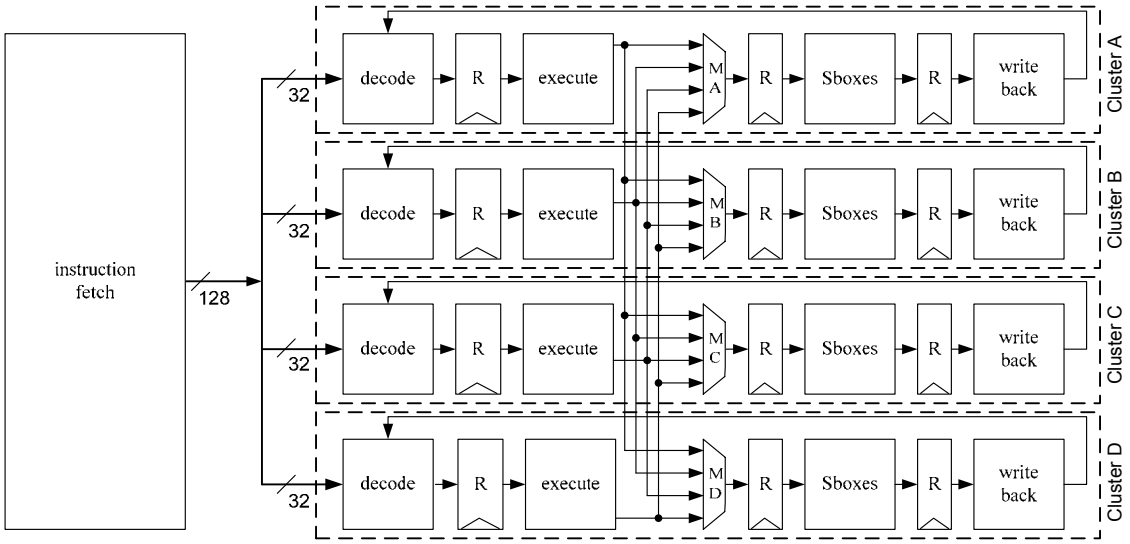
• Συγκεκριμένες εφαρμογές υλικού αλγορίθμου

• συνεπεξεργαστές υλικού με ISA που υποστηρίζει αλγόριθμους

Οι εφαρμογές υλικού παρέχουν την υψηλότερη ταχύτητα από τις εφαρμογές λογισμικού. Η συγκεκριμένη κατηγορία εφαρμογής υλικού αλγορίθμου παρέχει την υπερβολική απόδοση ταχύτητας για κάθε συμμετρικό αλγόριθμο, λόγω των αφιερωμένων επεξεργαστών υλικού. Η τελευταία κατηγορία, είναι κάπως «στη μέση» των προηγούμενων δύο και εστιάζει σε συγκεκριμένους συμμετρικούς αλγόριθμους που υλοποιούνται σε συνεπεξεργαστές. Αυτή η κατηγορία μπορεί να χαρακτηριστεί ως η πιο δύσκολη για να υλοποιηθεί, επειδή απαιτεί τη βαθιά παράλληλη ανάλυση πολλών συμμετρικών αλγορίθμων και επιπρόσθετης προσπάθειας, προκειμένου να επιτευχθεί η ισορροπία μεταξύ της απόδοσης και της ευελιξίας της σχεδίασης. Ο Ccproc ανήκει στην τελευταία κατηγορία και για να υλοποιηθεί έγινε βαθιά ανάλυση στην επιλογή επιταγχυντών για την τελική υλοποίηση [17][18][19][20].

2.4 Η Αρχιτεκτονική του Ccproc

Ο Ccproc είναι [6] κρυπτογράφικός συνεπεξεργαστής VLIW που είναι αρκετά ευέλικτος να υποστηρίξει όλους τους συμμετρικούς αλγόριθμους που έφτασαν στο τελικό του AES, αλλά έχει και τη δυνατότητα να υποστηρίξει και νέους αλγόριθμους. Πολλές φορές κατά τη διάρκεια της επεξεργασίας, συμμετρικών αλγορίθμων χρησιμοποιούνται 64-bit, 96-bit ή ακόμα και 128-bit δεδομένων που πρέπει να αναλυθούν παράλληλα. Αυτό οδήγησε στην απόφαση να υλοποιηθεί ένας συνεπεξεργαστής VLIW με τέσσερις συστάδες, όπως φαίνεται στο σχήμα 1.



σχήμα 1 - Ccproc

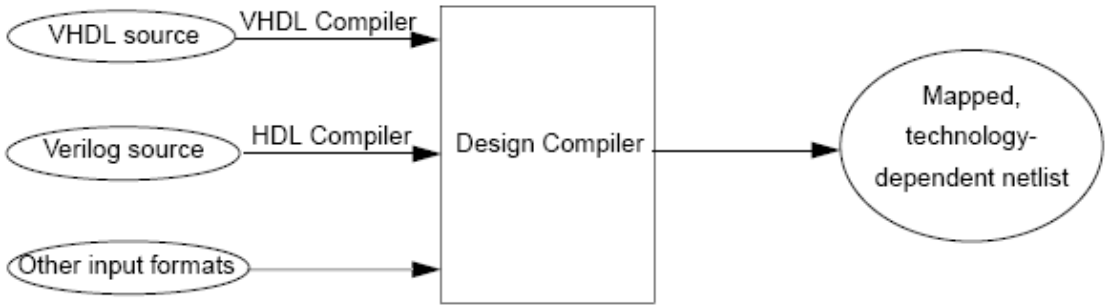
Καταρχήν, υπάρχει μια μονάδα ευρύτητας οδηγίας, η οποία παίρνει μία 128-bit λέξη και την περνάει στις τέσσερις συστάδες ως τέσσερις 32-bit εντολές. Αφότου έχει διαιρεθεί η 128-bit λέξη σε τέσσερις 32-bit εντολές εισάγεται η κάθε μία στο στάδιο αποκωδικοποίησης κάθε συστάδας. Το επόμενο στάδιο είναι το στάδιο εκτέλεσης, όπου εκεί εκτελούνται όλες οι λογικές και αριθμητικές πράξεις (υπάρχουν υλοποιημένα σε υλικό μονάδες όπως πολλαπλασιαστής [16], ALU κ.τ.λ.). Το επόμενο στάδιο είναι το στάδιο μνήμης. Όλα τα Sboxes έχουν τοποθετηθεί στο στάδιο μνήμης κάθε συστάδας και σε αυτό τα στάδιο γίνονται όλες οι προσπελάσεις μνημών. Τέλος, είναι το στάδιο write-back όπου γίνεται η εγγραφή σε καταχωρητές.

3. Σύνθεση

Σε αυτό το κεφάλαιο αναλύουμε τη μεθοδολογία σύνθεσης και τα εργαλεία που χρησιμοποιήσαμε για να ολοκληρώσουμε αυτή την διαδικασία. Η σύνθεση είναι ένας γενικός όρος που αναφέρεται στην αυτόματη μετάφραση κώδικα HDL σε ένα ισοδύναμο netlist ψηφιακών κυττάρων. Το netlist δεν είναι παρά η δομική περιγραφή λογικής, ισοδύναμη με τον κώδικα εισαγωγής HDL. Ο κώδικας HDL είναι η ανεξάρτητη από τεχνολογία περιγραφή λογικής που γράφεται από τον σχεδιαστή. Το netlist περιέχει τα συγκεκριμένα ψηφιακά κύτταρα τεχνολογίας που απαιτούνται για το πραγματικό σχέδιο [I1]. Ο πρωτοπόρος στην αγορά στο λογισμικό σύνθεσης λογικής είναι το Synopsys.

3.1 Synopsys

Για τη διαδικασία της σύνθεσης χρησιμοποιήσαμε το Design Compiler του Synopsys [1]. Περιλαμβάνει τα εργαλεία που συνθέτουν τις σχεδιάσεις HDL μας, σε σχεδιάσεις βελτιστοποιημένες τεχνολογικά εξαρτώμενες (σε επίπεδο λογικών πυλών). Υποστηρίζει βελτιστοποιήσεις σχεδίασης σε ταχύτητα, μέγεθος και κατανάλωση ισχύος.



σχήμα 2 - επισκόπηση διαδικασίας σύνθεσης με Design Compiler

3.2 Βασική ροή σύνθεσης

Για την ολοκλήρωση της διαδικασίας της σύνθεσης απαιτείται να ακολουθηθούν μία σειρά από στάδια.

1. Ανάπτυξη αρχείων περιγραφής υλικού.

Τα αρχεία που εισάγονται στον Design Compiler γράφονται συχνά χρησιμοποιώντας μία γλώσσα περιγραφής υλικού (HDL) όπως Verilog ή VHDL. Αυτές οι περιγραφές σχεδίου πρέπει να γραφτούν προσεκτικά για να πετύχουμε τα καλύτερα αποτελέσματα σύνθεσης. Κατά το γράψιμο του κώδικα HDL μας, πρέπει πάντα να εξετάζουμε τις επιπτώσεις σε επίπεδο υλικού. Ένα καλός κώδικας μπορεί να επιφέρει μικρότερες και γρηγορότερες σχεδιάσεις.

2. Προσδιορισμός των βιβλιοθηκών

Διευκρινίζουμε τις βιβλιοθήκες για το Design Compiler με τη χρησιμοποίηση του link_, target_, symbol_, και των synthetic_library εντολών. Οι βιβλιοθήκες τεχνολογίας καθορίζουν το σύνολο σε λογικά κύτταρα του προμηθευτή ημιαγωγών και σχετικές πληροφορίες για αυτά.

3. Διάβασμα της σχεδίασης.

Ο Design compiler χρησιμοποιεί τον μεταγλωττιστή HDL για να διαβάσει τη σχεδίαση σε RTL και σε επίπεδο πυλών.

4. Μοντελοποίηση του περιβάλλοντος της σχεδίασης

Ο Design Compiler απαιτεί να διαμορφώνουμε το περιβάλλον της σχεδίασης που συντίθεται. Αυτό το πρότυπο περιλαμβάνει τους εξωτερικούς παράγοντες λειτουργίας (θερμοκρασία, τάση), τα φορτία, fanouts και τα πρότυπα καλωδίων. Αυτή η μοντελοποίηση επηρεάζει άμεσα τα αποτελέσματα σύνθεσης και βελτιστοποίησης της σχεδίασης.

5. Καθορισμός περιορισμών της σχεδίασης

Ο Design Compiler χρησιμοποιεί κανόνες σχεδίασης και περιορισμούς βελτιστοποίησης για να ελέγξει τη σύνθεση του σχεδίασης.

6. Επιλογή στρατηγικής μεταγλώττισης

Οι δύο πιο διαδεδομένες στρατηγικές μεταγλώττισης είναι η top-down και η bottom-up[3]. Στην top-down η υψηλότερη ιεραρχικά σχεδίαση εισάγεται στον Design Compiler μαζί με τις υποσχεδιάσεις της και μεταγλωττίζονται μαζί. Αντιθέτως, στην bottom-up κάθε υποσχεδίαση εισάγεται στον Design Compiler και μεταγλωττίζεται μόνη της. Έτσι ξεκινάμε από το χαμηλότερο ιεραρχικά επίπεδο και συνεχίζουμε στο αμέσως υψηλότερο. Μόλις τελειώσουμε και με αυτό συνεχίζουμε την ίδια διαδικασία ώσπου να φτάσουμε στο υψηλότερη ιεραρχικά σχεδίαση.

7. Βελτιστοποίηση της σχεδίασης

Αφού πάρουμε κάποια αποτελέσματα που αφορούν τη σχεδίασή μας προσπαθούμε να βελτιστοποιήσουμε τη σχεδίαση σε μέγεθος και απόδοση. Αυτό μπορούμε να το πετύχουμε με κάποιες εντολές του Design Compiler ή με το να προσέξουμε και να αλλάξουμε τα αρχεία εισαγωγής.

8. Ανάλυση και επίλυση σχεδιαστικών προβλημάτων

Σε αυτό το στάδιο βλέπουμε αναλυτικά τα αποτελέσματα που μας δίνει ο Design Compiler όσον αφορούν τη σχεδίασή μας. Στη συνέχεια, αν δεν είμαστε ευχαριστημένοι από τη σχεδίασή μας, επιστρέφουμε στο βήμα 7.

9. Αποθήκευση της σχεδίασης

Αφού τελειώσουμε με τις βελτιστοποιήσεις της σχεδίασής μας σώζουμε το αποτέλεσμα της σύνθεσης.

4. Υλοποίηση

Σε αυτό το κεφάλαιο εστιάζουμε στη διαδικασία που επιλέξαμε για να υλοποιήσουμε τον Ccproc σε ASIC. Αρχικά έπρεπε να τροποποιήσουμε μερικά από τα εξαρτώμενα HDL αρχεία της FPGA σε μία τεχνολογικά-ανεξάρτητη μορφή. Έπρεπε επίσης να αλλάξουμε όλα τα δομικά στοιχεία της σχεδίασης όπου είχαν παραχθεί από τη γεννήτρια κώδικα του ISE. Η γεννήτρια κώδικα της Xilinx χρησιμοποιεί υλικό από μία συγκεκριμένη τεχνολογία που είναι άγνωστη στο Synopsys. Κατόπιν θα διευκρινίσουμε τη βιβλιοθήκη τεχνολογίας που θα χρησιμοποιήσουμε. Έπειτα θα θέσουμε τους περιορισμούς σχεδίασης και θα επιλέξουμε τη στρατηγική μεταγλώττισης. Κατόπιν θα δούμε μερικές βελτιστοποιήσεις και τελικά θα ελέγξουμε το σχέδιό μας και θα αναλύσουμε τα αποτελέσματα.

4.1 Ανάπτυξη Αρχείων VHDL για το Synopsys

Τα περισσότερα από τα δομικά στοιχεία που είναι χαμηλά στην ιεραρχία έχουν παραχθεί από τη γεννήτρια κώδικα της Xilinx. Παραδείγματα τέτοιων στοιχείων είναι οι συγκριτές, οι αθροιστές, οι πολλαπλασιαστές και οι μνήμες, τα οποία έπρεπε να αλλάξουν με τέτοιο τρόπο ώστε το Synopsys να μπορέσει να τα χαρτογραφήσει σε μια τεχνολογία ASIC. Όλα τα δομικά στοιχεία, εκτός από της μνήμες, υλοποιήθηκαν τελικά με ένα ανεξάρτητα τεχνολογικά ύφος VHDL. Η διαδικασία που ακολουθήσαμε είναι η εξής:

1. Αλλάζουμε ένα αρχείο VHDL σε ένα τεχνολογικά ανεξάρτητο αρχείο VHDL

2. Εξετάζουμε αν το νέο στοιχείο λειτουργεί όπως το πρωτότυπο. Εάν δεν λειτουργεί όπως αυτό πρέπει να επιστρέψουμε στο βήμα 1. Εάν λειτουργεί με τον ίδιο τρόπο με το πρωτότυπο συνεχίζουμε στο βήμα 3. Χρησιμοποιήσαμε modelsim για να επαληθεύσουμε τη σχεδίασή μας.

3. Διαβάζουμε το νέο αρχείο VHDL με το synopsys.
4. Ζητάμε από το Synopsys να μεταγλωττίσει τον κώδικά μας.

5. Ζητάμε από Synopsys να παραγάγει τη σχεδίασή μας σε επίπεδο πυλών.

6. Εξετάζουμε το νέο αρχείο VHDL ή VERILOG σε επίπεδο πυλών για να είμαστε βέβαιοι ότι λειτουργεί όπως το εξαρτώμενο (πρωτότυπο) αρχείο. Εάν λειτουργεί όπως αυτό, έχουμε τελειώσει τη διαδικασία μας, εάν όχι πρέπει να επιστρέψουμε στο βήμα 1.

4.1.2 Αλλαγή Μνημών

Η διαδικασία αλλαγής των μνημών είναι διαφορετική. Η διαφορά είναι ότι πρέπει να αποφασίσουμε πρώτα την τεχνολογία που θέλουμε να χρησιμοποιήσουμε. Τα στοιχεία μνήμης δεν γράφονται σε ανεξάρτητη τεχνολογικά VHDL. Οι μνήμες που θα χρησιμοποιήσουμε είναι δομικά στοιχεία που σχεδιάζονται για μια συγκεκριμένη τεχνολογία και έχουν καθορισμένες προδιαγραφές. Ως εκ τούτου, το πρώτο πράγμα που πρέπει να κάνουμε είναι να παραγγείλουμε από τον προμηθευτή ημιαγωγών μνήμες με χαρακτηριστικά που θέλουμε για τη σχεδίασή μας και να είναι συμβατές με την βιβλιοθήκη τεχνολογίας που θέλουμε να χρησιμοποιήσουμε. Στη συνέχεια πρέπει να παράγουμε με τη βοήθεια του Design Compiler το .db αρχείο που αντιστοιχεί στη μνήμη που θέλουμε να χρησιμοποιήσουμε. Όταν έχουμε το .db αρχείο της μνήμης θεωρούμε τη μνήμη μας ως μαύρο κουτί. Για τις περαιτέρω δοκιμές στο modelsim χρησιμοποιούμε το .vhd ή .v αρχείο όπου ο προμηθευτής ημιαγωγών μας μας έχει παράσχει. Εάν οι προδιαγραφές της νέας μνήμης δεν συμπίπτουν με τις προδιαγραφές της μνήμης που θέλουμε να αλλάξουμε, πρέπει να περιβάλλουμε τη μνήμη μας με τα λογικά στοιχεία που θα την αναγκάσουν να λειτουργήσει με τον ίδιο τρόπο που λειτουργεί η μνήμη της FPGA (τη μνήμη που θέλουμε να αντικαταστήσουμε). Τα βήματα που πρέπει να ακολουθήσουμε για να αλλάξουμε μια μνήμη από μία FPGA σε μία τεχνολογία ASIC είναι:

1. Αποφασίζουμε την τεχνολογία που θέλουμε να χρησιμοποιήσουμε.

2. Παραγγέλνουμε από τον προμηθευτή ημιαγωγών μας μία μνήμη συμβατή με την τεχνολογία που θα χρησιμοποιήσουμε τελικά για τη σχεδίασή μας. Η μνήμη που παραγγέλνουμε πρέπει να έχει παρόμοιες προδιαγραφές με την μνήμη που προσπαθούμε να αλλάξουμε (μέγεθος, σήματα ελέγχου).

3. Περιβάλλουμε τη μνήμη μας με επιπλέον λογική σε ένα νέο αρχείο vhdl χρησιμοποιώντας το .vhd της μνήμης που έχουμε ως δομικό στοιχείο (μαύρο κουτί). Ο σκοπός μας είναι ότι η νέα μνήμη με την πρόσθετη λογική θα έχει την ίδια συμπεριφορά με τη μνήμη που θέλουμε να αλλάξουμε. Θέλουμε να είμαστε βέβαιοι ότι το νέο δομικό στοιχείο λειτουργεί ακριβώς με τον ίδιο τρόπο με το πρωτότυπο, έτσι προσπαθούμε να κάνουμε τα πεδία δοκιμών μας να περιλάβουν όσο το δυνατόν περισσότερες περιπτώσεις. Εάν καταφέρουμε τελικά το νέο στοιχείο μας να συμπεριφερθεί με τον ίδιο τρόπο με αυτό που θέλουμε να αλλάξουμε, προχωράμε στο επόμενο βήμα, εάν όχι, επαναλαμβάνουμε το βήμα 3.

4. Διαβάζουμε με τη βοήθεια του Design Compiler το .lib αρχείο που αντιστοιχεί στη νέα μνήμη μας και παράγουμε το ισοδύναμο .db.

5. Τώρα διαβάζουμε το νέο αρχείο vhdl μας με την πρόσθετη λογική και το συνδέουμε με τη βιβλιοθήκη της νέας μνήμης μας (με το .db).

6. Μεταγλωττίζουμε το αρχείο vhdl μας με το Synopsys.

7. Ζητάμε από Synopsys να παραγάγει τη σχεδίασή μας σε επίπεδο πυλών.

8. Εξετάζουμε τη νέα μνήμη σε επίπεδο πυλών για να είμαστε βέβαιοι ότι λειτουργεί όπως η μνήμη που αλλάξαμε. Εάν λειτουργεί όπως το πρωτότυπο, έχουμε τελειώσει τη διαδικασία μας, εάν όχι, πρέπει να επιστρέψουμε στο βήμα 3.

4.2 Προσδιορισμός Βιβλιοθηκών

Το επόμενο βήμα της διαδικασίας μας ήταν να αποφασιστεί η βιβλιοθήκη λογικών κυττάρων. Επιλέξαμε την τυποποιημένη βιβλιοθήκη λογικών κυττάρων υψηλής πυκνότητας 0.13um. Η τυποποιημένη βιβλιοθήκη κυττάρων υψηλής πυκνότητας 0.13um παρέχει στα σημερινά προηγμένα εργαλεία σύνθεσης τις απαραίτητες δομικές μονάδες για να εφαρμόσει αποτελεσματικά την διαδικασία της σύνθεσης για μια αποδοτική σχεδίαση.

4.3 Προσδιορισμός Περιορισμών

Αποφασίσαμε να επιλέξουμε μία μεγάλη περίοδο ρολογιού για την πρώτη μεταγλώττιση προκειμένου να δούμε εάν η σχεδίασή μας λειτουργεί καλά. Αργότερα στο στάδιο βελτιστοποίησης θα ασχοληθούμε για την επίτευξη της μέγιστης συχνότητας. Έτσι επιλέγουμε μία περίοδο 7 ns(142MHz).

4.4 Επιλογή Στρατηγικής Μεταγλώττισης

Η στρατηγική που επιλέξαμε να ακολουθήσουμε είναι η bottom-up. Προτιμήσαμε αυτήν την στρατηγική από την top-down επειδή θελήσαμε να ασχοληθούμε με κάθε υποσχεδίαση ξεχωριστά. Με αυτόν τον τρόπο έχουμε το πλεονέκτημα το ότι είμαστε βέβαιοι για τη σταθερότητα κάθε δομικού στοιχείου. Σε αυτήν τη στρατηγική βλέπουμε τη σχεδίασή μας ως δέντρο και αρχίζουμε να μεταγλωττίζουμε τα φύλλα του. Στο επόμενο βήμα μεταγλωττίζουμε τα δομικά στοιχεία του επόμενου υψηλότερου επιπέδου της ιεραρχίας. Συνεχίζουμε τη διαδικασία μέχρι να φτάσουμε στο κορυφαίο στοιχείο.

4.5 Βελτιστοποίηση της Σχεδίασης

Όπως αναφέρθηκε και παραπάνω, η σχεδίαση υλοποιήθηκε αρχικά στα 142 MHZ. Η πρώτη μας υπόθεση ήταν ότι η κρίσιμη πορεία μας θα έπρεπε να

είναι η μνήμη εντολών (350 MHZ). Αφότου είχαμε τα αποτελέσματα της σύνθεσης ανακαλύψαμε ότι η κρίσιμη πορεία μας βρέθηκε στο στάδιο εκτέλεσης στον πολλαπλασιαστή (mmult32x32). Η σχεδίασή μας δεν μπορούσε να υπερβεί τα 200 MHZ λόγω του πολλαπλασιαστή. Ο κώδικας vhdl προσπαθούσε να μιμηθεί τον πρωτότυπο πολλαπλασιαστή. Η υπόθεση που κάναμε ήταν ότι οι κατάλογοι (από τα δύο στάδια σωληνώσεων) δεν είχαν ισορροπηθεί καλά. Κατά συνέπεια χρησιμοποιήσαμε την εντολή balance_register πάνω στον πολλαπλασιαστή. Τα νέα αποτελέσματα που πήραμε ήταν σαφώς καλύτερα, με τη σχεδίασή μας να φτάνει στα 250 MHZ (βελτιστοποίηση 25%). Η νέα κρίσιμη πορεία μας είναι τώρα η ALU (επειδή υποστηρίζει τις διπλές εντολές).

decstageA (register) -> exstageA (ALU input) 0.16 ns
exstageA (ALU input) -> exstageA (ALU output) 3.91 ns
exstageA (ALU output) -> AluOutMemRegister 3.94 ns

Δεν προσπαθήσαμε να αλλάξουμε την ALU επειδή στα συστήματα κρυπτογράφησης συνηθίζονται διπλές οδηγίες και αυτό σημαίνει ότι θα ξοδεύαμε δύο κύκλους ρολογιών αντί του ενός, το οποίο ξοδεύουμε τώρα. Αυτό θα σήμαινε τη μείωση του throughput της σχεδίασής μας.

