ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Χρυσανθόπουλος Νίκος

ΑΜ:1617

ΘΕΜΑ: *Σχεδιασμός βασικών ψηφιακών κυκλωμάτων με τεχνικές υψηλών επιδόσεων η και τεχνικές χαμηλής κατανάλωσης σε FPGA ολοκληρωμένο κύκλωμα .*

Ευχαριστίες

Θα ήθελα να εκφράσω τις ευχαριστίες μου σε όλους όσους με οποιονδήποτε τρόπο συνέβαλαν, ώστε να υλοποιηθεί αυτή η διπλωματική εργασία. Ιδιαίτερες ευχαριστίες στον επιβλέποντα καθηγητή κ. Π. Κίτσο για την πολύτιμη καθοδήγηση, συνεργασία και βοήθεια που μου προσέφερε σε όλη τη διάρκεια της εκπόνησης της εργασίας αυτής, καθώς και για την ευκαιρία που μου έδωσε να ασχοληθώ με το συγκεκριμένο θέμα. Οι γνώσεις και οι εμπειρίες που αποκόμισα από τη διαδρομή αυτής της εργασίας είναι πολύ σημαντικές και σίγουρα θα μου φανούν χρήσιμες και στο μέλλον. Τέλος θα ήθελα να ευχαριστήσω την οικογένειά μου για την πλήρη συμπαράσταση, υποστήριξη και υπομονή τους καθ' όλη τη διάρκεια των σπουδών μου.

Περιεχόμενα

Ευχαριστίες ... 2
Πίνακας Εικόνων ... 7
Περίληψη ... 9
Εισαγωγή ... 10
Εργαλεία Γλώσσας και Ανάλυσης ... 10
Επί-Τόπου Προγραμματιζόμενος Πίνακας Πυλών (FPGA) ... 11
Τεχνικός σχεδιασμός ... 11
Ιστορία ... 12
Συγκρίσεις ... 12
Ασφάλεια ... 13
Εφαρμογές ... 14
Αρχιτεκτονική ... 14
Λογικά μπλοκ - Logic blocks ... 14
Σκληρά μπλοκ - Hardblocks ... 15
Clocking ... 16
Τρισδιάστατες αρχιτεκτονικές - 3D architectures ... 16
Σχεδιασμός και προγραμματισμός ... 17
Βασικοί τύποι τεχνολογίας διεργασιών ... 18
Σημαντικοί κατασκευαστές ... 18
VHDL ... 20
Xilinx ISE ... 23
User Interface ... 23
Προσομοίωση ... 24
Σύνθεση ... 24
Αθροιστές ... 25
Τύποι αθροιστών ... 25
Εφαρμογή και Αποτελέσματα ... 26
Σειριακός αθροιστής (BitSerialAdder) ... 26
Αθροιστής μετάδοσης κρατουμένου (RCA – RippleCarryAdder) ... 29
Αθροιστής πρόβλεψης κρατουμένου (CLA – CarryLookaheadAdder) ... 32
Αθροιστής παράκαμψης κρατουμένου (CSK – CarrySkipAdder) ... 34
Αθροιστής επιλογής κρατουμένου (CSL – CarrySelectAdder) ... 37
Αθροιστής με δέντρο Wallace ... 39
Πολλαπλασιαστές ... 43
Εφαρμογή και Αποτελέσματα ... 44

Πολλαπλασιαστής Booth των n·mbits βάσης-2 44
Πολλαπλασιαστής Booth των n·mbits βάσης-4 46
Πολλαπλασιασμός αριθμών με δέντρο Wallace και Dadda 49
Πολλαπλασιαστής με διάταξη πίνακα (ArrayMultiplier) 55
Modular πολλαπλασιαστής με βάση τον αλγόριθμο Montgomery 57
Pipeline 60
Εφαρμογή και Αποτελέσματα 60
Ripple Adder με τεχνική Pipeline .... **Σφάλμα! Δεν έχει οριστεί σελιδοδείκτης.**
Array Multiplier με τεχνική Pipeline 67
Carry Select Adder με τεχνική Pipeline 73
Wallace Tree με τεχνική Pipeline 81
Βιβλιογραφία 91
Παράρτημα Α – Κώδικες Κυκλωμάτων 92
Σειριακός αθροιστής (BitSerialAdder) 92
Αθροιστής μετάδοσης κρατουμένου (RCA – RippleCarryAdder) 95
Αθροιστής πρόβλεψης κρατουμένου (CLA – CarryLookaheadAdder) 95
Αθροιστής παράκαμψης κρατουμένου (CSK – CarrySkipAdder) 99
Αθροιστής επιλογής κρατουμένου (CSL – CarrySelectAdder) 103
Αθροιστής με δέντρο Wallace 103
Πολλαπλασιαστής Booth των n·mbits βάσης-2 108
Πολλαπλασιαστής Booth των n·mbits βάσης-4 109
Πολλαπλασιασμός αριθμών με δέντρο Wallace και Dadda 110
Πολλαπλασιαστής με διάταξη πίνακα (ArrayMultiplier) 117
Modular πολλαπλασιαστής με βάση τον αλγόριθμο Montgomery 118
Ripple Adder με τεχνική Pipeline 120
Array Multiplier με τεχνική Pipeline 122
Carry Select Adder με τεχνική Pipeline 123
Wallace Tree με τεχνική Pipeline 125
Παράρτημα Β – Κώδικες TESTBENCH Κυκλωμάτων 127
Σειριακός αθροιστής (BitSerialAdder) 128
Αθροιστής μετάδοσης κρατουμένου (RCA – RippleCarryAdder) 128
Αθροιστής πρόβλεψης κρατουμένου (CLA – CarryLookaheadAdder) 130
Αθροιστής παράκαμψης κρατουμένου (CSK – CarrySkipAdder) 131
Αθροιστής επιλογής κρατουμένου (CSL – CarrySelectAdder) 132
Αθροιστής με δέντρο Wallace 134
Πολλαπλασιαστής Booth των n·mbits βάσης-2 135