4.6 Επαλήθευση και Αποτελέσματα

Χρησιμοποιήσαμε modelsim για την επαλήθευση της σχεδίασής μας. Έπρεπε να συγκρίνουμε τις κυματομορφές που είχαμε από την εφαρμογή της FPGA με τις κυματομορφές από τη σχεδίασή μας σε επίπεδο πυλών.

4.6.1 Λειτουργική και Χρονική Προσομοίωση

Κάθε φορά που αλλάζαμε μία ενότητα σε επίπεδο πυλών την αντι-καθιστούσαμε με την πρωτότυπη στον Ccproc. Κατόπιν ελέγχαμε τη σχεδίαση με το νέο δομικό στοιχείο αν λειτουργούσε σωστά. Συγκρίναμε σήμα προς σήμα τις κυματομορφές της πρωτότυπης σχεδίασης με εκείνες της νέας.

Συγκρίναμε τις κυματομορφές κάθε αλγορίθμου που ο Ccproc υποστηρίζει (AES, SERPENT, MARS, TWOFISH, RC6). Όταν ελέγξαμε όλη την ASIC σχεδίασή μας (Ccproc σε επίπεδο πυλών) ότι είχε την ίδια λειτουργία με την εφαρμογή FPGA προχωρήσαμε στην χρονική προσομοίωση. Για να ολοκληρωθεί η χρονική προσομοίωση εισήγαμε το .sdf αρχείο από το Synopsys το οποίο εμπεριέχει όλες τις χρονικές πληροφορίες της σχεδίασής μας. Η σχεδίασή μας έχει τα ίδια αποτελέσματα λειτουργίας με την αρχική.

4.6.2 Αποτελέσματα

Η απόδοση της σχεδίασής μας έφθασε στα 250MHz όπως είδαμε και στο 4.5. Η ανώτατη συχνότητα σε FPGA ήταν 108 MHZ. Το σχέδιό μας χρησιμοποίησε 93.185 λογικά κύτταρα και 94885 καλώδια. Στην FPGA χρησιμοποιήθηκαν 275.452 πύλες. Τα αποτελέσματα από το Synopsys φαίνονται παρακάτω :

Αποτελέσματα	αριθμός	Χώρος μm^2	%χώρος
Cells	93185	5.343.404	100
sp_mem64x32	4	101.770	1,9
sp_mem256x32	88	3773985,531	70,6
sp_mem256x128	1	146625	2,7
sp_mem512x32	4	279.299	5,2
memories	97	4301679,344	80,5
flip-flop and latch	8796	320644,2271	6
non combinational	8893	4622323,571	86,5
combinational	84292	721080,429	13,4

Πίνακας 1 - αποτελέσματα

Στον πίνακα 2 βλέπουμε τις διαφορές μεταξύ της ASIC σχεδίασης και της FPGA σχεδίασης.

Σύγκριση	Max frequency(Mhz)	αλγόριθμοι	Clock Cycles	Throughput(Mbits/sec)
ASIC	250	AES	79	405
		MARS	338	94
		RC6	242	132
		SERPENT	375	85
		TWOFISH	178	179
FPGA	108	AES	79	174
		MARS	338	40
		RC6	242	57
		SERPENT	375	36
		TWOFISH	178	77

Πίνακας 2 - Σύγκριση

5. Εμπειρίες και Συμπεράσματα

Όπως αναφέρθηκε και στην αρχή ψάχνουμε τις λύσεις που θα παράσχουν το γρηγορότερο χρόνο στην αγορά, το χαμηλότερο κόστος και την υψηλότερη απόδοση για τις διαδοχικές γενεές των προϊόντων. Οι FPGAs πάσχουν από υψηλό κόστος ανά-μονάδων, χαμηλή απόδοση, χαμηλά επίπεδα ολοκλήρωσης και μεγάλη κατανάλωση ισχύος. Αντίθετα, τα ASICs έχουν αρκετά χαμηλότερο κόστος ανά μονάδα σε επίπεδο μαζικής παραγωγής, προσφέρουν υψηλότερες αποδόσεις και καταναλώνουν λιγότερη ισχύ .

Σε αυτή την εργασία προσπαθήσαμε να αλλάξουμε μία FPGA εφαρμογή σε μία ASIC για τους λόγους που αναφέραμε παραπάνω. Κατορθώσαμε επιτυχώς να υλοποιήσουμε τον CCproc σε ASIC.

Η εμπειρία από τη διαδικασία σύνθεσης είναι πολύ χρήσιμη σε έναν σχεδιαστή υλικού. Αρχικά, ο σχεδιαστής, μαθαίνει πώς να γράφει τεχνολογικά ανεξάρτητη VHDL. Μαθαίνει επίσης την ακριβή μετάφραση σε υλικό του behavioral κώδικά του. Η επόμενη σημαντική εμπειρία για έναν σχεδιαστή υλικού είναι ότι μαθαίνει πώς να χρησιμοποιεί σημαντικά CAD εργαλεία (όπως ISE και SYNOPSYS).

Για να συνοψίσουμε, μεγάλο μέρος των FPGA εφαρμογών καταλήγουν σε ASIC όταν θέλουμε να επιτύχουμε τους στόχους που αναφέραμε παραπάνω.

6. Αναφορές

[1] Synopsys Corporation, Design Compiler™ User Guide Version 2002.05, June 2002

[2] Sreesa Akella, Guidelines For Design Synthesis Using Synopsys Design Compiler, Department of Computer Science Engineering University of South Carolina Columbia, December 2000

[3] Synosys corporation, Design Compiler Technology Backgrounder, April 2006

[4] M. Keating and P. Bricaud, “Reuse Methodology Manual for System-on-a-ChipDesigns 3rdEdition”, Kluwer Academic Publishers Norwell, 1998

[5] D.Theodoropoulos, *“CCproc: A custom VLIW cryptography co-processor for symmetric key cipher”*”, Master Thesis, department of electronic and computer engineering, Technical Univercity of Crete, 2005

[6] Howard M. Heys, “A tutorial on linear and differential cryptanalysis”, Cryptologia, July 2002

[7] Joan Daemen, Vincent Rijmen, “AES Proposal: Rijndael”, Document Version 2, March 9, 1999

[8] C. Burwick, D. Coppersmith, E. D’Avignon, R. Gennaro, S. Halevi, C. Jutla, S. M. Matyas Jr., L. O’Connor, M. Peyravian, D. Safford, N. Zunic, “MARS - a candidate cipher for AES”,Second Aes Workshop, IBM Corporation, 1999

[9] Bruce Scheier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, Niels Ferguson, “Twofish: A 128-bit Block Cipher”, Wiley ,15 June 1998

[10] Ronald L. Rivest, M.J.B. Robshaw, R. Sidney, Y.L. Yin, “The RC6 Block Cipher”, IT, August 20, 1998.

[11] Ross Anderson, Eli Biham, Lars Knudsen, “Serpent: A Proposal for the Advanced Encryption Standard”, 5th workshop on Fast Software Encryption, 1998.

[12] B. Schneier, “Description of a new variable-length key, 64-bit block cipher (Blowfish)”, Fast Software Encryption, Cambridge Security Workshop Proceedings (December 1993), Springer-Verlag, 1994.

[13] Ronald L. Rivest, “The RC5 Algorithm”, RSA Security, October 1996

[14] MediaCrypt, “International Data Encryption Algorithm”, Technical Description

[15] J. L. Smith, "The design of Lucifer: A cryptographic device for data communications, " Technical report, IBM T.J. Watson Research Center, Yorktown Heights, N.Y., 10598, U.S.A., 1971.

[16] Xilinx Corporation, “Xtreme DSP design considerations user guide”, February 2005

[17] J. Burke, J. McDonald, T. Austin, “Architectural Support for Fast Symmetric-Key Cryptography”, ACM Press, 2000

[18] A. Murat Fiskiran and Ruby B. Lee, “Performance Impact of Addressing Modes on Encryption Algorithms”, Proceedings of the International Conference on Computer Design (ICCD 2001), pp. 542-545, September 2001

[19] A. Murat Fiskiran and Ruby B. Lee, “On-Chip Lookup Tables for Fast Symmetric-Key Encryption”, Proceedings of the IEEE 16th International Conference on Application-Specific Systems, Architectures and Processors

(ASAP), pp. 356-363, July 23-25, 2005

[20] M. Jung, F. Madlener, M. Ernst and S. A. Huss, “A Reconfigurable Coprocessor for Finite Field Multiplication in GF(28)”, Darmstadt University of Technology, Germany, IEEE Workshop on Heterogeneous reconfigurable Systems on Chip, Hamburg, April 2002

Internet Links:

[I1] http://hdlplanet.tripod.com/
[I2] http://www.ssh.com/support/cryptography/introduction/
[I3] http://www.soccentral.com
[I4] http://en.wikipedia.org/
[I5] www.fact-index.com/r/rc/rc4_cipher.html
[I6] www.aci.net/kalliste/des.htm
[I7] http://csrc.nist.gov/CryptoToolkit/aes/round1/round1.htm#algorithms

Παράρτημα Α : Υπόβαθρο και η Αρχιτεκτονική Ccproc

In this chapter we focus on describing Cryptium. Firstly we will see some very important things about cryptography and the most popular and safe ciphers that nowdays are used. Then we focus on describing CCproc's Architecture as implemented before synthesis in Synopsys.

2.1 Cryptography

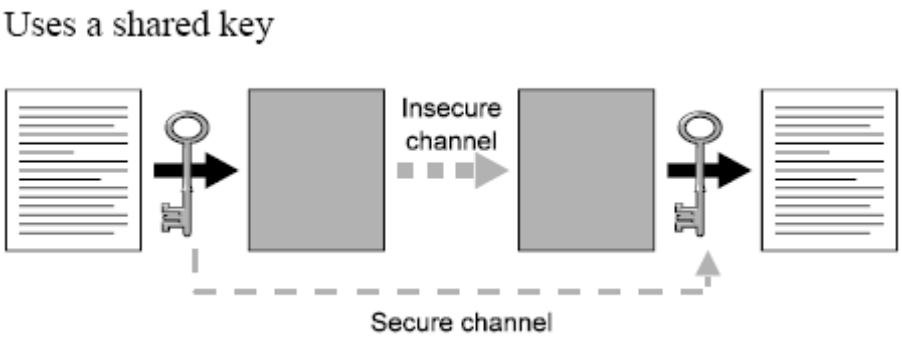
Cryptography [I3] (derived from Greek κρυπτός kryptós "hidden," and γράφειν gráfein "to write") is a discipline of mathematics concerned with information security and related issues, particularly encryption, authentication, and access control. Its purpose is to hide the meaning of a message rather than its existence. In modern times, it has also branched out into computer science. Cryptography is central to the techniques used in computer and network security for such things as access control and information confidentiality.

A cryptographic algorithm, or cipher, is a mathematical function used in the encryption and decryption process. A cryptographic algorithm works in combination with a key(a word, number, or phrase) to encrypt the plaintext. The same plaintext encrypts to different ciphertext with different keys. The security of encrypted data is entirely dependent on two things: the strength of the cryptographic algorithm and the secrecy of the key.

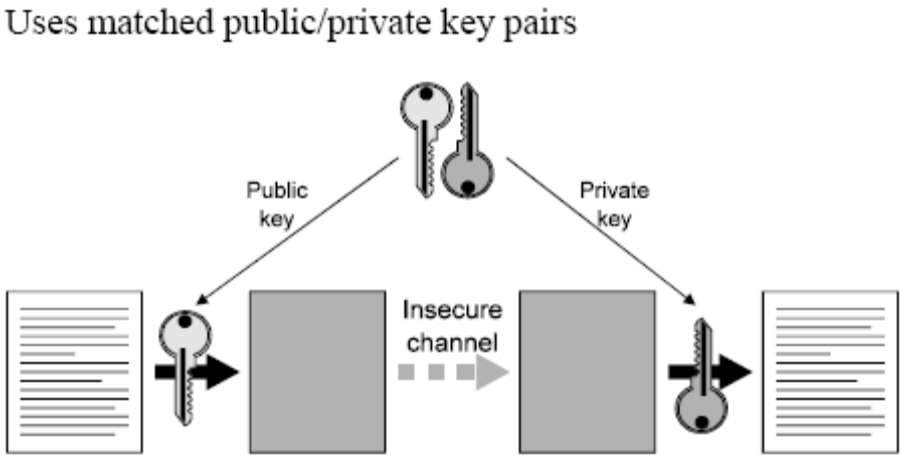
2.2 Public and Private Key Ciphers

Two forms of cryptography are commonly used in information systems today, which are shown in next figure:

- Symmetric Key or Private Key ciphers.
- Asymmetric Key or Public Key ciphers.



Private Key Cipher



Public Key Cipher

Figure1 - Public-key (down) and Symmetric-key (up) algorithms

As it is shown from figure 1, public key ciphers, use two types of keys, a public that is used to encrypt data, and a private that is used to decrypt the encrypted data. On the other hand, private key ciphers use only a private key for both data encryption and decryption. Another fact is that private key ciphers are much faster than public key ciphers[I2], so during a secure data exchange, at first a private key is shared between the users with a public key cipher and then all other data are being transmitted with a private key cipher, that uses the previous private key for encryption / decryption. More

specifically, the entire process of a secure communication channel establishment is as follows:

1. Person A sends his public key to person B through an unsecured communication channel.
2. B encrypts its secret key with a public key cipher and sends it to A.
3. A decrypts the encrypted secret key with his private key.
4. From this moment all data are encrypted / decrypted with symmetric key ciphers.

However, there still are various ways to decipher encrypted communications without knowing the proper keys. Examples are *brute force* attacks, where all possible keys are being tried, *ciphertext-only* attacks, where the attacker tries to guess the plaintext with theoretical methods such linear and differential cryptanalysis[6], and *man-in-the-middle* attacks, where an adversary positions himself between A and B persons and intercepts each signal they send to each other[I2].

2.3 Private Key Algorithms

In Orded to understand Why Cryptium supports specific private key ciphers in this chapter we focus on symmetric key ciphers. Specific attention was paid to choose which algorithms to study, because many of them have weaknesses. So, in order to make our analysis as complete as possible, the following algorithms were chosen: Rijndael [7] MARS [8], Twofish [9] Serpent [10], Blowfish [12], RC4 [I5], DES (Data Encryption Standard) [I6], RC5 [13], International Data Encryption Standard (IDEA) [14]. This group contains only the five AES (Advanced Encryption Standard) finalists of round 2 [I7], i.e. the strongest ones of the AES candidates. It also, has the previous standard encryption algorithm DES, plus Blowfish, IDEA, RC4, RC5 which are older and widely used. All these facts it is believed that led in a very realistic and representative choice of the best symmetric key ciphers ever designed, in order to proceed into further analysis.

Every symmetric cipher has the following three important parameters:

1. The number of bits in its secret key.
2. The size of the data block that operates on (also in bits).
3. The number of processing rounds.

Depending on the size of data block, symmetric ciphers have two categories:

- Block ciphers that operate on large data blocks.
- Stream ciphers that operate usually on one bit.

Table 1 shows all these attributes for the ciphers mentioned before. Figure 2 shows a generic schematic for the encryption / decryption process. Before message encryption starts, every symmetric cipher has an initialization phase, which is mainly the key expansion. More specifically, the secret key is processed in a certain way and the result is a number of other keys that some of them are used in different encipher / decipher rounds. In rare cases, also other required operations occur, such as in Blowfish, where its substitution boxes are being created.

Algorithm	Type	Key size (bits)	Block size (bits)
Blowfish	Block	up to 448	64
Twofish	Block	up to 256	128
DES	Block	64	64
Rijndael	Block	up to 256	128
MARS	Block	128 to 400	128
Serpent	Block	256	128
IDEA	Block	128	64
RC4	Stream	up to 2048	8
RC5	Block	up to 2040	>0
RC6	Block	up to 2040	>0

Table 1 – Symmetric ciphers categories

After the entire initialization phase is completed, encryption process begins.

The latter consists of a certain number of various types of arithmetic operations that are being applied on the plaintext for a specific number of rounds. Once the defined round number has been reached, encryption process is finished and ciphertext is ready to be transmitted. Decryption process in most cases, if it is not identical, then it is almost the same as the encryption process, where again various types of arithmetic operations are being performed on ciphertext for a specific number of rounds, in order the recipient to retrieve the original message.

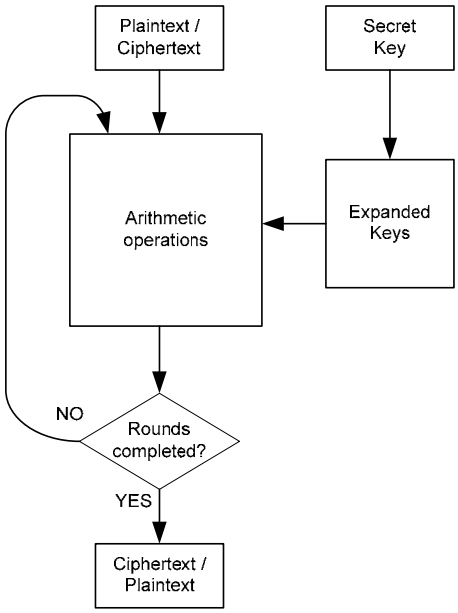


figure2 – Encryption / Decryption process

Symmetric key ciphers are designed in such a way that it will as difficult to break as possible. In order to achieve the highest security level, designers have to consider, among others, the kind of arithmetic operations that will be used, the size of the data block and secret key, and the number of processing rounds.

Data block and key size affect the hardware resources that will be needed, mostly the number of registers and memory allocation. Key size also heavily contributes to the cipher's security level, because, when using brute force attack, the required computing power increases exponentially with it. Today an acceptable key size is at least 80-bit, while 128-bit will probably remain

unbreakable by brute force attacks for the foreseeable future.

The number of rounds also affects considerably a cipher's security level, because, in each one of them, previous processed data get "scrambled" even more. It is on the designer's decision of how many total rounds a cipher will consist of. Fewer rounds mean lesser security, but on the other hand, quicker data block processing. As a result, the appropriate round number depends on the round's itself strength, i.e. the arithmetic operations that are applied to a data block in each one of them.

When the above ciphers had been designed, processors were still 32-bit and, consequently, most of the arithmetic operations are chosen to take advantage of it. Also, it is imperative that these operations present rapid bit diffusion, in order to increase the cipher's security. Theodoropoulos [5] concluded that the operations and structures most commonly used are:

10. Unsigned addition and subtraction modulo 2^{32}
11. Multiplication modulo 2^{32}
12. Exclusive or (xor) between 32-bit data
13. Fixed shifts and rotations
14. Data depended shifts and rotations
15. Finite field polynomial multiplication in 2^8 modulo a prime polynomial
16. Expansions and permutations (Xboxes)
17. Substitution boxes (Sboxes)
18. Feistel network structures [15].

2.4 Algorithm Implementations and Symmetric cipher accelerators

There are three kind of algorithm implementation :

- C and Assebly Implementations
- Algorithm Specific Hardware Implementations
- Symmetric Ciphers ISA extensions and Hardware Co-Processors

Hardware implementations provide higher speed than software implementations. The algorithm specific hardware Implementation category provides ultra speed performance for each symmetric algorithm, because of the dedicated hardware processors. The last category , is somehow “in the middle” of the previous two and it focuses on symmetric ciphers specific hardware co-processors. These designs may extend an existing processor’s architecture in order to support more efficiently symmetric ciphers, or even introduce new co-processors specifically for some of them. As it may be easily comprehended, this category can be characterized as the hardest of all, because it requires deep parallel analysis of many symmetric ciphers and extra effort, in order to obtain a balanced between performance and flexibility design.

Burke et al in [17] are trying to improve the performance of symmetric ciphers for the Alpha 21264 processor by examining eight algorithms. After analysis of bottleneck in these ciphers, they conclude to an extended ISA that consists of hardware rotations, modulo multiplication, permutation and Sbox access instructions and may achieve up to a 74% speedup over the baseline machine.

Murat Fiskiran et al in [18] study the effect of different addressing modes that can be used to calculate the effective address during Sbox access. More specifically they determinate how performance is affected on 1, 2, 4 and 8 wide EPIC (Explicitly Parallel Instruction Computer) processors depending on

addressing mode of the architecture, issue width of the processor and number of memory ports. The results indicate that speedups exceeding 2x can be obtained when fast addressing modes are used.

Another similar approach comes from [19], where the same authors describe a new hardware module called PTLU (Parallel Table Look Up). It consists of multiple LUTs that can be accessed in parallel and its purpose is again Sbox access acceleration. Their results show maximum speedups of 7.7x for AES and 5.4x for DES, all tested on a single-issue 64-bit RISC processor.

Finally, Jung et al in [20]are trying to accelerate multiplication in GF (2^n) execution, an operation rather frequent in symmetric ciphers as stated in section 2.2. To be more specific, in this project they automate the design process for this kind of multipliers with VHDL (Very high speed intergraded circuit Hardware Description Language) and compare their results with other GF multipliers both on FPGA and ASIC implementations.

2.5 The Ccproc Architecture

Ccproc is a [6] VLIW symmetric cipher co-processor who is flexible enough to support all Advanced Encryption Standard (AES) round two finalists, but also potential to new ones symmetric ciphers.
It has been observed that all AES round two finalists treat 128-bit plaintext as four 32-bit words. Many times during processing, symmetric ciphers require 64-bit, 96-bit or even 128-bit data values at the same time in order to proceed. This led to the decision to build a VLIW co-processor with four clusters, as shown in figure 3.