Πολλαπλασιαστής Booth των n·mbits βάσης-4 136
Πολλαπλασιασμός αριθμών με δέντρο Wallace και Dadda 137
Πολλαπλασιαστής με διάταξη πίνακα (ArrayMultiplier) 140
Modular πολλαπλασιαστής με βάση τον αλγόριθμο Montgomery 142
Ripple Adder με τεχνική Pipeline 143
Array Multiplier με τεχνική Pipeline 145
Carry Select Adder με τεχνική Pipeline 146
Wallace Tree με τεχνική Pipeline 148

Πίνακας Εικόνων

Εικόνα 1: RTL σχηματική (top-level block) - Bit Serial Adder 26
Εικόνα 2: RTL σχηματική - Bit Serial Adder 27
Εικόνα 3: Περίληψη σχεδίασης - Bit Serial Adder 27
Εικόνα 4: Προβολή τεχνολογικών σχημάτων (top-level block) - Bit Serial Adder 27
Εικόνα 5: Προβολή τεχνολογικών σχημάτων - Bit Serial Adder 28
Εικόνα 6: Bit Serial Adder 28
Εικόνα 7: TestBench Serial Adder 28
Εικόνα 8: RTL σχηματική (top-level block) - Ripple Carry Adder 29
Εικόνα 9: RTL σχηματική - Ripple Carry Adder 29
Εικόνα 10: Περίληψη σχεδίασης - Ripple Carry Adder 30
Εικόνα 11: Προβολή τεχνολογικών σχημάτων (top-level block) - Ripple Carry Adder 30
Εικόνα 12: Προβολή τεχνολογικών σχημάτων - Ripple Carry Adder 31
Εικόνα 13: Ripple Carry Adder 31
Εικόνα 14: TestBench Ripple Carry Adder 31
Εικόνα 15: RTL σχηματική (top-level block) - Carry Look Ahead Adder 32
Εικόνα 16: RTL σχηματική - Carry Look Ahead Adder 32
Εικόνα 17: Περίληψη σχεδίασης - Carry Look Ahead Adder 33
Εικόνα 18: Προβολή τεχνολογικών σχημάτων (top-level block) - Carry Look Ahead Adder... 33
Εικόνα 19: Προβολή τεχνολογικών σχημάτων - Carry Look Ahead Adder 33
Εικόνα 20: Carry Look Ahead Adder 34
Εικόνα 21: TestBench Carry Look Ahead Adder 34
Εικόνα 22: RTL σχηματική (top-level block) - Carry Skip Adder 34
Εικόνα 23: RTL σχηματική - Carry Skip Adder 35
Εικόνα 24: Περίληψη σχεδίασης - Carry Skip Adder 35
Εικόνα 25: Προβολή τεχνολογικών σχημάτων (top-level block) - Carry Skip Adder 35
Εικόνα 26: Προβολή τεχνολογικών σχημάτων - Carry Skip Adder 36
Εικόνα 27: Carry Skip Adder 36
Εικόνα 28: TestBench Carry Skip Adder 36
Εικόνα 29: RTL σχηματική (top-level block) - Carry Select Adder 37
Εικόνα 30: RTL σχηματική - Carry Select Adder 37
Εικόνα 31: Περίληψη σχεδίασης - Carry Select Adder 38
Εικόνα 32: Προβολή τεχνολογικών σχημάτων (top-level block) - Carry Select Adder 38
Εικόνα 33: Προβολή τεχνολογικών σχημάτων - Carry Select Adder 38
Εικόνα 34: Carry Select Adder 39
Εικόνα 35: TestBench Carry Select Adder 39
Εικόνα 36: RTL σχηματική (top-level block) - Wallace Tree Adder 39
Εικόνα 37: RTL σχηματική - Wallace Tree Adder 40
Εικόνα 38: Περίληψη σχεδίασης - Wallace Tree Adder 40
Εικόνα 39: Προβολή τεχνολογικών σχημάτων (top-level block) - Wallace Tree Adder 40
Εικόνα 40: Προβολή τεχνολογικών σχημάτων - Wallace Tree Adder 41
Εικόνα 41: Wallace Tree Adder 41
Εικόνα 42: TestBench Wallace Tree Adder (1) 41
Εικόνα 43: TestBench Wallace Tree Adder (2) 42
Εικόνα 44: Προβολή τεχνολογικών σχημάτων / RTL σχηματική (top-level block) - Booth βάσης-2 Multiplier 44
Εικόνα 45: RTL σχηματική - Booth βάσης-2 Multiplier 44

Εικόνα 46: Περίληψη σχεδίασης - Booth βάσης-2 Multiplier 45
Εικόνα 47: Προβολή τεχνολογικών σχημάτων - Booth βάσης-2 Multiplier 45
Εικόνα 48: Booth βάσης-2 Multiplier 45
Εικόνα 49: TestBench Booth βάσης-2 Multiplier 46
Εικόνα 50: RTL σχηματική (top-level block) - Booth βάσης-4 Multiplier 46
Εικόνα 51: RTL σχηματική - Booth βάσης-4 Multiplier 47
Εικόνα 52: Περίληψη σχεδίασης - Booth βάσης-4 Multiplier 47
Εικόνα 53: Προβολή τεχνολογικών σχημάτων (top-level block) - Booth βάσης-4 Multiplier. 47
Εικόνα 54: Προβολή τεχνολογικών σχημάτων - Booth βάσης-4 Multiplier 48
Εικόνα 55: Booth βάσης-4 Multiplier 48
Εικόνα 56: TestBench Booth βάσης-4 Multiplier 48
Εικόνα 57: RTL σχηματική (top-level block) - Wallace Multiplier 49
Εικόνα 58: RTL σχηματική - Wallace Multiplier 49
Εικόνα 59: Περίληψη Σχεδίασης - Wallace Multiplier 50
Εικόνα 60: Προβολή τεχνολογικών σχημάτων (top-level block) - Wallace Multiplier 50
Εικόνα 61: Προβολή τεχνολογικών σχημάτων - Wallace Multiplier 51
Εικόνα 62: Wallace Multiplier 51
Εικόνα 63: TestBench Wallace Multiplier 51
Εικόνα 64: RTL σχηματική (top-level block) - Dadda Multiplier 52
Εικόνα 65: RTL σχηματική - Dadda Multiplier 52
Εικόνα 66: Περίληψη σχεδίασης - Dadda Multiplier 53
Εικόνα 67: Προβολή τεχνολογικών σχημάτων (top-level block) - Dadda Multiplier 53
Εικόνα 68: Προβολή τεχνολογικών σχημάτων - Dadda Multiplier 54
Εικόνα 69: Dadda Multiplier 54
Εικόνα 70: TestBench Dadda Multiplier 54
Εικόνα 71: RTL σχηματική (top-level block) - Array Multiplier 55
Εικόνα 72: RTL σχηματική - Array Multiplier 55
Εικόνα 73: Περίληψη σχεδίασης - Array Multiplier 56
Εικόνα 74: Προβολή τεχνολογικών σχημάτων (top-level block) - Array Multiplier 56
Εικόνα 75: Προβολή τεχνολογικών σχημάτων - Array Multiplier 56
Εικόνα 76: Array Multiplier 57
Εικόνα 77: TestBench Array Multiplier 57
Εικόνα 78: RTL σχηματική (top-level block) - Montgomery Multiplier 57
Εικόνα 79: RTL σχηματική - Montgomery Multiplier 58
Εικόνα 80: Περίληψη σχεδίασης - Montgomery Multiplier 58
Εικόνα 81: Προβολή τεχνολογικών σχημάτων (top-level block) - Montgomery Multiplier.... 58
Εικόνα 82: Προβολή τεχνολογικών σχημάτων - Montgomery Multiplier 59
Εικόνα 83: Montgomery Multiplier 59
Εικόνα 84: TestBench Montgomery Multiplier 59

Περίληψη

Στην παρούσα πτυχιακή εργασία παρουσιάζεται η ανάπτυξη λογικών αριθμητικών μονάδων (αθροιστών - πολλαπλασιαστών) που βρίσκονται σε κάθε υπολογιστή. Αρχικά θα κάνουμε μία μικρή εισαγωγή για τα εργαλεία γλώσσας και ανάλυσης που θα χρησιμοποιήσουμε για να δημιουργήσουμε τις λογικές αριθμητικές μονάδες και στη συνέχεια θα μελετήσουμε κυκλώματα διαφόρων τύπων αθροιστών και πολλαπλασιαστών .