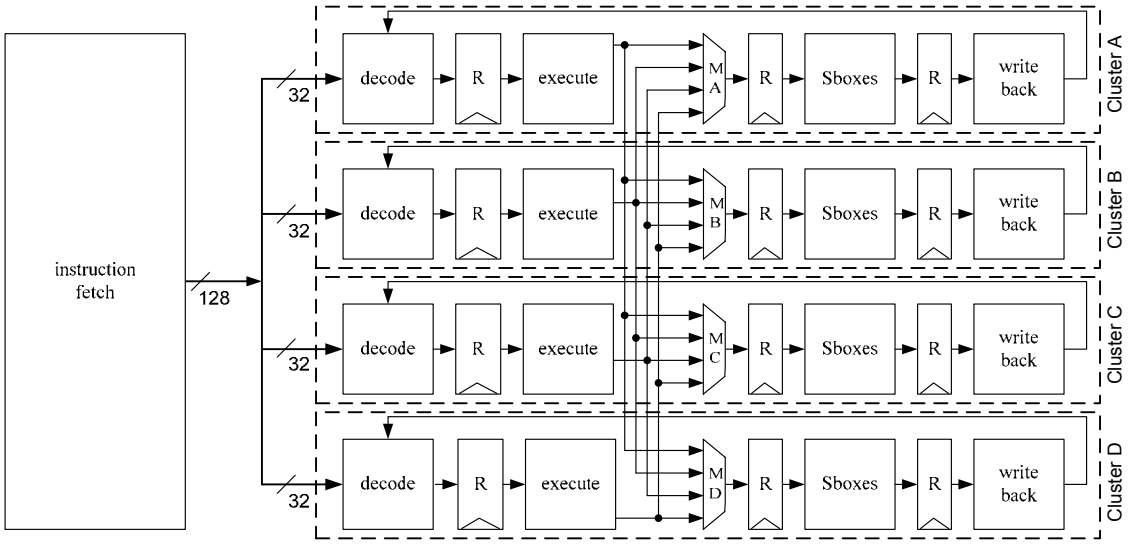


Figure 3 – CCproc's schematic overview

First of all, there is an instruction fetch unit, which takes on to fetch a 128-bit data value and pass it to the four clusters as four 32-bit instructions. Bits 127 down to 96 form cluster's A instruction, bits 95 down to 64 form cluster's B instruction, bits 63 down to 32 form cluster's C instruction and bits 31 down to 0 form cluster's D instruction. As it can be seen in figure 4, there is an instruction cache 256x128 size, where quads of 32-bit instructions are stored. PC is the program counter register that holds the instruction cache's access address. Multiplexer A selects with "nPCsel" between PC address and an address "label" generated from the "loop controller" unit, in case there is a loop instruction, while multiplexer B selects again with "nPCsel" between "label" and PC to pass into the adder for next instruction's address effective calculation.

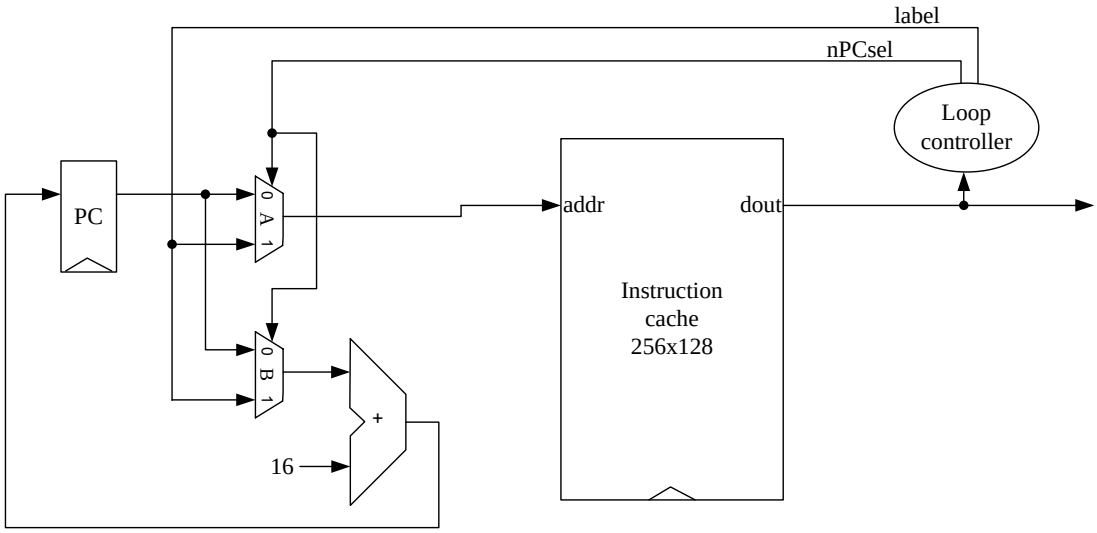


Figure 4 – CCproc's instruction fetch unit

.

After a 128-bit data value has been fetched, it is separated to four 32-bit instructions that are directly connected with each cluster's decode stage, whose high level schematic is shown in figure 5.

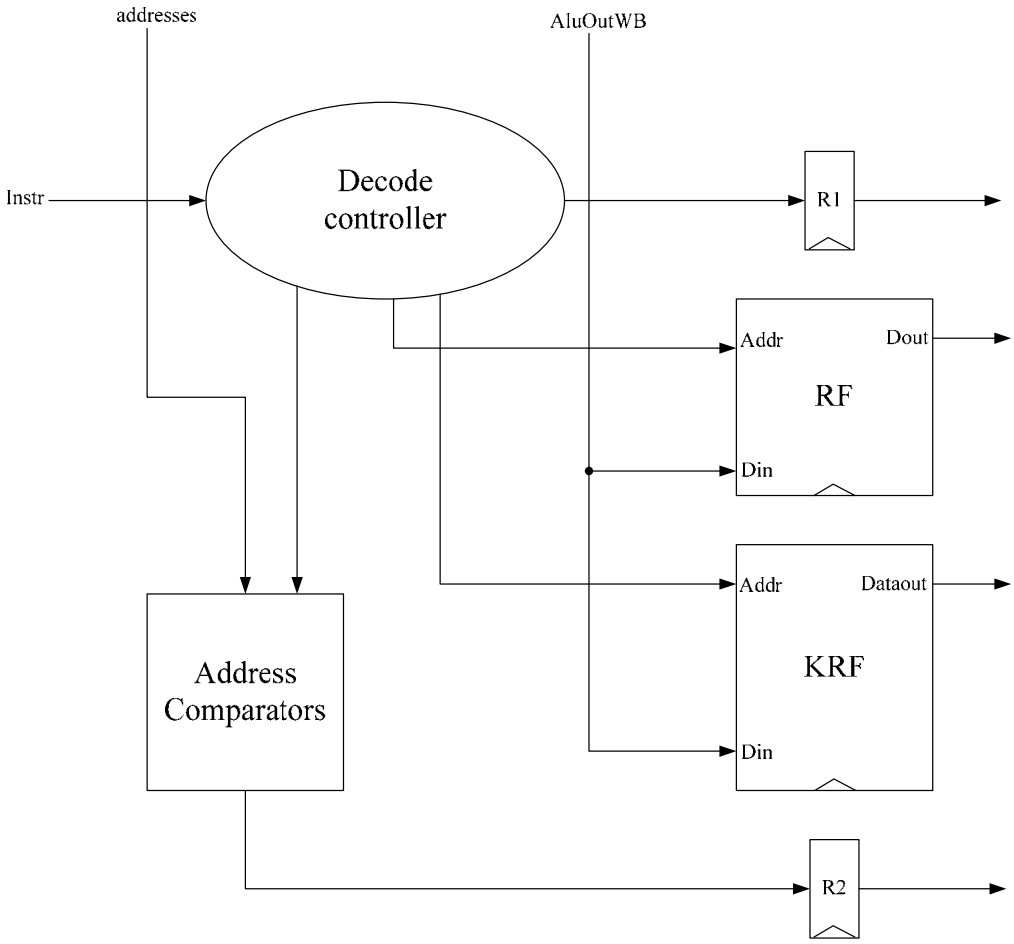


Figure 5 – Decode unit

The main unit in this stage is the Decode controller, which decodes each 32-bit “Instr” instruction comes from the “instruction fetch” unit. It produces valid RF and KRF addresses, plus many other control signals that pass through the next stages via the R1 pipeline register. “AluOutWB” contains every data that will be stored to RF or KRF. Finally there is an “Address Comparator” unit that compares “addresses” signals, which contain RF write addresses to next stages with current’s instruction target register in RF. Every control signal that this unit produces, pass through pipeline register R2. It should be noted that R1 and R2 pipeline registers have the same meaning with the R one between “decode” and “execution” stages in figure 3.

The register file was designed, as shown in figure 6, having three copies of an 8x32 register set.With this design we read up to three different registers in

a single clock cycle RF is fully synchronous, which means that reading from and writing to it occurs on the positive clock edge. During a read operation, each one of them can provide an independent 32-bit register, through each one of the “RdAddr1”, “RdAddr2” and “RdAddr3” address signals, resulting up to three 32-bit registers to “DataOut1”, “DataOut2” and “DataOut3” signals in single clock cycle. This is particularly useful when double-instructions occur where three operands are needed at the same time. However, all copies must always be identical to each other, so there is only one “WrAddr”, “WrEn” and “DataIn” signal, writing every time the same data in each RF core.

When one or more of the “RdAddr1”, “RdAddr2” and “RdAddr3” signals are equal to the “WrAddr” signal, there is logic that passes immediately “DataIn” value to the appropriate “DataOutX” signal. In other words, this RF utilizes a Read-After-Write scheme.

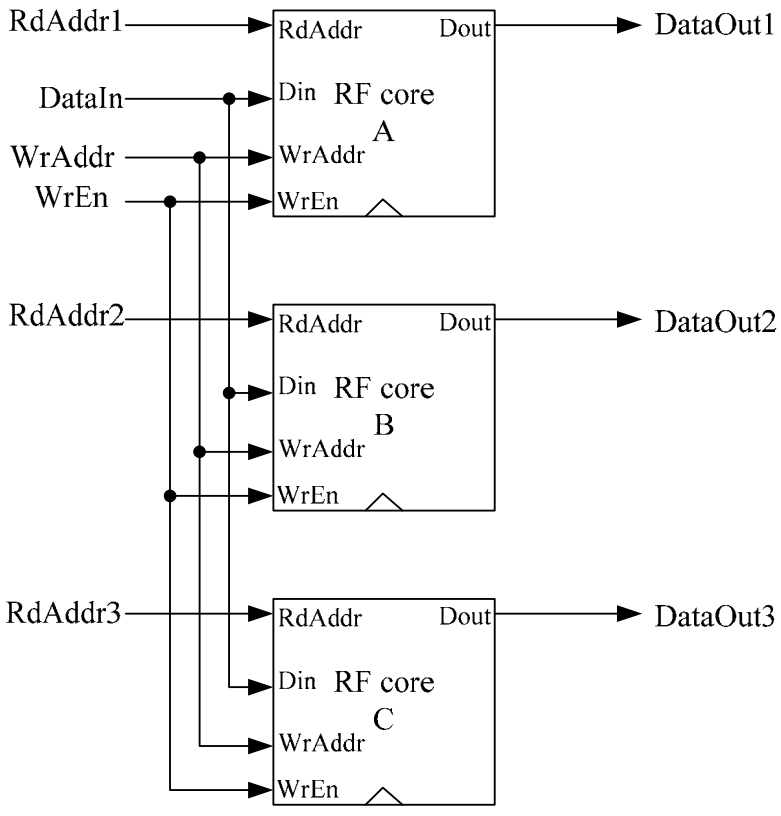


figure 6 - Register File

The KRF is a special RAM in each cluster’s decode stage, where a cipher’s expanded keys are stored. Every KRF is a 64x32 data space, meaning it has

sixty four 32-bit registers

Next stage is the execution stage, where all logic and arithmetic operations are performed and is shown in figure 7.

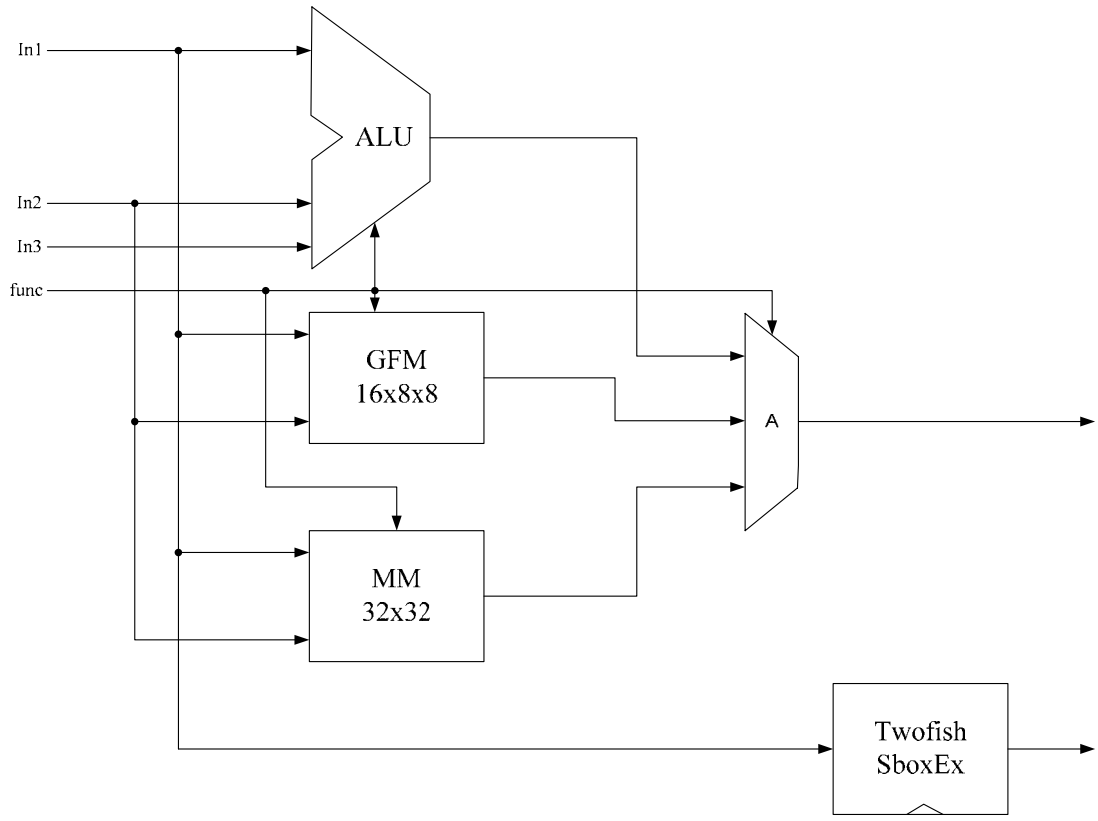


Figure 7 – Execution stage

The ALU, GF multiplier 16x8x8 and MM functional units are all placed here. RF outputs that have come from the previous pipeline stage are their inputs and depending on the operation that needs to be performed, multiplexer A selects the appropriate result. As it was shown in figure 7, before the next pipeline register, there is a “MX” (X=A, B, C, D) 4-to-1 multiplexer that selects among the multiplexer’s A outputs of each cluster’s execution stage. These multiplexers are used in case data need to be switched between clusters, by using the appropriate “move” instruction. Also in this stage there is Twofish’s Sboxes first portion.

In symmetric ciphers there is a high frequency occurrence of two dependent, back-to-back instructions. Examples are double additions, subtractions and XOR, and addition or subtraction followed by a XOR. In order to save valuable computing clock cycles it was decided that this type of double-instructions should be included in CCproc's ISA. This is the reason that the processor's ALU has three 32-bit inputs and one 32-bit output. As it can be observed in figure 8, there are three 32-bit ASUs (Addition / Subtraction Units), three 2-input 32-bit xors and three multiplexers. ASU A adds or subtracts inputs "In1" and In2", while gate A makes a xor operation between them. If there is a double-instruction, results from ASU A and gate A, are passed, in combination with "In3", through ASUs B and C, and gates B and C. Finally multiplexers A, B and C are used to select appropriate data depending on the value of "func" field while, in arrows before multiplexer C is shown operation allocation. ALU instructions have the below specific format:

Result ← (In1 op1 In2) op2 In3

where "In1", "In2" and "In3" are the three "ALU core 1" inputs and op1, op2 are the two operations that may be performed.

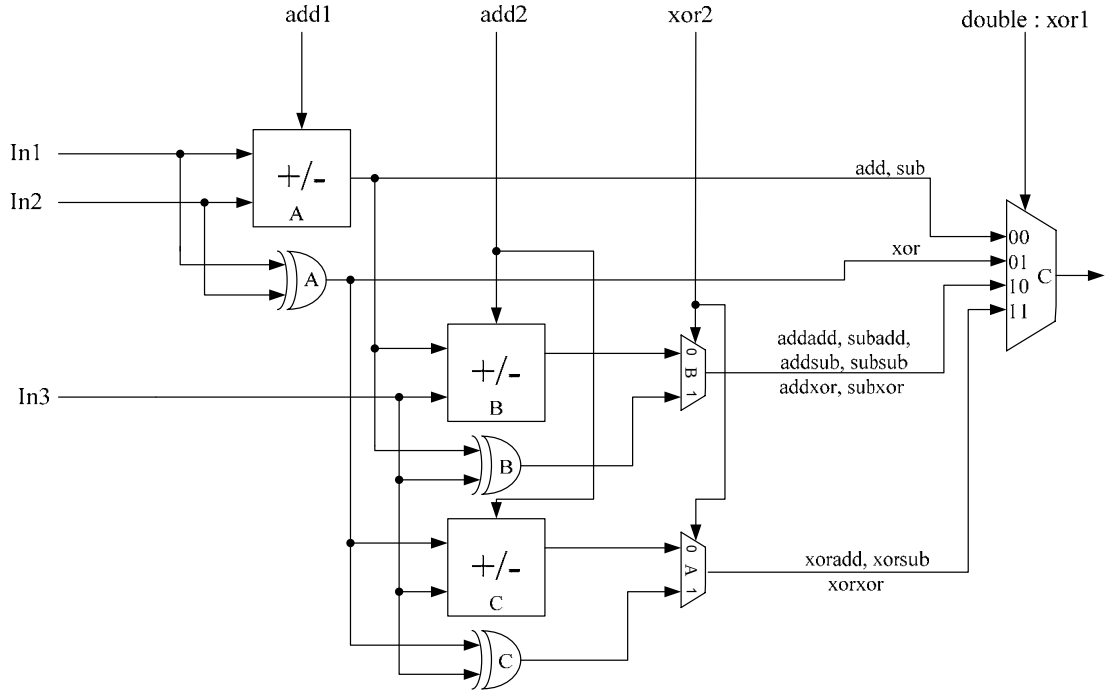


figure 8 - ALU

Besides the “ALU core 1” there is another functional unit, called “ALU core 2”, that is used for data rotations and shifts, and is shown in figure 9. More specifically there are two SRUs (Shift / Rotate Units), which take as inputs 32-bit “DataIn” that will be shifted / rotated, a 5-bit “amount” that indicates the specific shift / rotation amount plus a “shift / rotate” signal that selects shift or rotation. Once the two SRU’s have finished, multiplexer A selects the appropriate direction, depending on instruction that were issued.

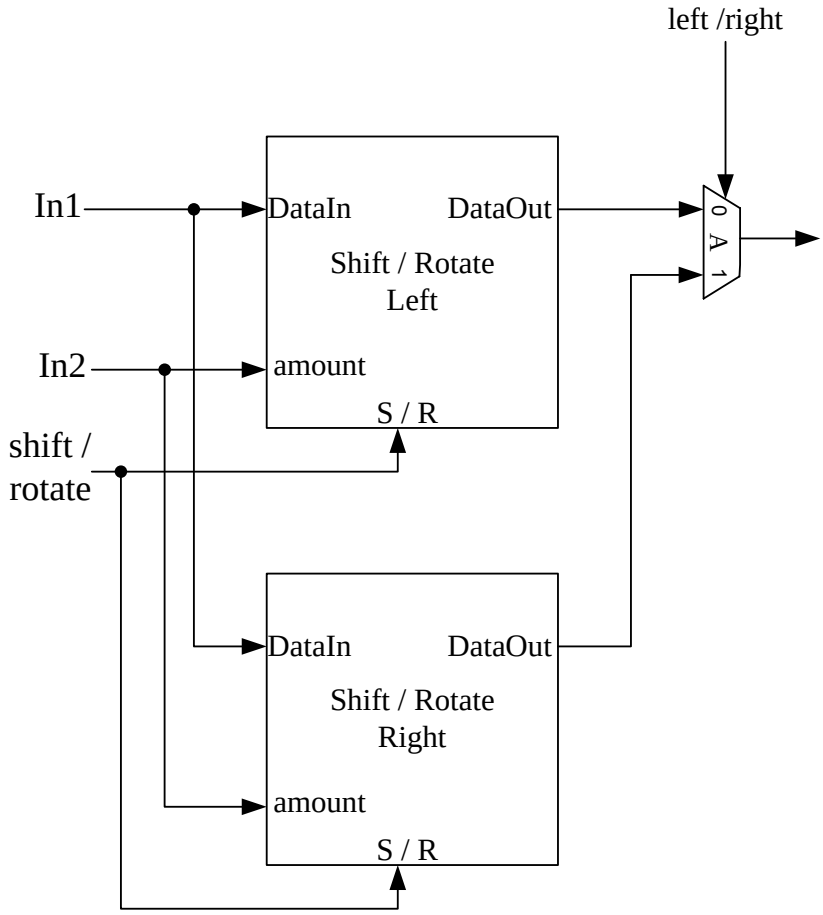


figure 9 – ALU core 2

In Virtex 4 FPGAs there is a new block called “XtremeDSP slice” that intergrades an 18x18 multiplier along with a 48x48 adder. Reference [16] has an application note, which was used, on how to form a 32x32 multiplier from these smaller ones; however its 32 MSBs were omitted, in order to perform modulo 2^{32} computations, as it is shown in figure 10.

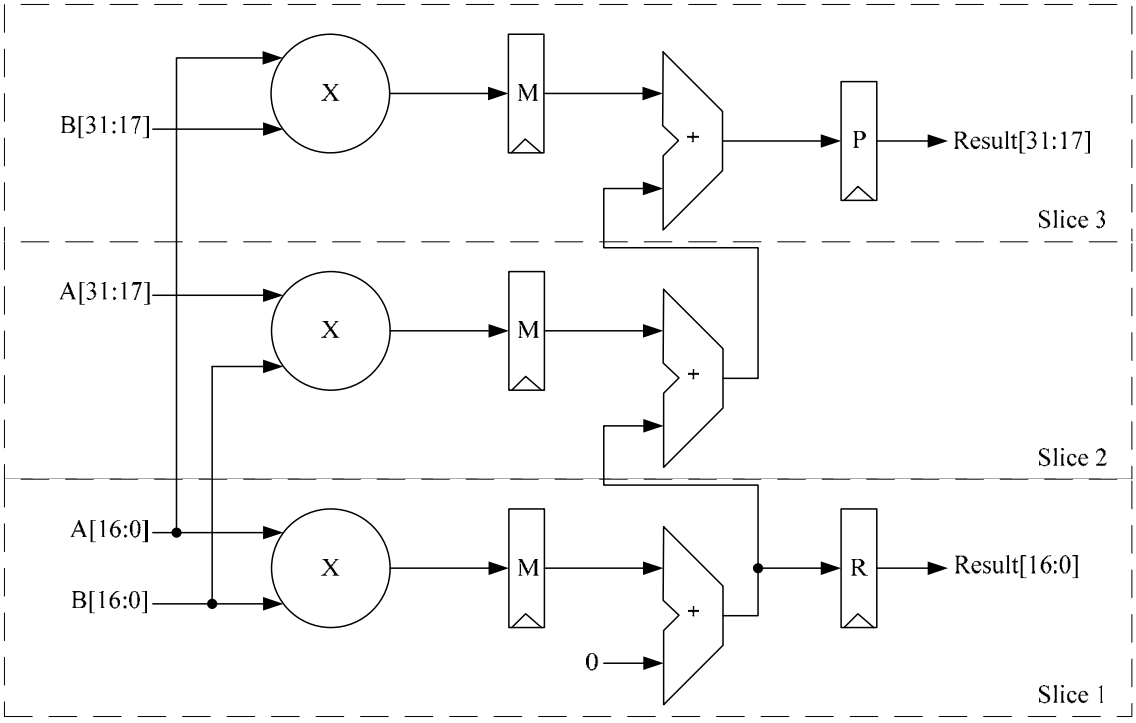


figure 10 – 32x32 multiplier modulo 2^{32}

As it was mentioned, in each slice there is an 18x18 multiplier, resulting in the utilization of three such units along with their respective adders. Slices 1 and 3 are used to produce the final result's bits, while slice 2 to compute an intermediate product. It should be also noted that every slice has, among others, a register between multiplier and adder called "M", plus one before each output called "P" and have been used to increase its maximum operating frequency. These registers in combination with the external "R" one, lead to a cost of 2 computing clock cycles per modulo multiplication.