Στο πρώτο κεφάλαιο περιγράφεται η εισαγωγή της παρούσας διπλωματικής εργασίας. Στο δεύτερο κεφάλαιο παρουσιάζεται ο Επί-Τόπου Προγραμματιζόμενος Πίνακας Πυλών (FPGA) όπου εκεί υλοποιούνται όλα τα κυκλώματα. Στο τρίτο και τέταρτο κεφάλαιο θα δούμε ποια γλώσσα περιγραφής υλικού θα χρησιμοποιήσουμε καθώς και το πρόγραμμα ανάλυσης το οποίο θα συνθέσει τον κώδικα κατάλληλα για την εφαρμογή του πάνω στην πλακέτα υλικού που θα υλοποιηθούν τα κυκλώματα. Στα 3 τελευταία κεφάλαια θα δούμε υλοποιημένα κυκλώματα αθροιστών, πολλαπλασιαστών και κυκλώματα με τεχνική pipeline αντίστοιχα.

Εισαγωγή

Κάθε μέρα, η τεχνολογία IC προχωράει όσον αφορά τα μοτίβα και την ανάλυση της απόδοσης. Ένα πιο γρήγορο μοτίβο με χαμηλότερη κατανάλωση ενέργειας και μικρότερο χώρο είναι αυτονόητο στα μοντέρνα ηλεκτρονικά μοτίβα. Η πρόοδος στην τεχνολογία ηλεκτρονικών μοτίβων βελτιώνει τη χρήση της ενέργειας, επιτυγχάνει τη γνώση του κώδικα με επιτυχία, μεταδίδει πληροφορίες πιο σταθερά κ.λπ., πολλές από αυτές τις τεχνολογίες προσφέρουν χαμηλή κατανάλωση ενέργειας για να ικανοποιήσουν τις ανάγκες των διαφόρων εφαρμογών. Σε αυτά τα συστήματα εφαρμογής, ένας πολλαπλασιαστής θα μπορούσε να είναι μια βασική αριθμητική μονάδα και ευρέως χρησιμοποιούμενο κυκλώματα όπου η μέθοδος του πολλαπλασιασμού θα έπρεπε να βελτιστοποιηθεί σωστά. Οι πολλαπλασιαστές συνήθως έχουν εκτεταμένο λανθάνοντα χρόνο, τεράστιο χώρο στο κύκλωμα και καταναλώνουν σημαντική ποσότητα ισχύος. Έτσι το μοτίβο χαμηλής ισχύος έχει γίνει πολύ σημαντικό στα συστήματα VLSI. Δεδομένου ότι ο πολλαπλασιαστής είναι ως επί το πλείστον το πιο αργό στοιχείο κατά τη διάρκεια ενός συστήματος, η απόδοση του συστήματος αποφασίζεται από την απόδοση του πολλαπλασιαστή. Επομένως, η βελτιστοποίηση της ταχύτητας και του χώρου ενός πολλαπλασιαστή θα μπορούσε να είναι ένα σημαντικό θέμα αυτές τις μέρες. Ωστόσο, ο χώρος και η ταχύτητα συνήθως έχουν αντικρουόμενους περιορισμούς, ώστε η αυξανόμενη ταχύτητα να καταλήγει σε μεγαλύτερες περιοχές (χώρος) και αντίστροφα. Επιπλέον, η κατανάλωση χώρου και ρεύματος ενός κυκλώματος συσχετίζεται γραμμικά. Επομένως, πρέπει να γίνει ένας συμβιβασμός όσο αναφορά την κατανάλωση ενέργειας και το ποσοστό του χώρου που καταλαμβάνει ένα κύκλωμα πάνω στη πλακέτα.

Εργαλεία Γλώσσας και Ανάλυσης

Για να γραφτεί ένα πρόγραμμα για την υλοποίηση οποιουδήποτε ψηφιακού κυκλώματος, υπάρχουν ποικίλες γλώσσες, που αναφέρονται ως γλώσσα περιγραφής υλικού π.χ. Verilog, VHDL. Για το μοτίβο μας θα χρησιμοποιήσαμε τη VHDL για προγραμματισμό. Η VHDL είναι μία από τις συνήθεις τεχνικές που χρησιμοποιούνται στη μέθοδο aborning ψηφιακού συστήματος. Η τεχνική αυτή επιβάλλεται στο πακέτο δέσμευσης προγραμμάτων που πραγματοποιεί προσομοίωση και εξέταση του σχεδιαζόμενου συστήματος. Ο σχεδιαστής πρέπει αποκλειστικά να περιγράψει το σχεδιασμό του ψηφιακού κυκλώματος σε τύπο ύλης, ενώ δεν είναι το πρόβλημα να αλλάξει το υλικό. Η VHDL έχει τη δύναμη να μειώσει την αξία και το χρόνο ως προς τη δημιουργία ενός ψηφιακού συστήματος, είναι απλή στην αντιμετώπιση προβλημάτων, προσφέρει φορητότητα και πολλές πλατφόρμες την υποστηρίζουν. Υπάρχει η τάση να χρησιμοποιούμε την πλατφόρμα XILINX για να καταγράψουμε τα προγράμματά μας. Όλες οι προσομοιώσεις RTL έχουν γίνει από αυτό το πακέτου αποκλειστικά. Παράλληλα για την αναφορά του κυκλώματος χρησιμοποιήθηκε το εργαλείο σύνθεσης που ενσωματώνεται στο Xilinx.

Επί-Τόπου Προγραμματιζόμενος Πίνακας Πυλών (FPGA)

Μια **προγραμματισμένη σειρά πεδίων** (**FPGA**) είναι ένα ολοκληρωμένο κύκλωμα σχεδιασμένο για να διαμορφώνεται από έναν πελάτη ή έναν σχεδιαστή μετά την κατασκευή - και ως εκ τούτου " προγραμματίζεται στο πεδίο ". Η διαμόρφωση FPGA καθορίζεται γενικά χρησιμοποιώντας μια γλώσσα περιγραφής υλικού (HDL), παρόμοια με αυτή που χρησιμοποιείται για ένα ολοκληρωμένο κύκλωμα (ASIC) για μία συγκεκριμένη εφαρμογή . (Τα διαγράμματα κυκλώματος χρησιμοποιήθηκαν στο παρελθόν για να καθορίσουν τη διαμόρφωση, όπως ήταν για τα ASIC, αλλά αυτό είναι όλο και πιο σπάνιο.)

Τα FPGA περιέχουν μια σειρά προγραμματιζόμενων λογικών μπλοκ και μια ιεραρχία επαναπροσδιοριζόμενων διασυνδέσεων που επιτρέπουν τη δέσμευση των μπλοκ μαζί, όπως πολλές πύλες λογικής που μπορούν να διασυνδεθούν σε διαφορετικές διαμορφώσεις. Τα λογικά μπλοκ μπορούν να διαμορφωθούν έτσι ώστε να εκτελούν πολύπλοκες συνδυαστικές λειτουργίες ή απλά απλές λογικές πύλες όπως οι AND και XOR. Στα περισσότερα FPGAs, τα λογικά μπλοκ περιλαμβάνουν επίσης στοιχεία μνήμης, τα οποία μπορεί να είναι απλά flip-flops ή πιο ολοκληρωμένα μπλοκ μνήμης.

Τεχνικός σχεδιασμός

Οι σύγχρονες προγραμματιζόμενες συστοιχίες πύλης (FPGAs) διαθέτουν μεγάλους πόρους λογικών πυλών και μπλοκ RAM για την υλοποίηση σύνθετων ψηφιακών υπολογισμών. Καθώς τα σχέδια FPGA χρησιμοποιούν πολύ γρήγορους ρυθμούς I / O και αμφίδρομους διαύλους δεδομένων , γίνεται μια πρόκληση να επαληθευτεί ο σωστός χρόνος έγκυρων δεδομένων εντός του χρόνου εγκατάστασης και του χρόνου κράτησης. Ο σχεδιασμός του δαπέδου (Floor planning) επιτρέπει την κατανομή πόρων εντός του FPGA για να καλύψει αυτούς τους χρονικούς περιορισμούς. Τα FPGAs μπορούν να χρησιμοποιηθούν για την υλοποίηση οποιασδήποτε λογικής λειτουργίας που θα μπορούσε να εκτελέσει ένα ASIC. Η δυνατότητα ενημέρωσης των λειτουργιών μετά την αποστολή, η μερική αναδιάρθρωση ενός τμήματος του σχεδιασμού και το χαμηλό μη επαναλαμβανόμενο κόστος μηχανικής σε σχέση με ένα ASIC(παρά το γενικά υψηλότερο κόστος μονάδας), προσφέρουν πλεονεκτήματα για πολλές εφαρμογές.

Μερικά FPGA έχουν αναλογικά χαρακτηριστικά εκτός από τις ψηφιακές τους λειτουργίες. Το πιο συνηθισμένο αναλογικό χαρακτηριστικό είναι ένας προγραμματιζόμενος ρυθμός μετατόπισης σε κάθε πείρο εξόδου, ο οποίος επιτρέπει στον μηχανικό να ρυθμίζει χαμηλές συχνότητες σε ελαφρώς φορτισμένους ακροδέκτες που διαφορετικά θα μπορούσαν να χτυπήσουν ή να συζευχθούν απαράδεκτα και να θέσουν υψηλότερους ρυθμούς των φορτωμένων πύρων σε κανάλια υψηλής ταχύτητας διαφορετικά θα έτρεχαν πολύ αργά. Επίσης, συνηθισμένοι είναι οι ταλαντωτές με κρυστάλλους από χαλαζία , οι ταλαντωτές αντίστασης-χωρητικότητας σε chip και οι βρόχοι με κλειδωμένη φάση με ενσωματωμένους ταλαντωτές ελεγχόμενης τάσηςπου χρησιμοποιούνται για την παραγωγή και τη διαχείριση ρολογιών και για τα υψηλής ταχύτητας (SERDES) ρολόγια μετάδοσης και την ανάκτηση του ρολογιού του δέκτη. Ακριβώς συνηθισμένοι είναι οι διαφορικοί συγκριτές των ακροδεκτών εισόδου που έχουν σχεδιαστεί για σύνδεση με διαύλους διαφορικής σηματοδότησης . Μερικά " FPGA μεικτού σήματος " έχουν