Next stage is the memory stage. As it is shown in figure 11 all Sboxes have been placed to each cluster's memory stage, with the exception of Twofish, where a small portion is also in execution stage. Serpent also doesn't have its sboxes in every cluster because of its multiple instances.It should be noted that there is no reference to RC6 cipher, because it does not utilize any Sboxes at all.

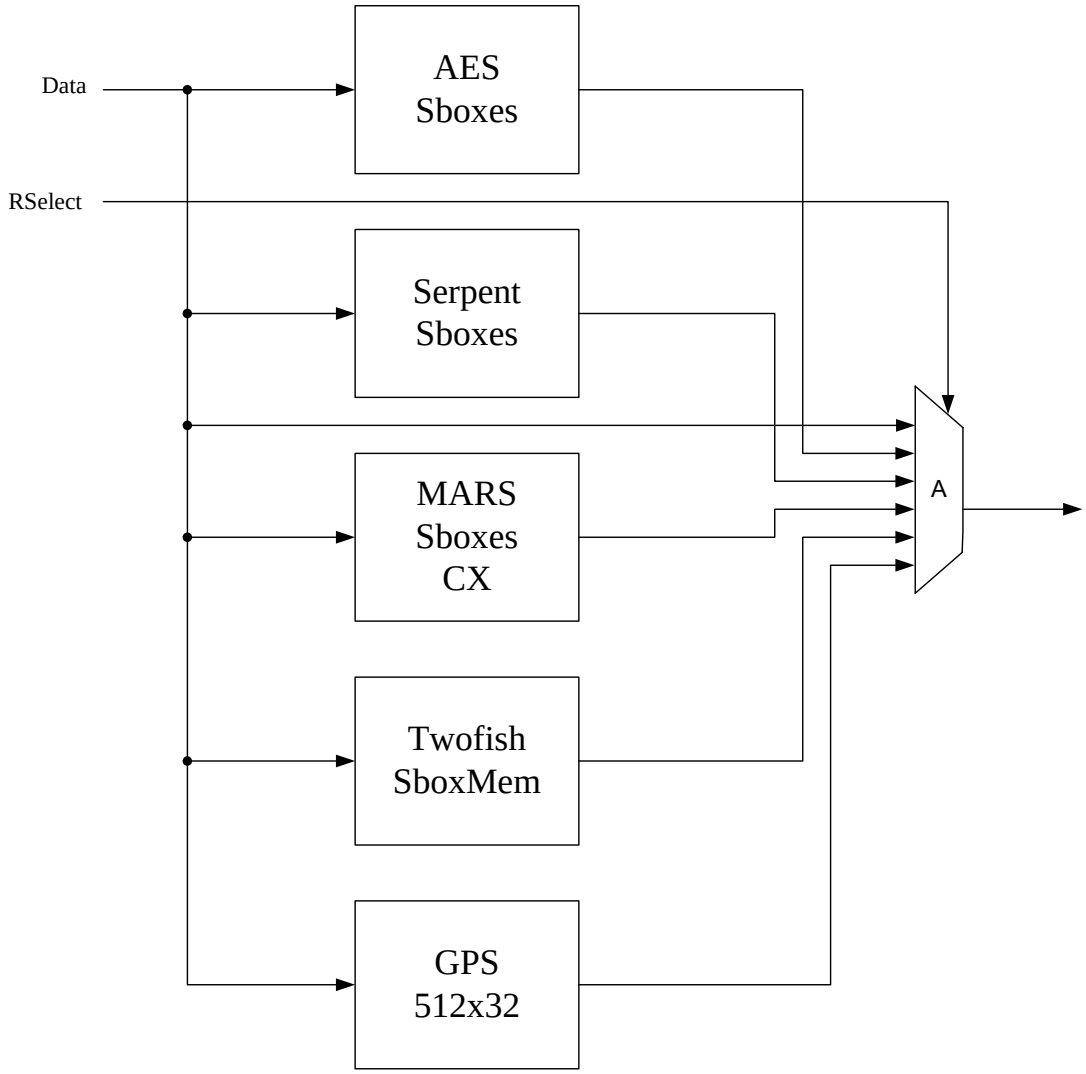


figure 11 – Sbox stage

Sboxes are usually non-linear structures that map an n-bit value to an m-bit value, essentially Look Up Tables (LUT). A symmetric cipher may have one or more different Sboxes, with each one of them having arbitrary dimensions, as shown in 12. Also the Sbox may even be the only non-linear part of the cipher.

Carefully chosen Sboxes can provide good resistance against linear and differential attacks, as well as good data and key bits avalanche. A drawback when using them is their relative slow software implementation. Also their index consists of a few bits (otherwise they would be too large), so they must

deliberately be placed in a cipher.

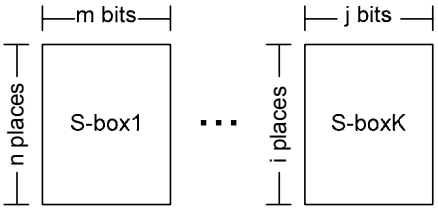


Figure 12 - Sboxes

Permutation is the structure where bits change place among each other, while in expansion, bits are also mixed but some of them appear more than once. They are linear operations, and thus not sufficient to guarantee security. However, when used with good non-linear Sboxes, they are vital for the security because they propagate the non-linearity uniformly over all bits.

2.6 Performance Evaluation on Xilinx Virtex 4 FPGA Devices

This section focuses on evaluating CCproc's performance while processing the AES round two finalists. Until now there is only a first prototype built on Virtex 4 FPGAs, which has been successfully verified in post-place and route simulation level. In order to evaluate its performance, first the total number of processing clock cycles needed for each cipher was measured and the results are shown in chart 1.

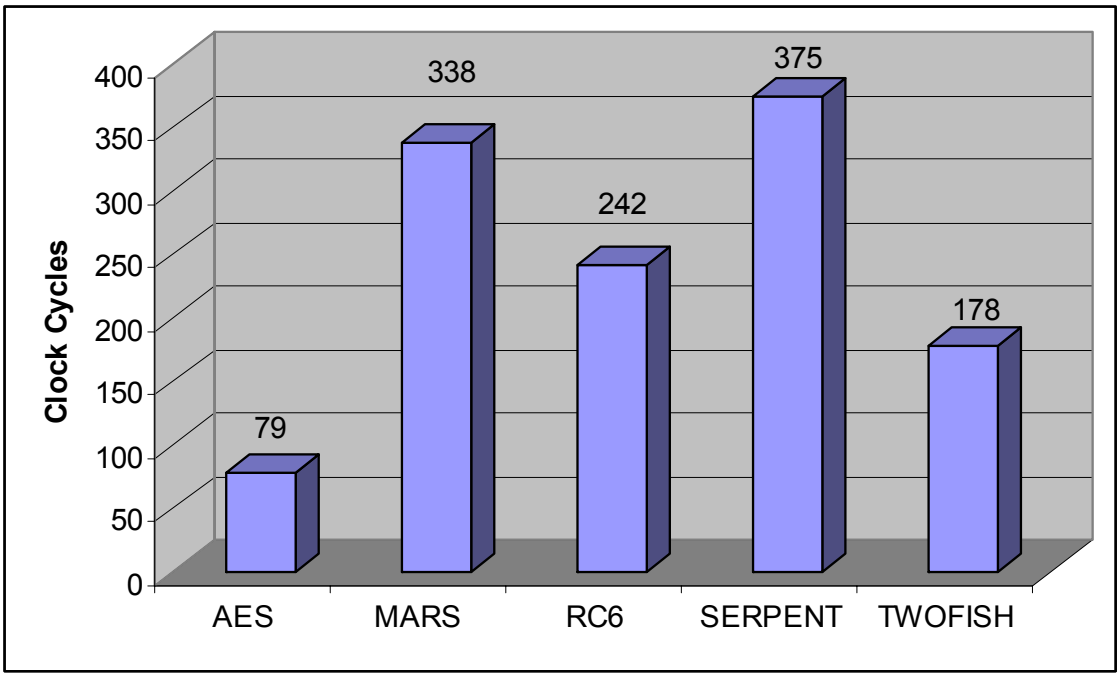


Chart 1 – CCproc's performance in clock cycles for the AES round two finalists

Xilinx XST (Xilinx Synthesis Tool) and ISE 7.1i reported the results shown in XY. The XC4VLX40 FPGA is the third smallest in the Virtex 4 series, a fact showing that CCproc is a compact design (275452 gates), capable to fit into today's smaller Virtex 4 FPGAs. The complete set consists of 1, 3 and 4-core implementations mapped on XC4VLX40, XC4VLX100, XC4VLX160 and XC4VLX 200 FPGAs. It should be noted that devices with speed grade equal to -12, create the fastest implementations.

FPGA	Speed Grade	CCproc Cores	Freq (MHz)	Utilization	Memory Blocks	Xtreme DSP
XC4VLX40	-12	1	108	95%	18	12
XC4VLX100	-12	1		36%	18	12
XC4VLX160	-12	3		77%	54	36
XC4VLX60	-11	1	95	19.6%	18	12
XC4VLX200	-11	4		78.6%	72	48

Table 2– CCproc's performance statistics

Based on the above performance results, chart 2 shows the achieved throughput for all AES round two finalists in ECB mode, in each multi-core

CCproc implementation. The formula that is used to extract results for 1-core implementations is the one below, where F is the design's operating frequency and cc are the processing clock cycles:

$$Throughput = \frac{128 \cdot F(Mhz)}{cc} Mbits/\sec$$

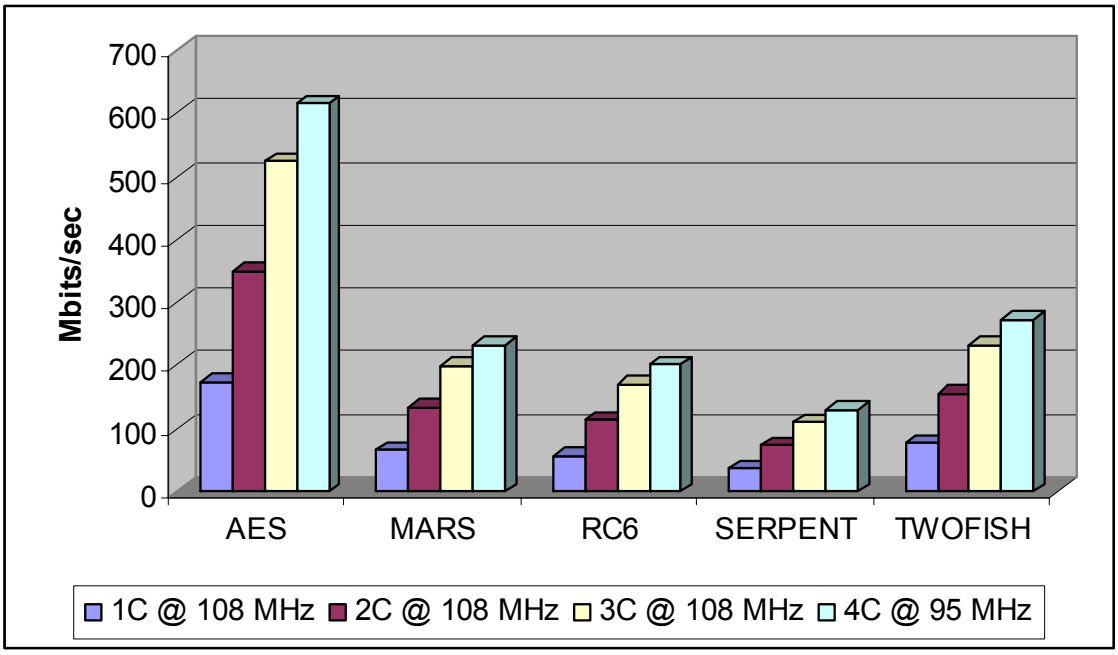


Chart 1 –CCproc Multi-core throughputs in ECB mode

Παραρτημα Β : Σύνθεση

In this chapter we analyze the synthesis methodology and the tools we used to accomplish this process. Synthesis is a blanket term which refers to the automatic translation of HDL code into an equivalent netlist of digital cells. Netlist is nothing but the structural logic description equivalent to the HDL input source. The source (HDL code) is the technology independent logic description written by the designer. The netlist contains the technology specific digital cells required for the actual design[I1]. The market leader in logic synthesis software is Synopsys.

3.1 Synopsys

For the procedure of synthesis we used the Synopsys Design Compiler[1]. The Design Compiler product is the core of the Synopsys synthesis software products. It comprises tools that synthesize our HDL designs into optimized technology-dependent, gate-level designs. It supports a wide range of flat and hierarchical design styles and can optimize both combinational and sequential designs for speed, area, and power.

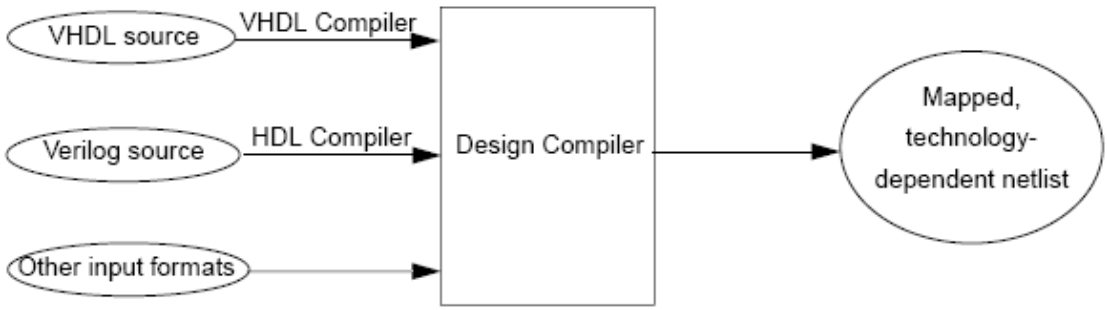


figure 13 - Design Compiler Synthesis Process Overview

Design Compiler reads and writes design files in all the standard electronic design automation (EDA) formats, including Synopsys internal database (.db) and equation (.eqn) formats. In addition, Design Compiler provides links to EDA tools, such as place and route tools, and to post-layout resynthesis

techniques, such as in-place optimization. These links enable information sharing, including forward-directed constraints and delays, between Design Compiler and external tools.

Design Compiler provides two user interfaces:

- The Design Compiler command-line interface, or shell, referred to as dc_shell. This interface supports both the Design Compiler shell language(dcsh) and an augmented tool command language (Tcl). To run.
- The Design Compiler graphical user interfaces (GUI), either the Design Analyzer or the Design Vision product.

3.2 Design Compiler Tools

Synopsys provides a spectrum of Design Compiler tools, which vary in complexity with the features offered. We choose the right tool for our design environment and synthesis requirements. Using Design Compiler tools, we can

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

3.3 Supported File Formats

Design compiler supports various design file formats (EDIF, VHDL, Verilog, db, sdf).The .db file format is the Synopsys database format and the .sdf is the standard delay format.

3.4 Resource Requirements

To compile 100K gates, Design Compiler requires at least 512 MB of RAM and 1 GB of swap space. In addition, approximately 1 GB of disk space is required for each 100K gates, and approximately 1 GB of disk space is needed for the Synopsys software

3.5 The High-Level Design Flow

In a basic high-level design flow, as we can see in figure 15, Design Compiler is used in both the design exploration stage and the final design implementation stage. In the exploratory stage, we use Design Compiler to carry out a preliminary, or default, synthesis. In the design implementation stage, we use the full power of Design Compiler to synthesize the design.

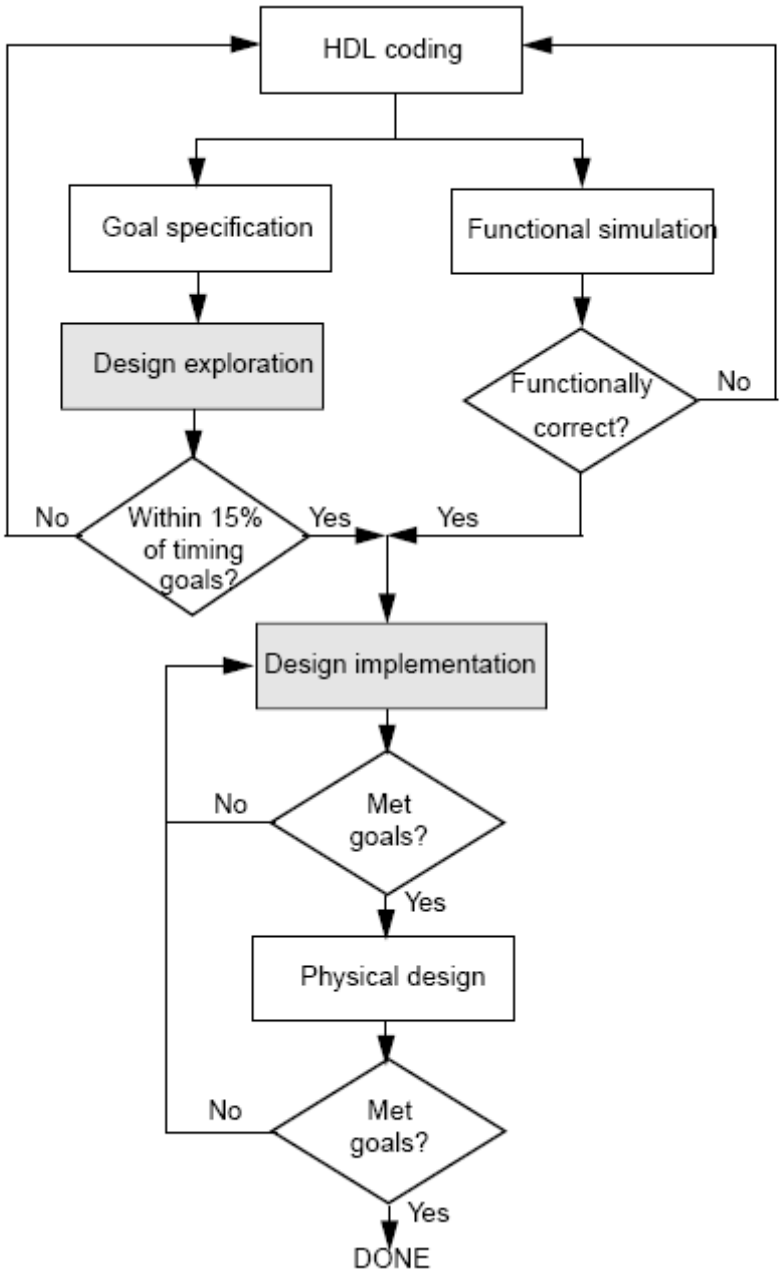


figure 14 - Basic High-Level Design Flow proposed by Synopsys

1. Start by writing an HDL description (Verilog or VHDL) of our design. We use good coding practices to facilitate successful Design Compiler synthesis of the design.
2. Perform design exploration and functional simulation in parallel.
    - In design exploration, we use Design Compiler to (a) implement specific design goals (design rules and optimization constraints) and (b) carry out a preliminary, “default” synthesis (using only the Design Compiler default options).
    - If design exploration fails to meet timing goals by more than 15 percent, we modify our design goals and constraints, or improve the HDL code. Then repeat both design exploration and functional simulation.
    - In functional simulation, determine whether the design performs the desired functions by using an appropriate simulation tool.
    - If the design does not function as required, we must modify the HDL code and repeat both design exploration and functional simulation.
    - We continue performing design exploration and functional simulation until the design is functioning correctly and is within 15 percent of the timing goals.

3. We perform design implementation synthesis by using Design Compiler to meet design goals. After synthesizing the design into a gate-level netlist, we verify that the design meets our goals. If the design does not meet our goals, we generate and analyze various reports to determine the techniques we might use to correct the problems.

4. After the design meets functionality, timing, and other design goals, we complete the physical design (either in-house or by sending it to our semiconductor vendor). We analyze the physical design’s performance by

using back-annotated data. If the results do not meet design goals, we return to step 3. If the results meet our design goals, we are finished with the design cycle.

3.6 Following the Basic Synthesis Flow

In figure 16 we see the basic synthesis flow. We can use this synthesis flow in both the design exploration and design implementation stages of the high-level design flow discussed previously. Also listed in figure are the basic dc_shell commands that are commonly used in each step of the basic flow.

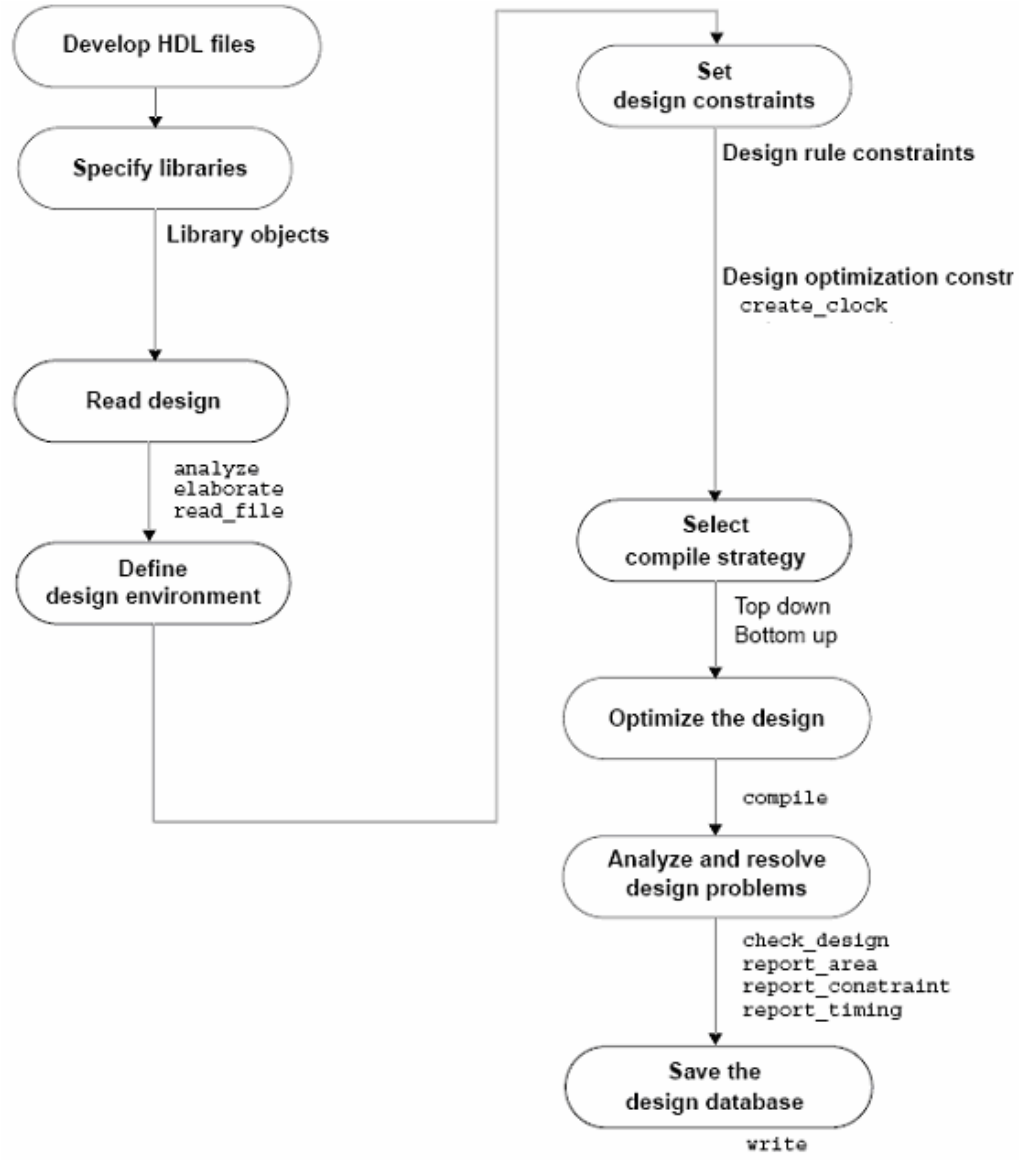


figure 15 - Basic Synthesis Flow

The basic synthesis flow consists of the following steps:

3.6.1 Develop HDL Files

The input design files for Design Compiler are often written using a hardware description language (HDL) such as Verilog or VHDL. These design descriptions need to be written carefully to achieve the best synthesis results possible. When writing HDL code, we need to consider design data management, design partitioning, and our HDL coding style. Partitioning and

coding style directly affect the synthesis and optimization processes.