ενσωματωμένους περιφερειακούς αναλογικούς μετατροπείς (ADC) και ψηφιακούς αναλογικούς μετατροπείς (DAC) με μπλοκ αναλογικού σήματος που τους επιτρέπουν να λειτουργούν ως σύστημα-on-a-chip.

Ιστορία

Η βιομηχανία FPGA ξεκίνησε από τη προγραμματιζόμενη μνήμη μόνο για ανάγνωση (PROM) και από τις προγραμματιζόμενες λογικές συσκευές (PLDs). Τα προγράμματα PROM και PLD είχαν την επιλογή να προγραμματίζονται σε παρτίδες στο εργοστάσιο ή στο πεδίο (προγραμματιζόμενα στο πεδίο). Ωστόσο, η προγραμματιζόμενη λογική ήταν ενσύρματη μεταξύ των λογικών πυλών.

Στα τέλη της δεκαετίας του 1980, το Ναυτικό Surface Warfare Center χρηματοδότησε ένα πείραμα που πρότεινε ο Steve Casselman για την ανάπτυξη ενός υπολογιστή που θα μπορούσε να υλοποιήσει 600.000 επαναπρογραμματιζόμενες πύλες. Ο Casselman ήταν επιτυχής και κατείχε ένα δίπλωμα ευρεσιτεχνίας σχετικό με το σύστημα που εκδόθηκε το 1992.

Ορισμένες από τις θεμελιώδεις έννοιες και τεχνολογίες της βιομηχανίας για προγραμματιζόμενες λογικές συστοιχίες, πύλες και λογικά μπλοκ βασίζονται σε διπλώματα ευρεσιτεχνίας που απονεμήθηκαν στους David W. Page και LuVerne R. Peterson το 1985.

Η Altera ιδρύθηκε το 1983 και παρήγαγε την πρώτη επαναπρογραμματισμένη λογική συσκευή του κλάδου το 1984 - το EP300 - που περιείχε ένα παράθυρο χαλαζία στο πακέτο που επέτρεψε στους χρήστες να ανάψουν μια υπεριώδη λάμπα στη μήτρα για να διαγράψουν τα περιεχόμενα της EPROM που κράτησαν τη διαμόρφωση της συσκευής.

Οι συνιδρυτές της Xilinx Ross Freeman και Bernard Vonderschmitt εφευρέθηκαν το 1985 το πρώτο εμπορικά βιώσιμο field-programmable gate array , το XC2064. Το XC2064 είχε προγραμματιζόμενες πύλες και προγραμματιζόμενες διασυνδέσεις μεταξύ των πυλών, την αρχή μιας νέας τεχνολογίας και της αγοράς. Το XC2064 είχε 64 διαμορφωμένα λογικά μπλοκ (CLBs), με δύο πίνακες αναζήτησης τριών εισόδων (LUTs). Πάνω από 20 χρόνια αργότερα, ο Freeman εισήλθε στην Εθνική αίθουσα εφευρέσεων της φήμης για την εφεύρεσή του.

Η Altera και η Xilinx συνέχισαν χωρίς αμφισβήτηση και αναπτύχθηκαν γρήγορα από το 1985 έως τα μέσα της δεκαετίας του 1990, όταν οι ανταγωνιστές έφτασαν στο επίπεδό τους, μειώνοντας το μερίδιο αγοράς. Μέχρι το 1993, η Actel (τώρα Microsemi) εξυπηρετούσε περίπου το 18% της αγοράς. Μέχρι το 2013, η Altera (31%), η Actel (10%) και η Xilinx (36%) αντιπροσώπευαν από κοινού περίπου το 77% της αγοράς FPGA. Η δεκαετία του 1990 ήταν μια περίοδος ταχείας ανάπτυξης για FPGAs, τόσο στην εξελιγμένη κυκλώματα όσο και στον όγκο παραγωγής. Στις αρχές της δεκαετίας του 1990, τα FPGA χρησιμοποιούνταν κυρίως στις τηλεπικοινωνίες και τη δικτύωση. Μέχρι το τέλος της δεκαετίας, τα FPGA βρήκαν τον δρόμο τους στους καταναλωτές, στις αυτοκινητοβιομηχανίες και στις βιομηχανικές εφαρμογές.