HDL coding is the foundation for synthesis because it implies the initial structure of the design. When writing our HDL source code, we always have to consider the hardware implications of the code. A good coding style can generate smaller and faster designs. Writing technology-independent HDL is the first step to succeed our goal. The input HDL is parsed (also called as analysis) and translated (also called as elaboration) to a data structure. This data structure is converted into a network of generic logic cells. This network of generic logic cells is technology-independent since cell libraries in any technology normally contain NAND gates and inverters. The next thing we have to be careful when writing HDL source is that the Design Compiler will not accidentally infer elements as latches and flip-flops. Design Compiler can infer a generic multiplexer cell from case statements in our HDL code. VHDL Compiler infers a D latch whenever we do not specify the resulting value for an output under all conditions, as in an incompletely specified if or case statement. When an if statement used in a Verilog always block or VHDL process as part of a continuous assignment does not include an else clause, Design Compiler creates a latch. The following examples show if statements that generate latches during synthesis.

Example Incorrect if Statement (Verilog)

```
if ((a == 1) && (b == 1))
z = 1;
```

Example Incorrect if Statement (VHDL)

```
if (a = '1' and b = '1') then
z <= '1';
end if;
```

An incomplete case statement results in the creation of a latch. VHDL does not support incomplete case statements. In Verilog we can avoid latch inference by using either the default clause or the full_case compiler directive. Although both the full_case directive and the default clause prevent latch inference, they have different meanings. The full_case directive asserts that

all valid input values have been specified and no default clause is necessary. The default clause specifies the output for any undefined input values. For best results, we use the default clause instead of the full_case directive. If the unspecified input values are don't care conditions, using the default clause with an output value of x can generate a smaller implementation. If we use the full_case directive, the gate-level simulation might not match the RTL simulation whenever the case expression evaluates to an unspecified input value. If we use the default clause, simulation mismatches can occur only if we specified don't care conditions and the case expression evaluates to an unspecified input value.

HDL Compiler can also infer SR latches and master-slave latches. HDL Compiler infers a D flip-flop whenever the sensitivity list of a Verilog always block or VHDL process includes an edge expression (a test for the rising or falling edge of a signal). HDL Compiler can also infer JK flip-flops and toggle flip-flops. We have to be careful when mixing rising- and falling-edge-triggered flip-flops in our design. If a module infers both rising- and falling-edge-triggered flip-flops and the target technology library does not contain a falling-edge-triggered flip-flop, Design Compiler generates an inverter in the clock tree for the falling-edge clock.

When writing HDL we have to avoid Inferring Registers Without Control Signals[4]. For inferring registers without control signals, make the data and clock pins controllable from the input ports or through combinational logic. If a gate-level simulator cannot control the data or clock pins from the input ports or through combinational logic, the simulator cannot initialize the circuit, and the simulation fails. We have to be cautious when Inferring Three-State Drivers**.** We assign the high-impedance value (1'bz in Verilog, 'Z' in VHDL) to the output pin to have Design Compiler infer three-state gates. Three-state logic reduces the testability of the design and makes debugging difficult. Where possible, we replace three-state buffers with a multiplexer. We never use high-impedance values in a conditional expression. HDL Compiler always evaluates expressions compared to high-impedance values as false, which can cause the gate-level implementation to behave differently from the RTL

description.

3.6.2 Specify Libraries

We specify the link, target, symbol, and synthetic libraries for Design Compiler by using the link_, target_, symbol_, and synthetic_library commands. The link and target libraries are technology libraries that define the semiconductor vendor's set of cells and related information, such as cell names, cell pin names, delay arcs, pin loading, design rules, and operating conditions.

3.6.3 Read Design

Design Compiler uses HDL Compiler to read both RTL designs and gate-level netlists as design file input. We use the analyze and elaborate commands to read RTL designs, and we use the read_file command (or read command) to read gate-level netlists. Design Compiler supports all the principal gate-level netlist formats.

3.6.4 Define Design Environment

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

In Design Compiler, the model is defined by a set of attributes and constraints that we assign to the design, using specific dc_shell commands. Figure 17 illustrates the commands used to define the design environment.

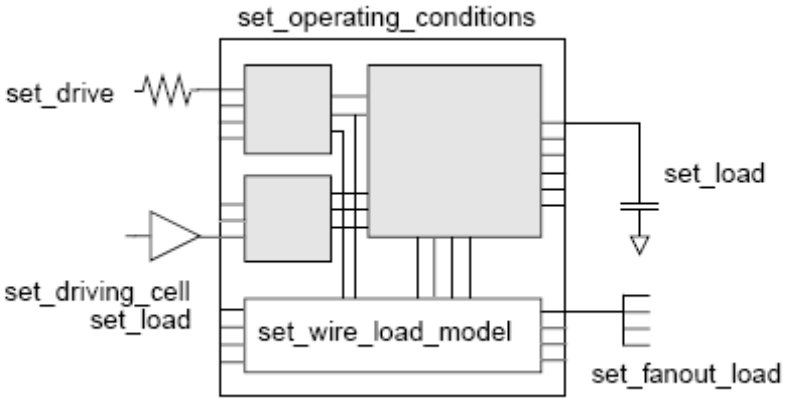


figure 16 - Commands Used to Define the Design Environment

In most technologies, variations in operating temperature, supply voltage, and manufacturing process can strongly affect circuit performance (speed). These factors, called operating conditions, have the following general characteristics:

- Operating temperature variation

  Temperature variation is unavoidable in the everyday operation of a design. Effects on performance caused by temperature fluctuations are most often handled as linear scaling effects, but some submicron silicon processes require nonlinear calculations.

- Supply voltage variation

  The design's supply voltage can vary from the established ideal value during day-to-day operation. Often a complex calculation (using a shift in threshold voltages) is employed, but a simple linear scaling factor is also used for logic-level performance calculations.

- Process variation

  This variation accounts for deviations in the semiconductor fabrication process. Usually process variation is treated as a percentage variation in the performance calculation.

When performing timing analysis, Design Compiler must consider the worst-case and best-case scenarios for the expected variations in the process, temperature, and voltage factors.

If the technology library contains operating condition specifications, we can let Design Compiler use them as default conditions. Alternatively, we can use the set_operating_conditions command to specify explicit operating conditions, which supersede the default library conditions.

One of the most important things we have to define is Wire Load Models**.** Wire load modeling allows us to estimate the effect of wire length and fanout on the resistance, capacitance, and area of nets. Design Compiler uses these physical values to calculate wire delays and circuit speeds. Semiconductor vendors develop wire load models, based on statistical information specific to the vendors' process. The models include coefficients for area, capacitance, and resistance per unit length, and a fanout-to-length table for estimating net lengths (the number of fanouts determines a nominal length). In the absence of back-annotated wire delays, Design Compiler uses the wire load models to estimate net wire lengths and delays. Design Compiler determines which wire load model to use for a design, based on the following factors, listed in order of precedence:

1. Explicit user specification
2. Automatic selection based on design area
3. Default specification in the technology library

If none of this information exists, Design Compiler does not use a wire load model. Without a wire load model, Design Compiler does not have complete information about the behavior of our target technology and cannot compute loading or propagation times for our nets; therefore, our timing information will be optimistic. In hierarchical designs, Design Compiler must also determine which wire load model to use for nets that cross hierarchical boundaries. The tool determines the wire load model for cross-hierarchy nets
based on one of the following factors, listed in order of precedence:

1. Explicit user specification
2. Default specification in the technology library
3. Default mode in Design Compiler

Design Compiler supports three modes for determining which wire load model to use for nets that cross hierarchical boundaries:

- Top

  Design Compiler models nets as if the design has no hierarchy and uses the wire load model specified for the top level of the design hierarchy for all nets in a design and its subdesigns. The tool ignores any wire load models set on subdesigns with the set_wire_load_model command. We use top mode if we plan to flatten the design at a higher level of hierarchy before layout.

- Enclosed

  Design Compiler uses the wire load model of the smallest design that fully encloses the net. If the design enclosing the net has no wire load model, the tool traverses the design hierarchy upward until it finds a wire load model. Enclosed mode is more accurate than top mode when cells in the same design are placed in a contiguous region during layout. We use enclosed mode if the design has similar logical and physical hierarchies.

- Segmented

  Design Compiler determines the wire load model of each segment of a net by the design encompassing the segment. Nets crossing hierarchical boundaries are divided into segments. For each net segment, Design Compiler uses the wire load model of the design containing the segment. If the design contains a segment that has no wire load model, the tool traverses the design hierarchy upward until it finds a wire load model. We use segmented mode if the wire load models in our technology have been characterized with net segments.

In figure 18 we see a sample design with a cross-hierarchy net, cross_net. The top level of the hierarchy (design TOP) has a wire load model of 50x50. The next level of hierarchy (design MID) has a wire load model of 40x40. The leaf-level designs, A and B, have wire load models of 20x20 and 30x30, respectively.

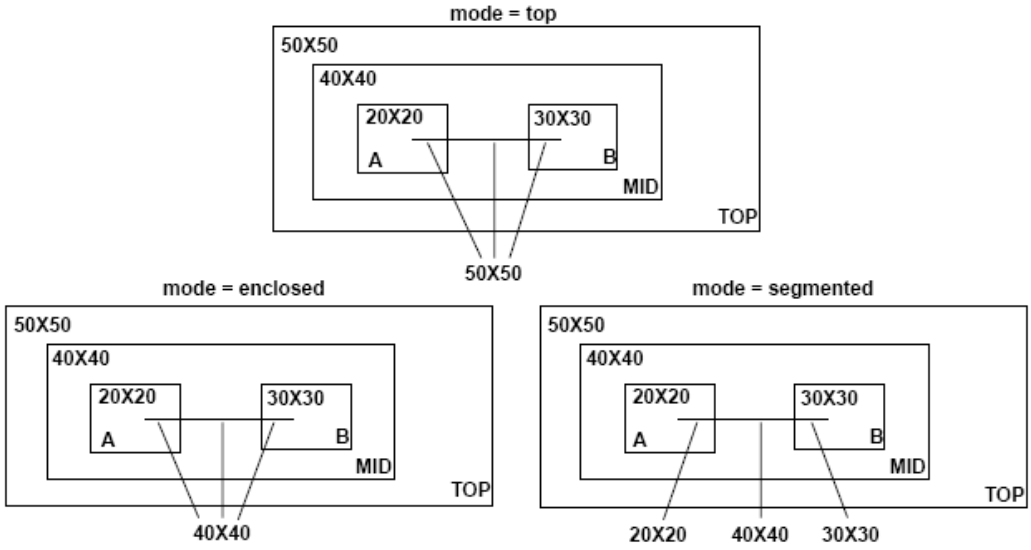


figure 17 - Comparison of Wire Load Mode

In top mode, Design Compiler estimates the wire length of net cross_net, using the 50x50 wire load model. Design Compiler ignores the wire load models on designs MID, A, and B.

In enclosed mode, Design Compiler estimates the wire length of net cross_net, using the 40x40 wire load model (the net cross_net is completely enclosed by design MID).

In segmented mode, Design Compiler uses the 20x20 wire load model for the net segment enclosed in design A, the 30x30 wire load model for the net segment enclosed in design B, and the 40x40 wire load model for the segment enclosed in design MID.

The technology library can define a default wire load model that is used for all designs implemented in that technology. The default_wire_load library attribute identifies the default wire load model for a technology library. Some libraries support automatic area-based wire load selection. Design Compiler uses the library function wire_load_selection to choose a wire load model based on the total cell area. For large designs with many levels of hierarchy, automatic wire load selection can increase runtime. To manage runtime, set

the wire load manually.

After specifying wire lode models and modes we have to model our systems interface. Design Compiler supports the following ways to model the design's interaction with the external system:

- Defining drive characteristics for input ports

Design Compiler uses drive strength information to buffer nets appropriately in the case of a weak driver. Drive strength is the reciprocal of the output driver resistance, and the transition time delay at an input port is the product of the drive resistance and the capacitance load of the input port.We use set_driving_cell and set_input_transition commands wich affect the port transition delay, but they do not place design rule requirements, such as max_fanout and max_transition, on input ports. However, the set_driving_cell command does place design rules on input ports if the driving cell has DRCs. Both the set_drive and the set_driving_cell commands affect the port transition delay. The set_driving_cell command can place design rule requirements, such as max_fanout or max_transition, on input ports if the specified cell has input ports.

- Defining loads on input and output ports

By default, Design Compiler assumes zero capacitive load on input and output ports. We use the set_load command to set a capacitive load value on input and output ports of the design. This information helps Design Compiler select the appropriate cell drive strength of an output pad and helps model the transition delay on input pads.

- Defining fanout loads on output ports

We can model the external fanout effects by specifying the expected fanout load values on output ports with the set_fanout_load command. Fanout load is not the same as load. Fanout load is a unitless value that represents a numerical contribution to the total fanout. Load is a capacitance value. Design Compiler uses fanout load primarily to measure

the fanout presented by each input pin. An input pin normally has a fanout load of 1, but it can have a higher value.

3.6.5 Set Design Constraints

Design Compiler uses design rules and optimization constraints to control the synthesis of the design. Design rules are provided in the vendor technology library to ensure that the product meets specifications and works as intended. Typical design rules constrain transition times (set_max_transition), fanout loads (set_max_fanout), and capacitances (set_max_capacitance). These rules specify technology requirements that we cannot violate. (we can, however, specify stricter constraints.) Optimization constraints define the design goals for timing (clocks, clock skews, input delays, and output delays) and area (maximum area). In the optimization process, Design Compiler attempts to meet these goals, but no design rules are violated by the process. To optimize a design correctly, we must set realistic constraints.

Design rule constraints are attributes specified in the technology library and, optionally, specified by us explicitly. If a technology library defines these attributes, Design Compiler implicitly applies them to any design using that library when it compiles the design or creates a constraint report. We cannot remove the design rule attributes defined in the technology library, because they are requirements for the technology, but we can make them more restrictive to suit our design. The transition time of a net is the time required for its driving pin to change logic values. This transition time is based on the technology library data. For the nonlinear delay model (NLDM), output transition time is a function of input transition and output load. Design Compiler and Library Compiler model transition time restrictions by associating a max_transition attribute with each output pin on a cell. During optimization, Design Compiler attempts to make the transition time of each net less than the value of the max_transition attribute. To change the maximum transition time restriction specified in a technology library, we use the set_max_transition command. This command sets a maximum transition

time for the nets attached to the identified ports or to all the nets in a design by setting the max_transition attribute on the named objects.

The maximum fanout load for a net is the maximum number of loads the net can drive. Design Compiler and Library Compiler model fanout restrictions by associating a fanout_load attribute with each input pin and a max_fanout attribute with each output (driving) pin on a cell. The fanout load value does not represent capacitance; it represents the weighted numerical contribution to the total fanout load. The fanout load imposed by an input pin is not necessarily 1.0. Library developers can assign higher fanout load values to model internal cell fanout effects. Design Compiler calculates the fanout of a driving pin by adding the fanout_load values of all inputs driven by that pin. To determine whether the pin meets the maximum fanout load restriction, Design Compiler compares the calculated fanout load value with the pin's max_fanout value. During optimization, Design Compiler attempts to meet the fanout load restrictions for each driving pin. If a pin violates its fanout load restriction, Design Compiler tries to correct the problem (for example, by changing the drive strength of the component). The technology library might specify default fanout constraints on the entire library or fanout constraints for specific pins in the library description of an individual cell. To set a more conservative fanout restriction than that specified in the technology library, we use the set_max_fanout command on the design or on an input port. (we use the set_fanout_load command to set the expected fanout load value for output ports.) If we use the set_max_fanout command and a library max_fanout attribute exists, Design Compiler tries to meet the smaller (more restrictive) fanout limit.

The transition time constraints do not provide a direct way to control the actual capacitance of nets. If we need to control capacitance directly, we use the set_max_capacitance command to set the maximum capacitance constraint. This constraint is completely independent, so we can use it in addition to the transition time constraints. Design Compiler and Library Compiler model capacitance restrictions by associating the max_capacitance attribute with the output ports or pins of a cell. Design Compiler calculates the capacitance on

the output net by adding the wire capacitance of the net to the capacitance of the pins attached to the net. To determine whether the net meets the capacitance constraint, Design Compiler compares the calculated capacitance value with the output pin's max_capacitance value. We can also use the set_min_capacitance command to define the minimum capacitance for input ports or pins. Design Compiler attempts to ensure that the load seen at the input port does not fall below the specified capacitance value, but it does not specifically optimize for this constraint.

After setting design rules we have to set the optimization constraints. The most commonly specified optimization Constraints are timing constraints and area constraints.

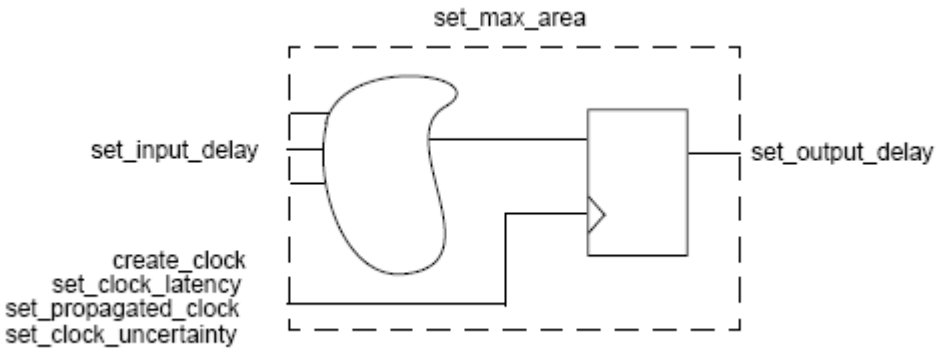


figure 18 - Commands Used to Define the Optimization Constraints

Timing constraints specify the required performance of the design.
To set the timing constraints we :

1. Define the clocks.

   For synchronous designs, the clock period is the most important constraint because it constrains all register-to-register paths in the design. We use the create_clock command to define the period (-period option) and waveform (-waveform option) for the clock. If we do not specify the clock waveform, Design Compiler uses a 50 percent duty cycle. In some cases,

a system clock might not exist in a block. We can use the create_clock -name command to create a virtual clock for modeling clock signals present in the system but not in the block. By creating a virtual clock, we can represent delays that are relative to clocks outside the block. By default, Design Compiler assumes that clock networks have no delay (ideal clocks). We use the set_clock_latency and set_clock_uncertainty commands to specify timing information about the clock network delay. We can use these commands to specify either estimated or actual delay information. We use the set_propagated_clock command to specify that we want the clock latency to propagate through the clock network. We use the -setup or -hold options of the set_clock_latency command to add some margin of error into the system to account for variances in the clock network resulting from layout.

2. Specify the I/O timing requirements relative to the clocks.

If we do not assign timing requirements to an input port, Design Compiler responds as if the signal arrives at the input port at time 0. In most cases, input signals arrive at staggered times. We use the set_input_delay command to define the arrival times for input ports. We define the input delay constraint relative to the system clock and to the other inputs. If we do not assign timing requirements to an output port, Design Compiler does not constrain any paths which end at an output port. We use the set_output_delay command to define the required output arrival time. We define the output delay constraint relative to the system clock. If an input or output port has multiple timing requirements (because of multiple paths), we use the -add_delay option to specify the additional timing requirements.

3. Specify the combinational path delay requirements.

For purely combinational delays that are not bounded by a clock period, we use the set_max_delay and set_min_delay commands to define the maximum and minimum delays for the specified paths. A common way to

produce this type of asynchronous logic in HDL code is to use asynchronous sets or resets on latches and flip-flops. Because the reset signal crosses several blocks, constrain this signal at the top level.

4. Specify the timing exceptions.

Timing exceptions define timing relationships that override the default single-cycle timing relationship for one or more timing paths. We use timing exceptions to constrain or disable asynchronous paths or paths that do not follow the default single-cycle behavior. Design Compiler recognizes only timing exceptions that have valid reference points. The valid startpoints in a design are the primary input ports and the clock pins of sequential cells. The valid endpoints are the primary output ports of a design and the data pins of sequential cells. We can specify the following conditions by using timing exception commands: False paths (set_false_path), minimum delay requirements (set_min_delay), maximum delay requirements (set_max_delay), multicycle paths (set_multicycle_path). Design Compiler does not report false paths in the timing report or consider them during timingoptimization. We use the set_false_path command to specify a false path. We use this command to ignore paths that are not timing critical, that can mask other paths that must be considered during optimization, or that never occur in normal operation. We can use the set_min_delay and set_max_delay commands to specify path delay requirements that are more conservative than those derived by Design Compiler based on the clock timing. The multicycle path condition is appropriate when the path in question is longer than a single cycle or when data is not expected within a single cycle. We use the set_multicycle_path command to specify the number of clock cycles Design Compiler should use to determine when data is required at a particular endpoint.

After timing constraints we have to specify the area constraints. The set_max_area command specifies the maximum area for the current design by placing a max_area attribute on the current design. Specify the area in the same units used for area in the technology library.

3.6.6 Select Compile Strategy

The two basic compile strategies that we can use to optimize hierarchical designs are referred to as top down and bottom up[3]. In the top-down strategy, the top-level design and all its subdesigns are compiled together. All environment and constraint settings are defined with respect to the top-level design. Although this strategy automatically takes care of interblock dependencies, the method is not practical for large designs because all designs must reside in memory at the same time. In the bottom-up strategy, individual subdesigns are constrained and compiled separately. After successful compilation, the designs are assigned the dont_touch attribute to prevent further changes to them during subsequent compile phases. Then the compiled subdesigns are assembled to compose the designs of the next higher level of the hierarchy (any higher-level design can also incorporate unmapped logic), and these designs are compiled. This compilation process is continued up through the hierarchy until the top-level design is synthesized. This method lets us compile large designs because Design Compiler does not need to load all the uncompiled subdesigns into memory at the same time. At each stage, however, we must estimate the interblock constraints, and typically we must iterate the compilations, improving these estimates, until all subdesign interfaces are stable. Each strategy has its advantages and disadvantages, depending on our particular designs and design goals. We can use either strategy to process the entire design, or we can mix strategies, using the most appropriate strategy for each subdesign.

The top-down compile strategy has these advantages:

- Provides a push-button approach
- Takes care of interblock dependencies automatically

On the other hand, the top-down compile strategy requires more memory and

might result in longer runtimes for designs with over 100K gates. To implement a top-down compile, carry out the following steps:

1. Read in the entire design.
2. Resolve multiple instances of any design references. A design that is referenced by more than one instantiated block or cell must be resolved. Otherwise, Design Compiler cannot compute which environmental attributes and constraints to apply to the design during optimization.
3. Apply attributes and constraints to the top level. Attributes and constraints implement the design specification.
4. Compile the design.