Συγκρίσεις

Ιστορικά, τα FPGAs ήταν πιο αργά, λιγότερο αποδοτικά όσο αναφορά την ενέργεια και γενικά είχαν αποκτήσει λιγότερη λειτουργικότητα από ό, τι τα αντίστοιχα ASIC .

Μια παλαιότερη μελέτη είχε δείξει ότι τα σχέδια που υλοποιήθηκαν σε FPGAs χρειάζονται κατά μέσο όρο 40 φορές περισσότερη περιοχή, αντλούν 12 φορές περισσότερη δυναμική ισχύ και τρέχουν στο ένα τρίτο της ταχύτητας των αντίστοιχων υλοποιήσεων πάνω στα ASIC. Πιο πρόσφατα, FPGA όπως το Xilinx Virtex-7 ή το Alter Stratix 5 έχει έρθει σε αντιπαράθεση με τις αντίστοιχες λύσεις ASIC και ASSP, παρέχοντας σημαντικά μειωμένη κατανάλωση ενέργειας, αυξημένη ταχύτητα, χαμηλότερο κόστος υλικών, ελάχιστη υλοποίηση στη πλακέτα και αυξημένες δυνατότητες επανακατασκευής "on the-fly". Όπου προηγουμένως ένα σχέδιο μπορεί να έχει συμπεριλάβει 6 έως 10 ASIC, το ίδιο σχέδιο μπορεί τώρα να επιτευχθεί χρησιμοποιώντας μόνο ένα FPGA.

Πλεονεκτήματα των FPGA περιλαμβάνουν την δυνατότητα επανεγγραφής στον τομέα για την επιδιόρθωση σφαλμάτων, και μπορούν να περιλαμβάνουν μικρότερο χρόνο στην αγορά και χαμηλότερο μη επαναλαμβανόμενο κόστος μηχανικής.

Η Xilinx ισχυρίζεται ότι αρκετές δυναμικές πτυχές της αγοράς και της τεχνολογίας αλλάζουν το πρότυπο ASIC / FPGA:

- Τα κόστη ανάπτυξης ολοκληρωμένου κυκλώματος αυξάνονται ραγδαία
- Η πολυπλοκότητα των ASIC έχει επιμηκύνει τον χρόνο ανάπτυξης
- Οι πόροι της Έρευνας& Ανάπτυξης καθώς και ο αριθμός των εργαζομένων μειώνονται
- Οι απώλειες εσόδων από την καθυστέρηση της υλοποίησης των προϊόντων στην αγορά αυξάνονται
- Οι οικονομικοί περιορισμοί σε μια φτωχή οικονομία οδηγούν σε χαμηλού κόστους τεχνολογίες

Αυτές οι τάσεις καθιστούν τα FPGA μια καλύτερη εναλλακτική λύση από τα ASICs. Ορισμένα FPGAs έχουν τη δυνατότητα μερικής αναδιάρθρωσης που επιτρέπει σε ένα τμήμα της συσκευής να επαναπρογραμματιστεί ενώ άλλα τμήματα συνεχίζουν να τρέχουν.

Ασφάλεια

Όσον αφορά την ασφάλεια, τα FPGA έχουν τόσο πλεονεκτήματα όσο και μειονεκτήματα σε σύγκριση με τα ASIC ή τους ασφαλείς μικροεπεξεργαστές. Η ευελιξία των FPGA καθιστά τις κακόβουλες τροποποιήσεις κατά τη διάρκεια της κατασκευής σε χαμηλότερο κίνδυνο. Προηγουμένως, για πολλά FPGA, το bitstream σχεδιασμού εκτέθηκε ενώ το FPGA το φορτώνει από εξωτερική μνήμη (συνήθως σε κάθε power-on). Όλοι οι μεγάλοι προμηθευτές FPGA προσφέρουν τώρα ένα φάσμα λύσεων ασφάλειας για τους σχεδιαστές, όπως κρυπτογράφηση bitstream και έλεγχος ταυτότητας . Για παράδειγμα, η Altera και η Xilinx προσφέρουν κρυπτογράφηση AES (έως 256 bit) για ροές bit που είναι αποθηκευμένες σε εξωτερική μνήμη flash.

Τα FPGAs που αποθηκεύουν τη διαμόρφωση τους στο εσωτερικό τους σε μη μνήμη flash δεν εκθέτουν το bitstream και δεν χρειάζεται κρυπτογράφηση. Επιπλέον, η μνήμη flash για έναν πίνακα αναζήτησης παρέχει προστασία απλής εκδήλωσης συμβάντων για ορισμένες εφαρμογές.

Με τους Stratix 10 FPGAs και SoCs, η Altera εισήγαγε ένα Secure Device Manager και φυσικά αθόρυβες λειτουργίες για την παροχή υψηλών επιπέδων προστασίας από τις φυσικές επιθέσεις.