There goes an example of a top-down compile strategy :

```
/* read in the entire design */
read -f verilog E.v
read -f verilog D.v
read -f verilog C.v
read -f verilog B.v
read -f verilog A.v
read -f verilog TOP.v
current_design TOP
link
/* resolve multiple references */
uniquify
/* apply constraints and attributes */
include defaults.con
/* compile the design */
compile
```

The bottom-up compile strategy provides these advantages:

- Compiles large designs by using the divide-and-conquer approach
- Requires less memory than top-down compile
- Allows time budgeting The bottom-up compile strategy requires
- Iterating until the interfaces are stable
- Manual revision control

The bottom-up compile strategy compiles the subdesigns separately and then incorporates them in the top-level design. The top-level constraints are applied, and the design is checked for violations. Although it is possible that no violations are present, this outcome is unlikely because the interface settings between subdesigns usually are not sufficiently accurate at the start.
To improve the accuracy of the interblock constraints, we read in the top-level design and all compiled subdesigns and apply the characterize command to the individual cell instances of the subdesigns. Based on the more realistic environment provided by the compiled subdesigns, characterize captures environment and timing information for each cell instance and then replaces the existing attributes and constraints of each cell's referenced subdesign with the new values. Using the improved interblock constraint, we recompile the characterized subdesigns and again check the top-level design for constraint violations. We should see improved results, but we might need to iterate the entire process several times to remove all significant violations.

The bottom-up compile strategy requires these steps:

1. Develop both a default constraint file and subdesign-specific constraint files. The default constraint file includes global constraints, such as the clock information and the drive and load estimates. The subdesign-specific constraint files reflect the time budget allocated to the subblocks.

2. Compile the subdesigns independently.

3. Read in the top-level design and any compiled subdesigns not

already in memory.

4. Set the current design to the top-level design, link the design, and apply the top-level constraints. If the design meets its constraints, we are finished. Otherwise, we continue with the following steps.

5. Apply the characterize command to the cell instance with the worst violations.

6. Use write_script to save the characterized information for the cell.

7. Use remove_design -all to remove all designs from memory.

8. Read in the RTL design of the previously characterized cell.

9. Set current_design to the characterized cell's subdesign and recompile, using the saved script of characterization data.

10. Read in all other compiled subdesigns.

11. Link the current subdesign.

12. Choose another subdesign, and repeat steps 3 through 9 until we have recompiled all subdesigns, using their actual environments.

When applying the bottom-up compile strategy, consider the following:

- The read_file command runs most quickly with the .db format. If we will not be modifying our RTL code after the first time we read (or elaborate) it, save the unmapped design to a .db file. This will save time when we reread the design.
- The compile command affects all subdesigns of the current design. If we want to optimize only the current design, we can remove or not include

its subdesigns in our database, or we can place the dont_touch attribute on the subdesigns (by using the set_dont_touch command).

There goes an example of a down top compile strategy :

```
all_blocks = {E,D,C,B,A}
/* compile each subblock independently */
foreach (block, all_blocks) {
/* read in block */
block_source = block + ".v"
read_file -format verilog block_source
current_design block
link
uniquify
/* apply global attributes and constraints */
include defaults.con
/* apply block attributes and constraints */
block_script = block + ".con"
include block_script
/* compile the block */
compile
}
/* read in entire compiled design */
read_file -format verilog TOP.v
current_design TOP
link
write -hierarchy -output first_pass.db
/* apply top-level constraints */
include defaults.con
include top_level.con
/* check for violations */
report_constraint
/* characterize all instances in the design */
all_instances = {U1,U2,U2/U3,U2/U4,U2/U5}
characterize -constraint all_instances
/* save characterize information */
foreach (block, all_blocks) {
current_design block
char_block_script = block + ".wscr"
```

```
write_script > char_block_script
}
/* recompile each block */
foreach (block, all_blocks) {
/* clear memory */
remove_design -all
/* read in previously characterized subblock */
block_source = block + ".v"
read -format verilog block_source
/* recompile subblock */
current_design block
link
uniquify
/* apply global attributes and constraints */
include defaults.con
/* apply characterization constraints */
char_block_script = block + ".wscr"
include char_block_script
/* apply block attributes and constraints */
block_script = block + ".con"
include block_script
/* recompile the block */
compile
}
```

We can take advantage of the benefits of both the top-down and the bottom-up compile strategies by using both. This compile strategy is called mixed compile strategy.

- We use the top-down compile strategy for small hierarchies of blocks.
- We use the bottom-up compile strategy to tie small hierarchies together into larger blocks.

3.6.7 Optimize the Design

We use the compile command to invoke the Design Compiler synthesis and optimization processes. Several compile options are available. In particular, the map_effort option can be set to low, medium, or high. In a preliminary compile, when we want to get a quick idea of design area and performance, we set map_effort to low. In a default compile, when we are performing design exploration, we use the medium map_effort option. Because this option is the default, we do not need to specify map_effort in the compile command. In a final design implementation compile, we might want to set map_effort to high. We should use this option judiciously, however, because the resulting compile process is CPU intensive. Often setting map_effort to medium is sufficient.

A fully optimized design is one, which has met the timing requirements and occupies the smallest area. The optimization can be done in two stages one at the code level, the other during synthesis. The optimization at the code level involves modifications to RTL code that is already been simulated and tested for its functionality. This level of modifications to the RTL code is generally avoided as sometimes it leads to inconsistencies between simulation results before and after modifications. However, there are certain standard model optimization techniques that might lead to a better synthesized design. Model optimizations are important to a certain level, as the logic that is generated by the synthesis tool is sensitive to the RTL code that is provided as input. Different RTL codes generate different logic. Minor changes in the model might result in an increase or decrease in the number of synthesized gates and also change its timing characteristics. A logic optimizer reaches different endpoints for best area and best speed depending on the starting point provided by a netlist synthesized from the RTL code. The different starting points are obtained by rewriting the same HDL model using different constructs. Some of the optimizations, which can be used to modify the model for obtaining a better quality design, are listed below[2].

- Resource Allocation.

This method refers to the process of sharing a hardware resource under mutuallyexclusive conditions. Consider the following *if* statement.

if A = ‘1’ then

E = B + C;

else

E = B + D;

end if;

The above code would generate two ALUs one for the addition of B+C and other for the addition B + D which are executed under mutually exclusive conditions. Therefore a single ALU can be shared for both the additions. The hardware synthesized for the above code is given below in figure 19.

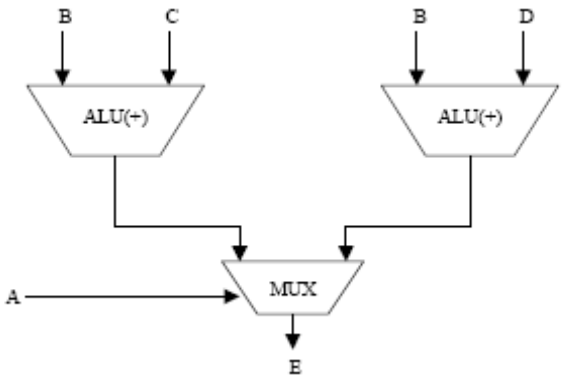


figure 19 - Without resource allocation.

The above code is rewritten with only on addition operator being employed. The hardware synthesized is given in figure 20.

if A = ‘1’ then

temp := C; // A temporary variable introduced.

else

temp := D;

end if;

E = B + temp;

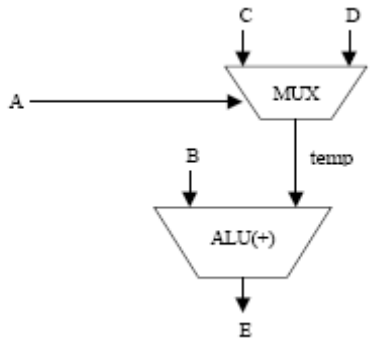


figure 20 - With resource allocation.

It is clear from the figure 20 that one ALU has been removed with one ALU being shared for both the addition operations. However a multiplexer is induced at the inputs of the ALU that contributes to the path delay. Earlier the timing path of the select signal goes through the multiplexer alone, but after resource sharing it goes through the multiplexer and the ALU datapath, increasing its path delay. However due to resource sharing the area of the design has decreased. This is therefore a trade-off that the designer may have to make. If the design is timing-critical it would be better if no resource sharing is performed.

- Common sub-expressions and Common factoring

It is often useful to identify common subexpressions and to reuse the computed values wherever possible. A simple example is given below.

```
B := R1 + R2;
…..
C <= R3 – (R1 + R2);
```

Here the subexpression R1 + R2 in the signal assignment for C can be replaced by B as given below. This might generate only one adder for the computation instead of two.

```
C <= R3 – B;
```

Common factoring is the extraction of common subexpressions in mutually-exclusive branches of an *if* or *case* statement.

```
if (test)
A <= B & (C + D);
else
J <= (C + D) | T;
end if;
```

In the above code the common factor C + D can be place out of the *if* statement, which might result in the tool generating only one adder instead of two as in the above case.

```
temp := C + D; // A temporary variable introduced.
if (test)
A <= B & temp;
else
J <= temp | T;
end if;
```

Such minor changes if made by the designer can cause the tool to synthesize better logic and also enable it to concentrate on optimizing more critical areas.

- Moving Code

In certain cases an expression might be placed, within a for/while loop statement, whose value would not change through every iteration of the loop. Typically a synthesis tool handles the a for/while loop statement by unrolling it the specified number of times. In such cases redundant code might be generated for that particular expression causing additional logic to be synthesized. This could be avoided if the expression is moved outside the loop, thus optimizing the design. Such optimizations performed at a higher level, that is, within the model, would help the optimizer to concentrate on more critical pieces of the code. An example is given below.

```
C := A + B;
……
for c in range 0 to 5 loop
……
```

T := C – 6;

// Assumption : C is not assigned a new value within the loop, thus the above expression would remain constant on every iteration of the loop.

……

end loop;

The above code would generate six subtracters for the expression when only one is necessary. Thus by modifying the code as given below we could avoid the generation of unnecessary logic.

C := A + B;

……

temp := C – 6; // A temporary variable is introduced

for c in range 0 to 5 loop

……

T := temp;

// Assumption : C is not assigned a new value within the loop, thus the above expression would remain constant on every iteration of the loop.

……

end loop;

- Constant folding and Dead code elimination

The are possibilities where the designer might leave certain expressions which are constant in value. This can be avoided by computing the expressions instead of the implementing the logic and then allowing the logic optimizer to eliminate the additional logic.

Ex:

C := 4;

….

Y = 2 * C;

Computing the value of Y as 8 and assigning it directly within our code can avoid the above unnecessary code. This method is called constant folding. The other optimization, dead code elimination refers to those sections of code, which are

never executed.

Ex.

```
A := 2;
B := 4;
if(A > B) then
……
end if;
```

The above *if* statement would never be executed and thus should be eliminated from the code. The logic optimizer performs these optimizations by itself, but nevertheless if the designer optimizes the code accordingly the tool optimization time would be reduced
resulting in faster tool running times.

- Flip-flop and Latch optimizations

Earlier in the develop HDL files section, it has been described how flip-flops and latches are inferred through the code by the synthesis tool. However there are only certain cases where the inference of the above two elements is necessary. The designer thus should try to eliminate all the unnecessary flip-flop and latch elements in the design. Placing only the clock sensitive signals under the edge sensitive statement can eliminate the unnecessary flip-flops. Similarly the unwanted latches can be avoided by specifying the values for the signals under all conditions of an *if/case* statement.

- Using Parentheses.

The usage of parentheses is critical to the design as the correct usage might result in better timing paths.
Ex.

```
Result <= R1 + R2 - P + M;
```

The hardware generated for the above code is as given below in the figure 21.

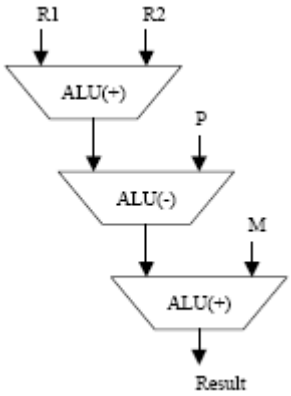


figure 21 - Without using parentheses

If the expression has been written using parentheses as given below, the hardware synthesized would be as given in figure 22.
Result <= (R1 + R2) – (P - M);

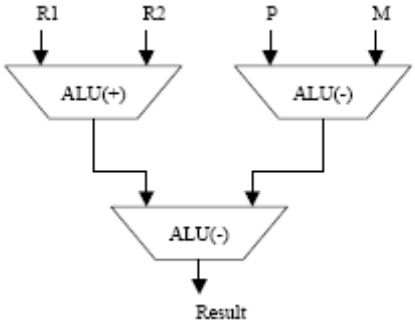


figure 22 - After using parentheses

It is clear that after using the parentheses the timing path for the datapath has been reduced as it does not need to go through one more ALU as in the earlier case.

- Partitioning and structuring the design.

A design should always be structured and partitioned as it helps in reducing design complexity and also improves the synthesis run times since it smaller sub blocks synthesis synthesize faster. Good partitioning results in the synthesis of a good quality design.

For the optimization of design, to achieve minimum area and maximum speed, a lot of experimentation and iterative synthesis is needed. The process of analyzing the design for speed and area to achieve the fastest logic with minimum area is termed – design space exploration. For the sake of optimization, changing of HDL code may impact other blocks in the design or test benches. For this reason, changing the HDL code to help synthesis is less desirable and generally is avoided. It is now the designer's responsibility to minimize the area and meet the timing requirements through synthesis and optimization using Design Compiler. Various optimization techniques that help in achieving better area and speed for
our design are given below.

- Compile the design

  The compilation process maps the HDL code to actual gates specified from the target library. This is done through the compile command. The compile command by default uses the –map_effort medium option. This usually produces the best results for most of the designs. It also default settings for the structuring and flattening attributes. The map_effort high should only be used, if target objectives are not met through default compile. The -incremental_mapping is used only after initial compile as it works only at gate-level. It is used to improve timing of the logic.

- Flattening and structuring

  Flattening implies reducing the logic of a design to a 2-level AND/OR representation. This approach is used to optimize the design by removing all intermediate variables and parenthesis. This option is set to "false" by default. The optimization is performed in two stages. The first stage involves the flattening and structuring and the second stage involves mapping of the resulting design to actual gates, using mapping optimization techniques. Flattening reduces the design logic in to a two level, sum-of-products of form, with few logic levels between the input and output. This results in faster logic. It is recommended for unstructured

designs with random logic. The flattened design then can be structured before final mapping optimization to reduce area. This is important as flattening has significant impact on area of the design. In general one should compile the design using default settings (flatten and structure are set as false). If timing objectives are not met flattening and structuring should be employed. It the design is still failing goals then just flatten the design without structuring it.

The default setting for structuring is “true”. This method adds intermediate variables that can be factored out. This enables sharing of logic that in turn results in reduction of area.

- Removing hierarchy

Dc by default maintains the original hierarchy that is given in the RTL code. The hierarchy is a logic boundary that prevents DC from optimizing across this boundary. Unnecessary hierarchy leads to cumbersome designs and synthesis scripts and also limits the DC optimization within that boundary, without optimizing across hierarchy. To allow DC to optimize across hierarchy one can use the following commands.dc_shell> **current_design <**design name>

dc_shell> **ungroup –flatten –all**

This allows the DC to optimize the logic separated by boundaries as one logic resulting in better timing and an optimal solution.

- Optimizing for Area

DC by default tries to optimize for timing. Designs that are not timing critical but area intensive can be optimized for area. This can be done by initially compiling the design with specification of area requirements, but no timing constraints. In addition, by using the **don_touch** attribute on the high-drive strength gates that are larger in size, used by default to improve timing, one can eliminate them, thus reducing the area considerably. Once the design is mapped to gates, the timing and area constraints should again be specified (normal synthesis) and the design re-compiled incrementally. The incremental compile ensures that DC maintains the

previous structure and does not bloat the logic unnecessarily.

3.6.8 Analyze and Resolve Design Problems

Design Compiler can generate numerous reports on the results of a design synthesis and optimization, for example, area, constraint, and timing reports. We use reports to analyze and resolve any design problems or to improve synthesis results. We can use the check_design command to check the synthesized design for consistency. A design is consistent when it does not contain errors such as unconnected ports, constant-valued ports, cells with no input or output pins, mismatches between a cell and its reference, multiple driver nets, connection class violations, or recursive hierarchy definitions.

3.6.9 Save the Design Database

We use the write command to save the synthesized designs.

Παράρτημα Γ : Υλοποίηση

In this chapter we focus on the procedure we choose to implement the Ccproc to ASIC. Firstly we had to modify some of the FPGA dependent HDL files into a technology-independent form. We also had to change all the components that where implemented with ISE core generator. Xilinx core generator uses hardware from a specific technology which is unknown to Synopsys. Then we will specify the technology library that we will use. Next we will set the design constraints and select a compile strategy for our design. Then we will see a few optimizations and finally we will verify our design and analyze the results.

4.1 Developing VHDL for Synopsys

Most of the components that are low in hierarchy where produced from Xilinx core generator. Elements such us comparators, adders, multipiers and memory's had to change in a way that Synopsys could map them to an ASIC technology. All the components except memory's where implemented finally with an in depended VHDL coding style. The procedure we followed is:

1. We change a VHDL file in a technology-independent VHDL.

2. We test the new element to see if it works as before. If it doesn't work as It did we have to return to 1. If It works with the same way as it did before the change we continue to next stage. We used modelsim to complete our testbenches.

3. We read the new VHDL file with synopsys .

4. We ask from Synopsys to compile our source.

5. We ask from Synopsys to produce a netlist(Gate level) of this component.

6. We test new Gate level VHDL or VERILOG file to be sure that it works as the dependent file did. If it works as it should, we have finished our procedure,

if not we have to return to step 1.

4.1.1 Modyfying An Adder Produced By Coregenerator

The adder (add1) is a VHDL file produced from Xilinx core generator. This adder has one input (6 bit) and one output (6 bit).This adder takes an input and produces an output incremented by one. We wrote behavioural VHDL for this element :

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity add1 is
  port (
    A : in STD_LOGIC_VECTOR ( 5 downto 0 );
    S : out STD_LOGIC_VECTOR ( 5 downto 0 )
  );
end add1;

architecture behave of add1 is
    begin
    process(A)
        begin
            S<= A+1;
        end process;

end behave;
```

After we wrote the independent VHDL code we tested it with modelsim and we took the same results with the core generator's adder. We also tested it with the hole design. After this we read the behavioural VHDL with design compiler.After this we compile add1.vhd and then take the netlist in VHDL or in VERILOG :

```
Read    -f vhdl  add1.vhd
Compile
Write –f vhdl  -o  add1_gatelevel.vhd
```

The output of synopsys (add1_gatelevel) is :

```
library IEEE;
```

```
use IEEE.std_logic_1164.all;

package CONV_PACK_add1 is

-- define attributes
attribute ENUM_ENCODING : STRING;

end CONV_PACK_add1;

library IEEE;

use IEEE.std_logic_1164.all;

use work.CONV_PACK_add1.all;

entity add1_DW01_inc_6_0 is

  port( A : in std_logic_vector (5 downto 0);  SUM : out std_logic_vector (5
       downto 0));

end add1_DW01_inc_6_0;

architecture SYN_rpl of add1_DW01_inc_6_0 is

  component HDEXOR2DL
    port( A1, A2 : in std_logic;  Z : out std_logic);
  end component;

  component HDINVD1
    port( A : in std_logic;  Z : out std_logic);
  end component;

  component HDADHALFD1
    port( A, B : in std_logic;  CO, S : out std_logic);
  end component;

  signal carry_5_port, carry_4_port, carry_3_port, carry_2_port : std_logic;

begin

  U5 : HDEXOR2DL port map( A1 => carry_5_port, A2 => A(5), Z => SUM(5));
  U6 : HDINVD1 port map( A => A(0), Z => SUM(0));
  U1_1_1 : HDADHALFD1 port map( A => A(1), B => A(0), CO => carry_2_port, S
=>
                    SUM(1));
  U1_1_2 : HDADHALFD1 port map( A => A(2), B => carry_2_port, CO =>
                    carry_3_port, S => SUM(2));
  U1_1_3 : HDADHALFD1 port map( A => A(3), B => carry_3_port, CO =>
                    carry_4_port, S => SUM(3));
  U1_1_4 : HDADHALFD1 port map( A => A(4), B => carry_4_port, CO =>
                    carry_5_port, S => SUM(4));

end SYN_rpl;

library IEEE;
```

```
use IEEE.std_logic_1164.all;

use work.CONV_PACK_add1.all;

entity add1 is

  port( A : in std_logic_vector (5 downto 0);  S : out std_logic_vector (5
      downto 0));

end add1;

architecture SYN_behave of add1 is

  component add1_DW01_inc_6_0
    port( A : in std_logic_vector (5 downto 0);  SUM : out std_logic_vector
        (5 downto 0));
  end component;

begin

  add_21_plus_plus : add1_DW01_inc_6_0 port map( A(5) => A(5), A(4) => A(4),
                    A(3) => A(3), A(2) => A(2), A(1) => A(1), A(0) =>
                    A(0), SUM(5) => S(5), SUM(4) => S(4), SUM(3) => S(3)
                    , SUM(2) => S(2), SUM(1) => S(1), SUM(0) => S(0));

end SYN_behave;
```

After we take our adder in gatelevel we test it again with the same testbenches we have tested the adder in behavioural source. The results are the same so we are sure that our VHDL file in a technology-independent VHDL is ready for synthesis. The truth is that we have already synthesized because we got the adder in gatelevel and it works properly but we still have some steps such as optimization and setting constraints to compete successfully the synthesis process.

4.1.2 Modifying A Multiplier That Was Mapped On A DSP

Another example of trying to change a component to a technology-independent VHDL is the Ccproc multiplier (mmul32x32.vhd). As we have already seen in chapter 2 our multiplier was implemented using hardware of Virtex 4. In this FPGA there is a new block called “XtremeDSP slice” that intergrades an 18x18 multiplier along with a 48x48 adder. We followed the

same process that we did with the adder and we finally got the multiplier in a technology-independent VHDL, ready for synthesis.

4.1.3 Altering Memories

Altering memory's process is different. The difference is that we have to decide first the technology we want to use. Memory components are not written in hardware independent VHDL. The memory's we use are components designed for a specific technology and have fixed specifications. Hence, the first thing we have to do is to order from our semiconductor vendor memory's with the characteristics that fits to our design and to the technology library we want to use. After we have to produse with the help of design compiler the .db file that corresponds to the memory we want to use. When we have produce the .db file of memory we consider our memory as a black box (having knowledge of it's characteristics from the data sheet they have supplied us). For further tests
In modelsim we use the .vhd or .v file that our semiconductor vendor has supplied us. If the specifications of our new memory don't feet to the specifications of the memory we want to alter we have to environ our memory with elements that will force it to work with the same way as the FPGA memory (the memory we want to alter) did. The steps we have to follow to alter a memory from an FPGA to an ASIC technology are:

1. We decide the technology we want to use.

2. We order from our semiconductor vendor a memory compatible to the technology we will finally use for our design. The memory we order have to meet as possible as it gets the specifications of the memory we are trying to alter (size, control signals).

3. We environ our memory with additional logic in a new vhdl file using the .vhd of the memory we have order as a component (black box). Our

purpose is that the new memory with the additionally logic will have the same behaviour with the memory we want to alter. We want to be sure that our component is working exactly in the same way with the one we want to change so we try to make our testbenches include as many cases as possible. If we succeed finally to make our new component to behave as the one we want to change we proceed to the next step, if not we repeat step 3.

4. We read with the help of design compiler the .lib file that corresponds to our new memory and producing the equivalent .db.

5. Now we read our new vhdl file with the additional logic and link it to the library of our new memory (with the .db).

6. We compile our vhdl file with Synopsys.

7. We ask from Synopsys to produce a netlist (Gate level) of this component.

8. We test the new Gate level memory to be sure that it works as the memory we changed did. If it works as it should, we have finished our procedure, if not we have to return to step 3.

An example of this procedure was the alteration of the instruction memory. The instruction memory is a read only memory with width 128 bits and depth 256. This memory is single port and supports read after write. All memory operations are synchronous with rising edge of clock input. This memory supports also ENABLE and SINIT (synchronous initialization) signals. The enable pin affects the read, write, and SINIT functionality of the port. When the Block Memory has an inactive enable pin, the output pins are held in the previous state and writing to the memory is disabled .The enable pin is active high. When enabled, the SINIT pin forces the data output latches to synchronously load the predefined SINIT value. For the Virtex

implementation, the SINIT value is zero. Therefore, asserting the SINIT pin causes the output latches to reset. The SINIT pin is active high.

On the other hand the memory we wanted to use is a single port SRAM and has a single address port (ADR, for both the read and write address) and separate data input (DI) and data output (DOUT) ports. Read and write cycles are timed with respect to a single edge of the clock (CK). During both read and write cycles, the write enable (WEN) and cell enable (CEN) inputs are sampled by the rising edge of the clock. During a read cycle, if CEN is de-asserted, data output bus values are held. When CEN is low, WEN is high and output enable (OEN) is low, data from the addressed location is propagated to the data output bus using self-timed circuitry. The CEN feature is used to save RAM power without the need for external gating. During a write cycle, if CEN is low, sampled input data is written to the specific address location, and written data is also propagated to the data output. The DOUT bus has an asynchronous 3-state output enable control (OEN). ADR, DI, WEN, and CEN are latched on the rising edge of the clock. For read and write operations, the output will appear on the DOUT pins after
the access time delay. OEN is asynchronous and not latched. The data will appear on the output (after delay and access time) only if OEN is asserted. If OEN is always asserted, the RAM will actively drive the output pins.

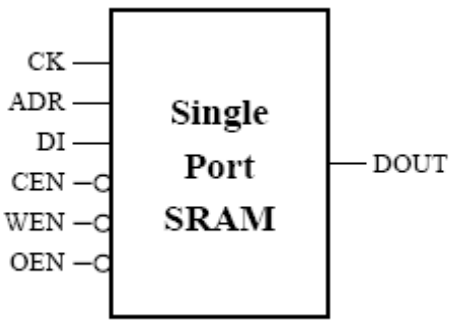


figure 23 – Single Port memory from our semiconductor vendor

As we can see in figure 23 there is no an SINIT signal and the OEN differs from the ENABLE.

The first thing we have to do is to keep WEN to ‘1’ and DI to zero because we want a read only memory. We also need OEN and CEN to ‘0’ because we will not use them :

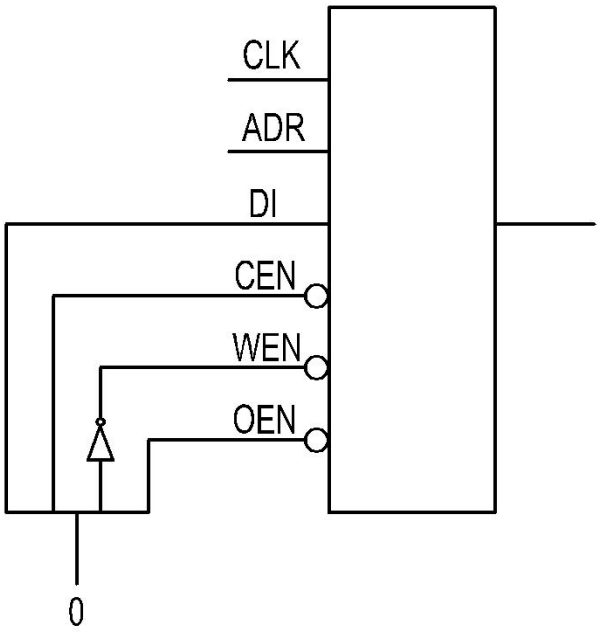


figure 24 – first design change to memory

The next thing we have to consider is to create a synchronous enable and an SINIT signal. Every time that a new address occurs at the next rising clock edge the memory output produces new data. But if the enable signal becomes ‘0’ we need to keep the output on the previous stage. Our new memory doesn’t support this functionality. So we have to keep our previous data in a register when enable is ‘0’. ENABLE signal and new addresses always changes at the first half of the clock period (This happens always at the instruction memory because they depend from the data of the memory output). Having that in mind our register will be triggered at the falling edge when ENABLE is ‘0’:

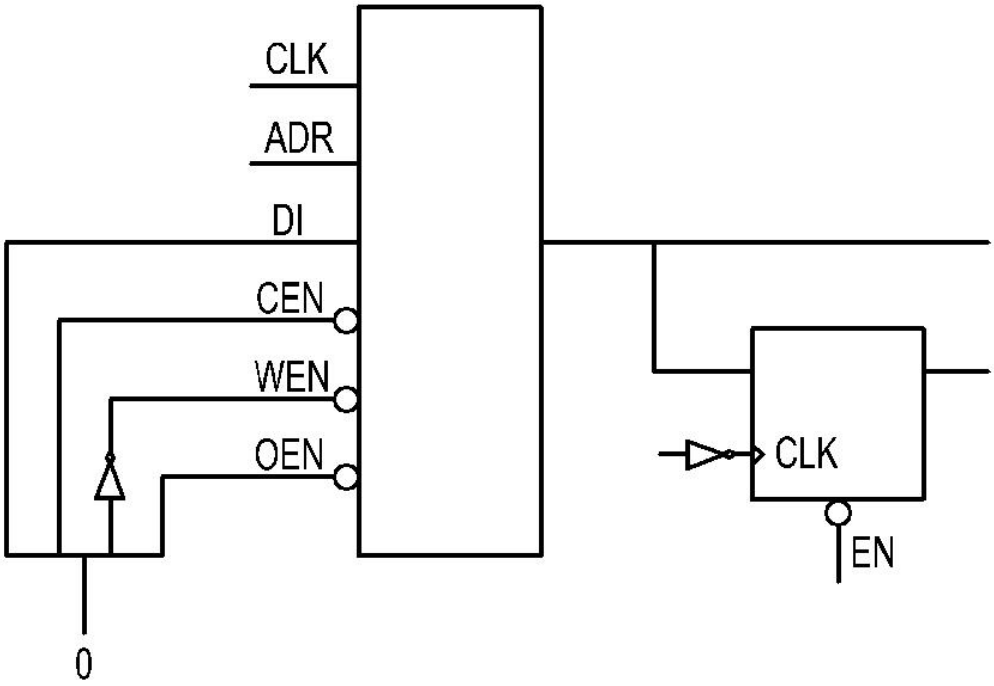


figure 25 – Second design change to memory

The next step is to choose between the memory output and the register output when the next rising edge occurs. We use a multiplex with a synchronous control signal (the enable passes through a register) as we can see in figure 26.

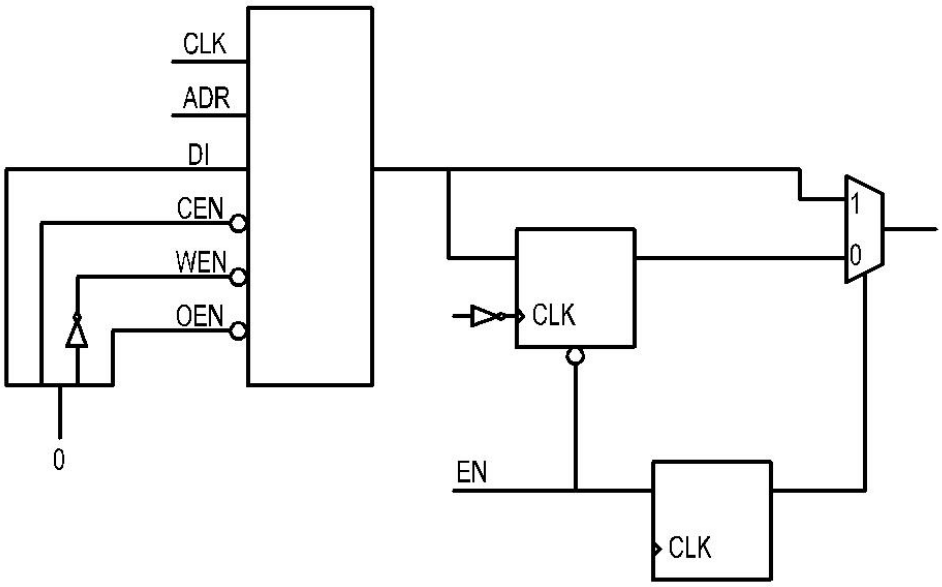


figure 26 – third design change to memory

The SINIT signal also doesn't exist to our memory. Our memory has its register at the address and not at the output as the one we want to change. So we have to simulate the animalization of the output register of the memory. We accomplished this functionality adding to our design an extra multiplexer as we can in figure 27.

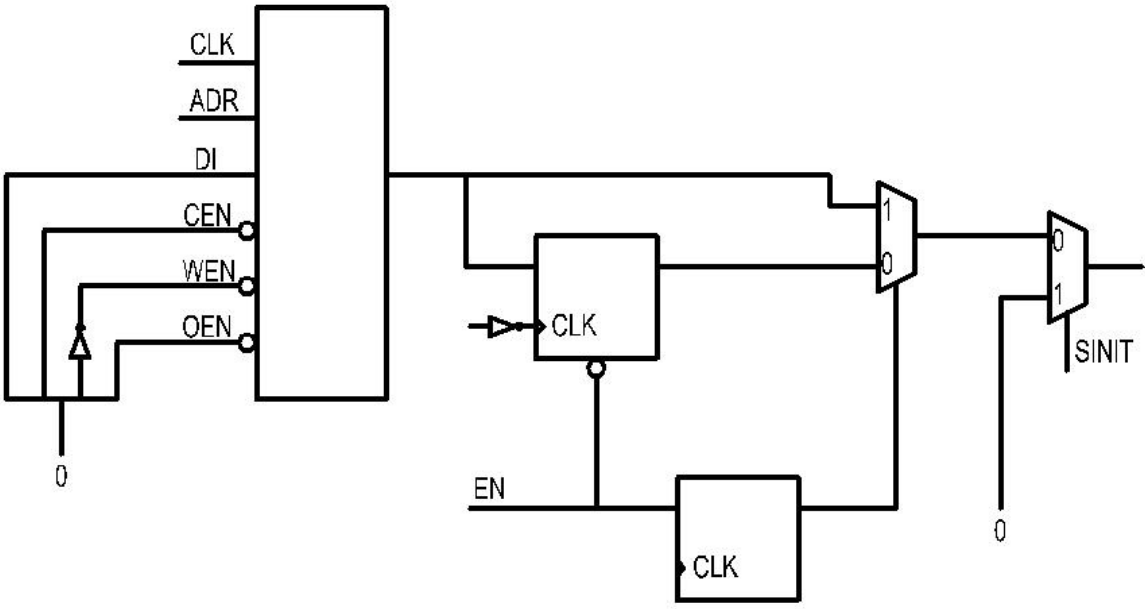


figure 27 – fourth design change to memory

One more thing we have to consider is that we want the output only at the rising edge. Having the multiplex to the output every time a change occurs (at his inputs) we will have a new output. To overcome this problem we inserted a synchronous latch between the multiplex output and the final output as we can see in figure 28.

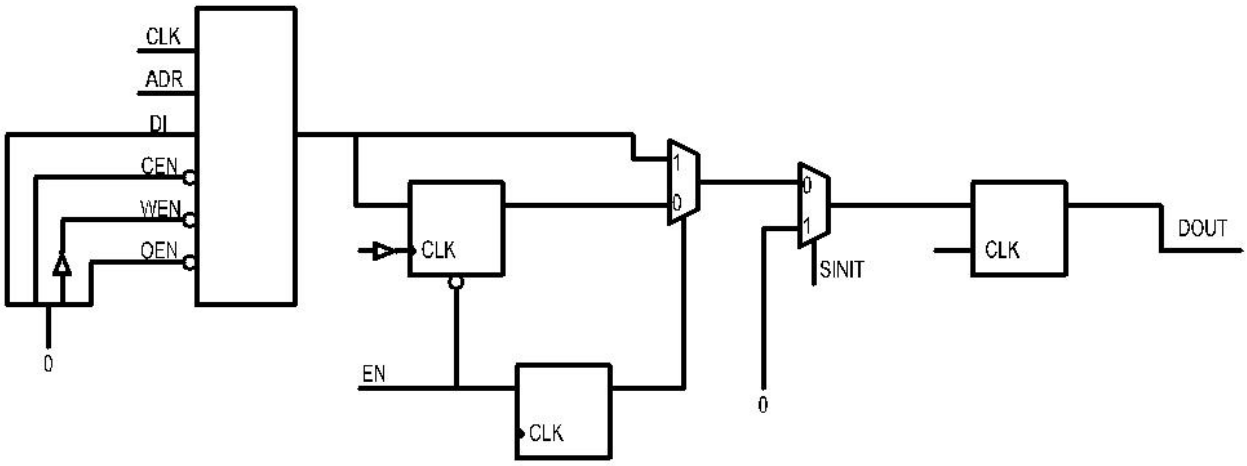
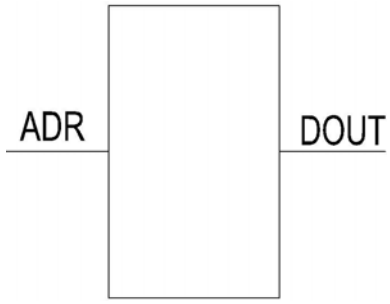


figure 28 – Instruction memory using the memory of our our semiconductor vendor

After we checked our new design and verified that it worked in the same way as the one we have changed we continued to next steps. When we have reached step 8 we verified our netlist with modelsim to make sure our processor had the attitude that had before the change.

Another situation, of the step 3 of our procedure, we had to face was that some of the original Ccproc memories where asynchronous. The memories we have ordered and would take their place where all synchronous. What we have done was to replace an asynchronous memory with a synchronous. An example of this situation was aessbox which had an address input and a data output as we can see in figure 29.



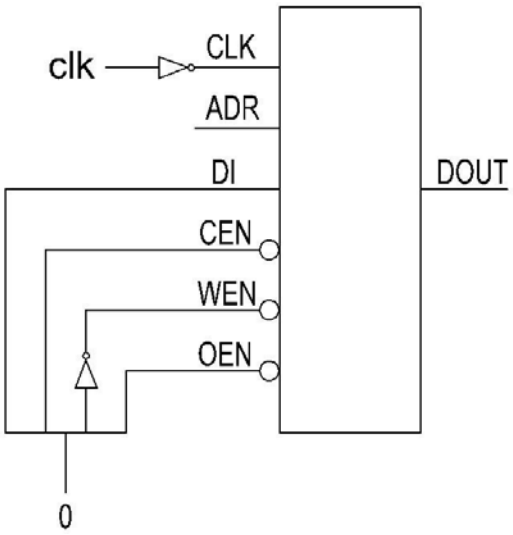


figure 29 – asynchronous memory

figure 30 - synchronous memory

We added a clock input and used as a component one of our synchronous memory's that fitted better to the size of the asynchronous memory. We also made our memory to give output to the falling edge in order to work in a similar way to an asynchronous memory as we can see in figure 30.

The reason that those designs work in the same way is that the asynchronous memory has registers before its input and registers next to its output that are triggered on the rising edge. So if the registers before input supplied our asynchronous memory after a rising edge the registers at the output will have the new data in next clock cycle. Considering that a synchronous memory lathed on the falling edge will supply the register at the output before the next rising edge. We can see what happens in figure 31.

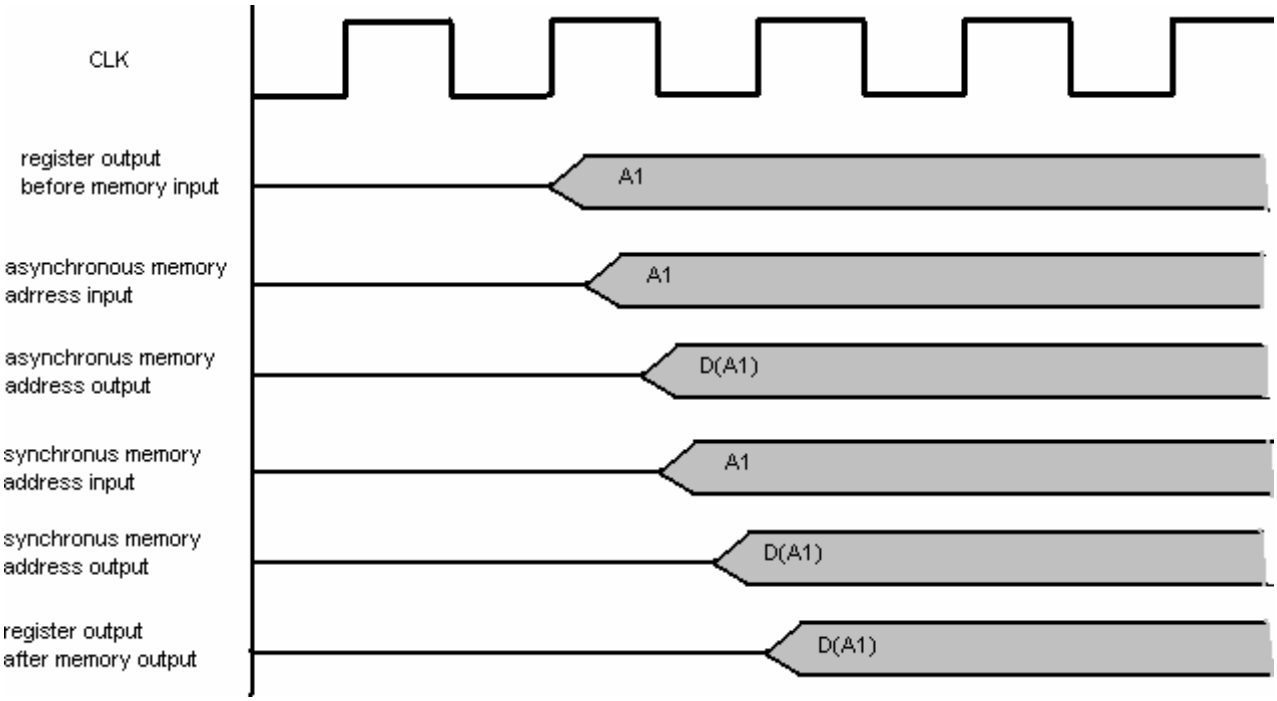


figure 31 – Waveform comparison

In the case of serpent sboxes we decided to change the asynchronous memory's with combinational logic. Every sbox is a memory block with width 4 bits and depth 16. In the final implementation there wold be 512 instances of those memory blocks. We decided to use combinational logic to avoid to use so many memory's in purpose to decrease the cost and the space of our design.

An example of VHDL coding of a serpent sbox :

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity serpent_s4 is
  port (
    A : in STD_LOGIC_VECTOR ( 3 downto 0 );
    SPO : out STD_LOGIC_VECTOR ( 3 downto 0 )
  );
end serpent_s4;
```

```
architecture STRUCTURE of serpent_s4 is

begin

process(A)
begin
case A is
  when "0000" => SPO <= "0001";
  when "0001" => SPO <= "1111";
  when "0010" => SPO <= "1000";
  when "0011" => SPO <= "0011";
  when "0100" => SPO <= "1100";
  when "0101" => SPO <= "0000";
  when "0110" => SPO <= "1011";
  when "0111" => SPO <= "0110";
  when "1000" => SPO <= "0010";
  when "1001" => SPO <= "0101";
  when "1010" => SPO <= "0100";
  when "1011" => SPO <= "1010";
  when "1100" => SPO <= "1001";
  when "1101" => SPO <= "1110";
  when "1110" => SPO <= "0111";
  when others => SPO <= "1101";

  end case ;

end process;

end STRUCTURE ;
```

4.2 Specify Libraries

The next step of our process was to decide a cell library. We ended up to the 0.13um High Density Standard Cell Library. The 0.13um High Density Standard Cell Library provides today's advanced synthesis tools with the necessary building blocks to efficiently implement the most complex,

performance demanding designs. This library uses an innovative, synthesis-optimized cell set and a density-optimized 9-track layout architecture to offer systems designers the maximum area efficiency without sacrificing performance. This 0.13um High Density library is enhanced by:

• A rich set of boolean functions.
• Multiple drive strengths per cell.
• Highly accurate timing and power characterizations.
The library makes use of:
• Handcrafted layouts that use only metal one within the cells
• A rich set of balanced clock buffers
• Careful attention to routing porosity

4.3 Set Design Constraints

We decided to choose a long clock period for the first compilation in order to see if our design functions well. Later on the optimization stage we will worry about max frequency. So we choose a period of 7 ns(142MHz). In the next stage we had to choose a compile strategy and compile the Ccproc.

4.4 Select Compile Strategy

The compile strategy we choose to follow was bottom-up. We preferred this strategy from up-bottom because we wanted to compile each subdesign seperatly. In this way we have the advantage to be sure for the stability of each component.

In this compile strategy we see our design as a tree and we begin to compile the leafs. In the next step we compile the designs of the next higher level of the hierarchy. We continue the procedure until we reach the top entity. As we can see in figure 32 the top entity is cryptium (root) and clusterA, clusrerB, clusterC, clusterD, ifstage and SerpentSboxes are the next stage in hierarchy.

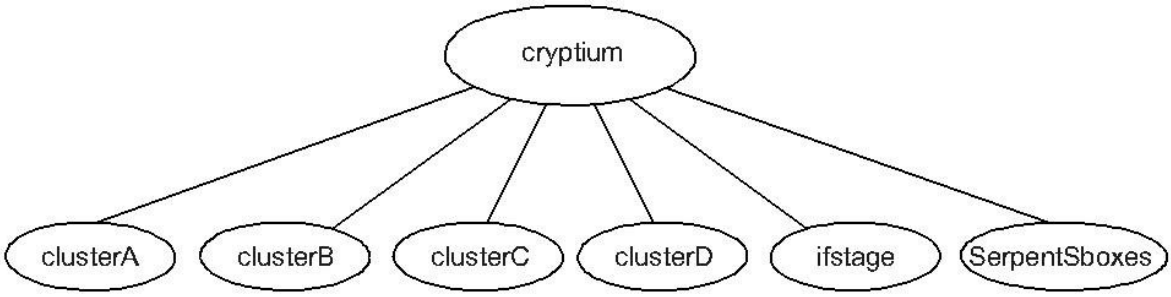


figure 32 – cryptium's tree

As we can see in figure 33 clusterA's components are decstageA, exstageA and memstageA .

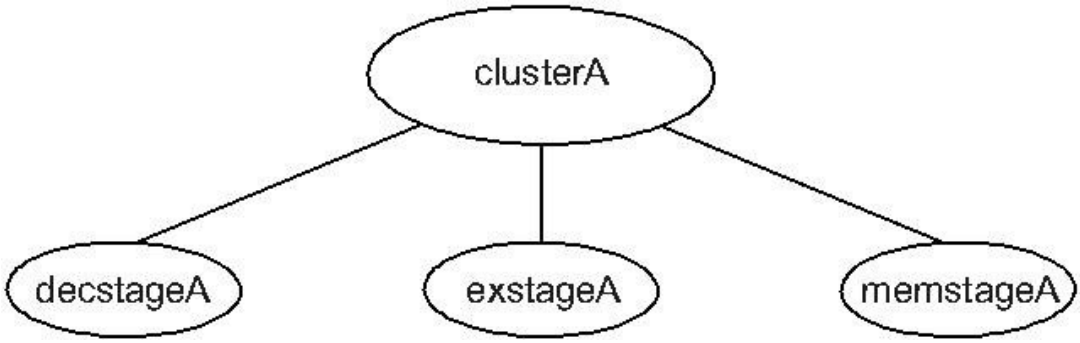


figure 33 – clusterA's tree

In figures 34, 35 and 36 we see the hierarchy in decstageA, exstageA and memstageA .

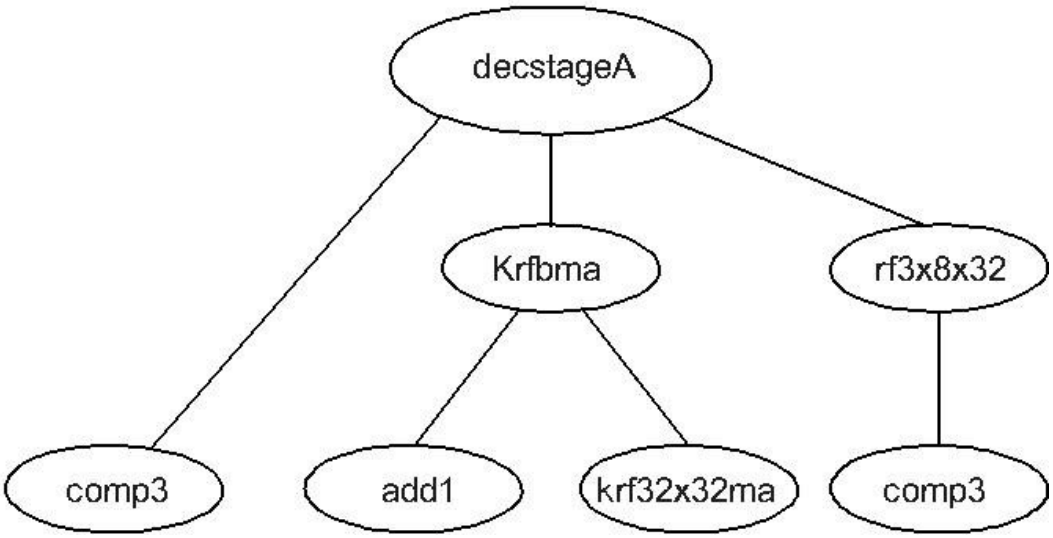


figure 34 – decstageA's tree

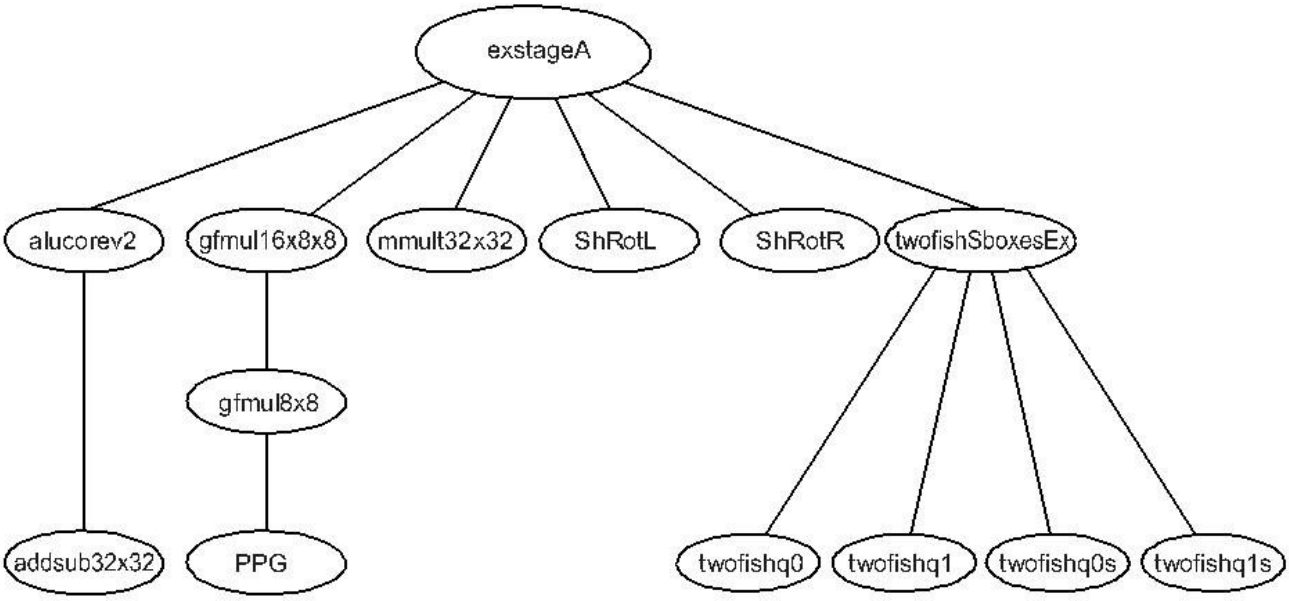


figure 35 – exstageA's tree

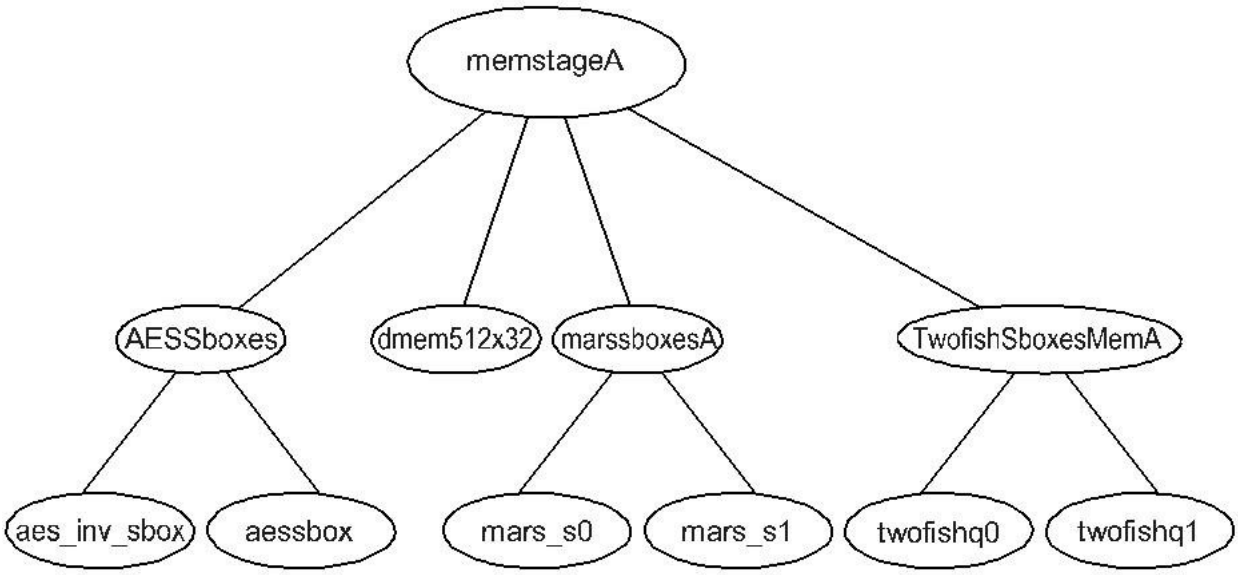


figure 36 – memstageA's tree

We begin to compile the components from the bottom of the tree. For example, if we want to compile decstageA , we compile firstly add1, Krf32x32bma and comp3. Then we proceed to Krfbma, wich includes add1 and Krf32x32bma, rf3x8x32, which includes comp3. We can see the steps of the procedure in figures 37, 38, 39.

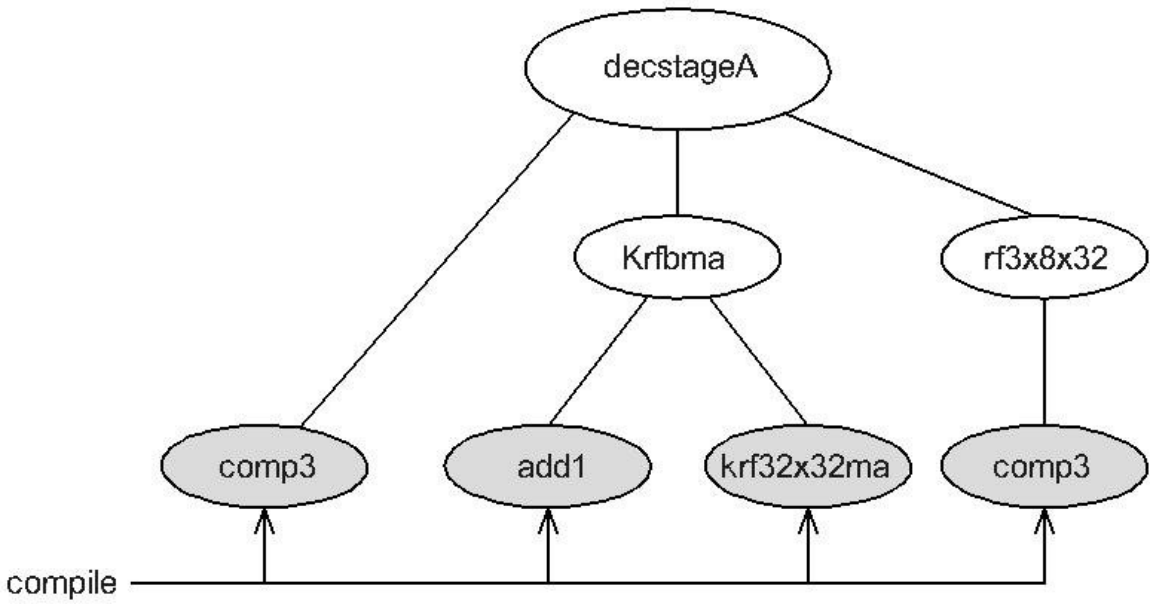


figure 37 – first compilation step for decstageA's tree

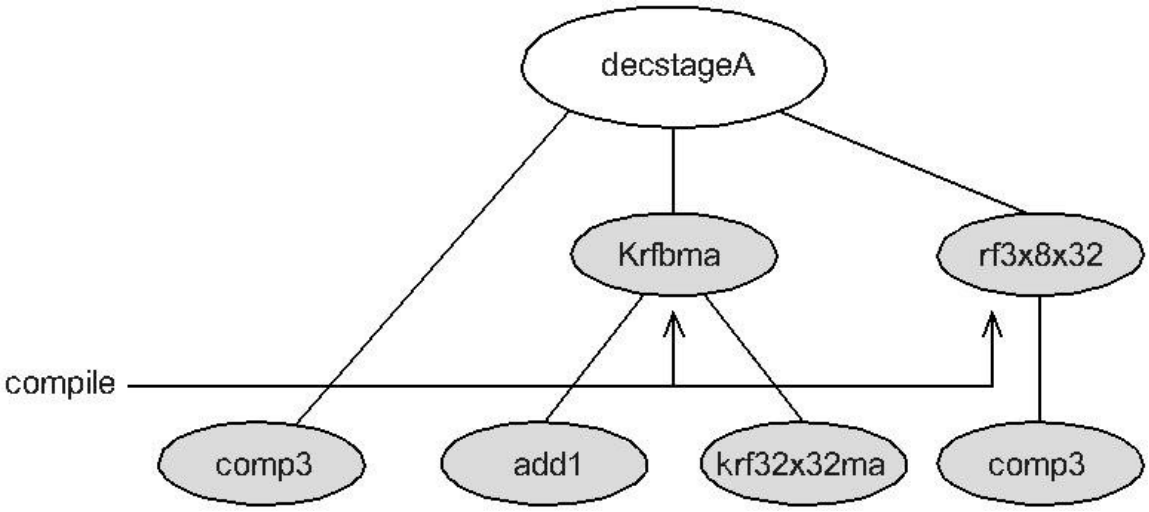


figure 38 – second compilation step for decstageA's tree

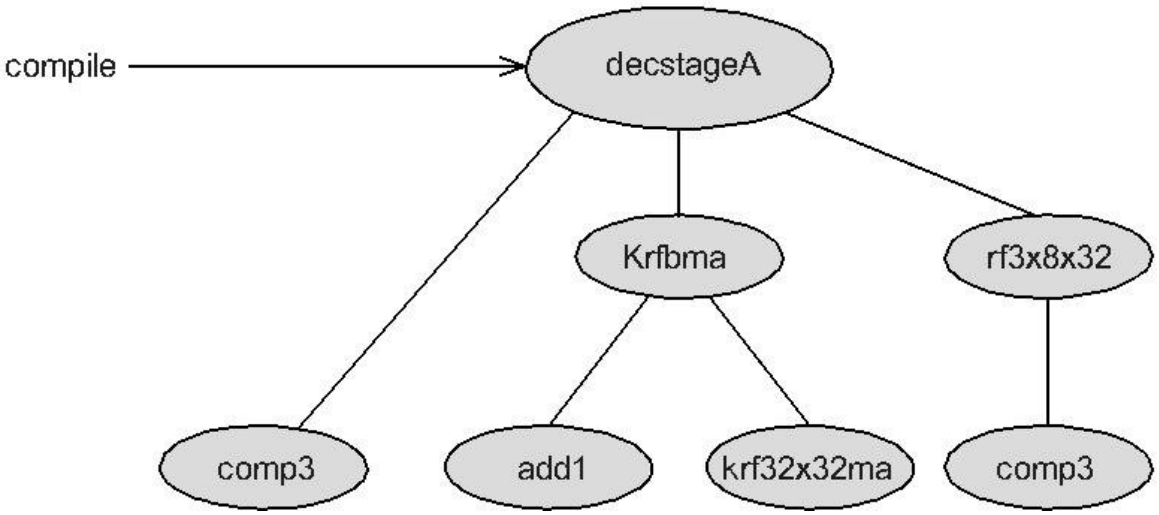


figure 39 – final compilation step for decstageA's tree

Follwing the same procedure we compile exstageA and memstageA.

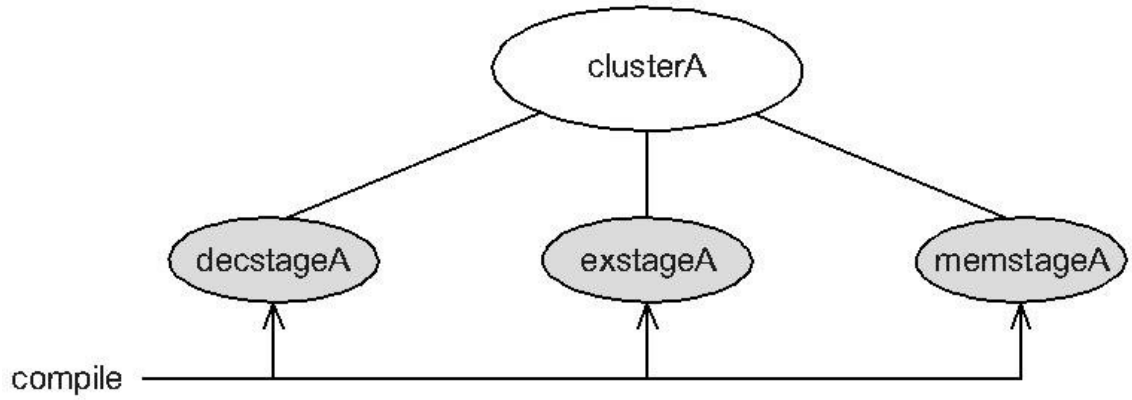


figure 40 – first compilation step for clusterA's tree

Then we compile clusterA.

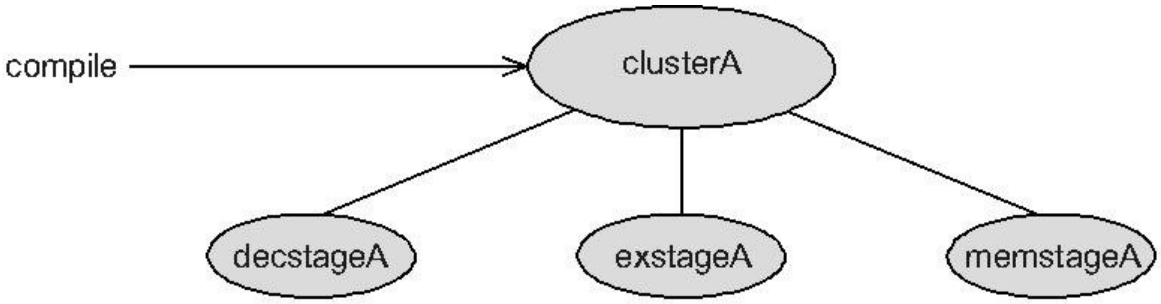


figure 41 – final compilation step for clusterA's tree

The next step is to compile clusrerB, clusterC, clusterD, ifstage and SerpentSboxes wich are in the same stage of hierarchy with clusterA. We use the same method as we did with clusterA and we finally get to cryptium.

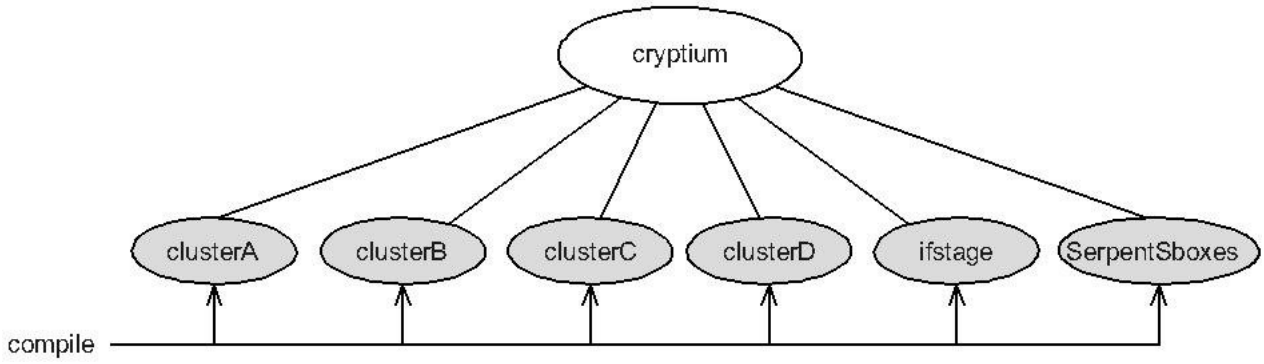


figure 42 – first compilation step for cryptium's tree

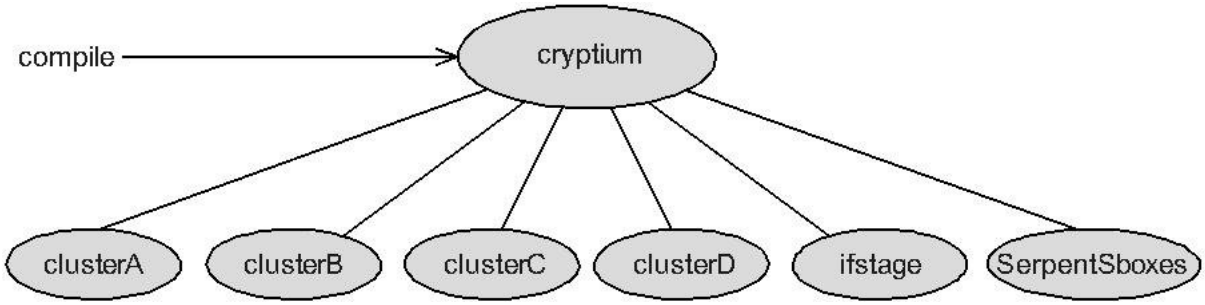


figure 43 – final compilation step for cryptium's tree

When we reach and compile successfully the top entity we save the .db ,the .v (or .vhd) gatelevel and the .sdf files that will use to verify our design.

When we make sure that our gatelevel works properly we continue to the optimization step.

4.5 Optimize the Design

As we discussed before our design was firstly implemented at 142 MHz. The first assumption was that our critical path should be the instruction memory (350 Mhz). After we had the timing reports of the synthesis we discovered that our critical path found at the execution stage at the multiplier(mmult32x32). Our design couldn't exceed the 200 Mhz because of the multiplier. The vhdl source was trying to imitate the implementation that we saw at 2.5 chapter. The assumption we made was that the registers (from

the two pipeline stage) our multiplier had, where not well balanced. Thus we used the balance_register command and recompiled the design. The timing results where much better. We accomplished to make our design to work at 250 Mhz (25% optimization). Our new critical path is now the ALU(because it supports double instructions).

decstageA(register)	-> exstageA (ALU input)	0.16 ns
exstageA (ALU input)	-> exstageA (ALU output)	3.91 ns
exstageA (ALU output)	-> AluOutMemRegister	3.94 ns

We didn't try to change the ALU because in cryptography we meat usually double instructions and that means we would spend two clock cycles instead of one, that we spend now. That would meant the reduction of our design's throughput.

4.6 Verification and Results

We used modelsim for the verification of our design. We had to compare the waveforms we had from the FPGA implementation with the waveforms from our gatelevel design.

4.6.1 Functional and Timing Simulation

Every time we changed a module to its gate-level form we substitute it with the original one on Ccproc. Then we where simulating the design with it's new component to verify its correctness. We where comparing signal by signal the waveforms we had from the original design with the new. We compared the waveforms of each algorithm that Ccproc supports (AES, SERPENT, MARS, TWOFISH, RC6). When our ASIC design (Ccproc in a gate-level form) had the same functionality (identiacal waveforms) with the FPGA implementation we introduced our constraints. We managed to achieve max frequency 250 Mhz with the following critical path :

decstageA(register) -> exstageA (ALU input) 0.16 ns
exstageA (ALU input) -> exstageA (ALU output) 3.91 ns
exstageA (ALU output) -> AluOutMemRegister 3.94 ns

Then we loaded the .sdf file to make the design's timing simulation. The delays of the signals didn't influence the functionality of our design. Our design has the same functionality with the original one.

4.6.2 Results

The performance of our design reached the 250MHz as we saw in 4.5. As we saw in chapter 2.6 the max frequency in an FPGA was 108 MHz.
Our design used 93185 cells and 94885 nets. In the FPGA there where used 275452 gates.
The results of the reports are:

RESULTS	number	area μm^2	%area
number of cells	93185	5.343.404	100
sp_mem64x32	4	101.770	1,90459116
sp_mem256x32	88	3773985,531	70,6288638
sp_mem256x128	1	146625	2,74403732
sp_mem512x32	4	279.299	5,22698288
memories	97	4301679,344	80,5044751
flip-flop and latch	8796	320644,2271	6,00074835
non combinational	8893	4622323,571	86,5052235
combinational	84292	721080,429	13,4947765

Table 3 - Results

In table 4 we see the differences between the ASIC design and the FPGA design.

comparison	Max frequency(Mhz)	Ciphers	Clock Cycles	Throughput(Mbits/sec)
ASIC	250	AES	79	405,0632911
		MARS	338	94,67455621
		RC6	242	132,231405
		SERPENT	375	85,33333333
		TWOFISH	178	179,7752809
FPGA	108	AES	79	174,9873418
		MARS	338	40,89940828
		RC6	242	57,12396694
		SERPENT	375	36,864
		TWOFISH	178	77,66292135

Table 4 - Comparison