Το 2012 οι ερευνητές Σεργκέι Σκορομπωγκάτοφ και Κρίστοφερ Γουντς κατέδειξαν ότι τα FPGAs μπορούν να είναι ευάλωτα σε εχθρικές προθέσεις. Ανακάλυψαν ότι μια κρίσιμη δυσλειτουργία του backdoor είχε κατασκευαστεί σε πυρίτιο ως τμήμα του Actel / Microsemi ProAsic 3, καθιστώντας το ευάλωτο σε πολλά επίπεδα, όπως επαναπρογραμματισμό κρυπτογραφιών και κλειδιών πρόσβασης, πρόσβαση σε μη κρυπτογραφημένο bitstream, τροποποίηση χαρακτηριστικών πυριτίου χαμηλού επιπέδου και εξαγωγή δεδομένων διαμόρφωσης.

Εφαρμογές

Ένα FPGA μπορεί να χρησιμοποιηθεί για την επίλυση οποιουδήποτε προβλήματος που μπορεί να υπολογιστεί . Αυτό αποδεικνύεται τεχνητά από το γεγονός ότι το FPGA μπορεί να χρησιμοποιηθεί για την υλοποίηση ενός απλού μικροεπεξεργαστή , όπως το Xilinx MicroBlaze ή το Altera Nios II. Το πλεονέκτημά τους έγκειται στο ότι μερικές φορές είναι σημαντικά ταχύτερα για ορισμένες εφαρμογές λόγω της παράλληλης φύσης και της βελτιστοποίησης από πλευράς του αριθμού θυρών που χρησιμοποιούνται για μια συγκεκριμένη διαδικασία.

Τα FPGA αρχικά ξεκίνησαν ως ανταγωνιστές στα CPLD για την εφαρμογή της προσωποποιημένης λογικής για τα PCB . Καθώς το μέγεθος, οι δυνατότητές τους και η ταχύτητα αυξήθηκαν, αναλάμβαναν πρόσθετες λειτουργίες μέχρι το σημείο όπου ορισμένα πωλούνται τώρα ως πλήρη συστήματα (SoC). Ιδιαίτερα με την εισαγωγή εξειδικευμένων πολλαπλασιαστών σε αρχιτεκτονικές FPGA στα τέλη της δεκαετίας του 1990, εφαρμογές οι οποίες παραδοσιακά ήταν μοναδικά υλοποιήσιμα σε DSP άρχισαν να ενσωματώνονται σε FPGAs. Μια άλλη τάση στη χρήση των FPGAs είναι η επιτάχυνση υλικού, όπου κάποιος μπορεί να χρησιμοποιήσει το FPGA για να επιταχύνει ορισμένα τμήματα ενός αλγορίθμου και να μοιραστεί μέρος του υπολογισμού μεταξύ του FPGA και ενός γενικού επεξεργαστή.

Παραδοσιακά, τα FPGA έχουν δεσμευτεί για ειδικές εφαρμογές, όπου ο όγκος παραγωγής είναι μικρός. Για αυτές τις εφαρμογές χαμηλού όγκου, το ασφάλιστρο που καταβάλλουν οι εταιρείες στο κόστος υλικού ανά μονάδα για ένα προγραμματιζόμενο τσιπ είναι πιο προσιτό από τους αναπτυξιακούς πόρους που δαπανώνται για τη δημιουργία ενός ASIC. Σήμερα, η νέα δυναμική του κόστους και απόδοσης έχει διευρυνθεί στο φάσμα των βιώσιμων εφαρμογών.

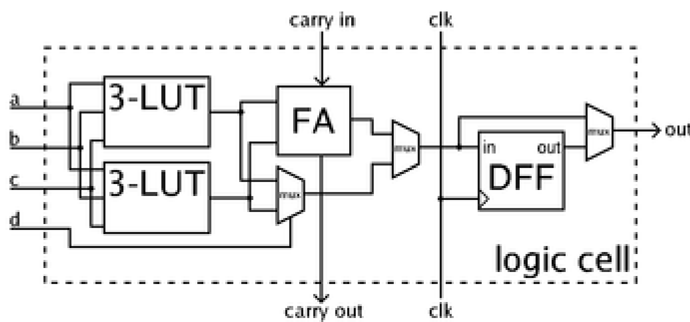
Αρχιτεκτονική

Λογικά μπλοκ - Logic blocks

Η πιο συνηθισμένη αρχιτεκτονική FPGA αποτελείται από μια σειρά λογικών μπλοκ (ονομάζεται προγραμματιζόμενο λογικό μπλοκ, CLB ή λογικό μπλοκ πίνακα, LAB, ανάλογα με τον προμηθευτή), τα υποθέματα εισόδου / εξόδου και τα κανάλια δρομολόγησης. Γενικά, όλα τα κανάλια δρομολόγησης έχουν το ίδιο πλάτος (αριθμός καλωδίων). Τα πολλαπλά μαξιλάρια εισόδου / εξόδου μπορούν να χωρέσουν στο ύψος μιας σειράς ή στο πλάτος μίας στήλης στον πίνακα.

Ένα κύκλωμα εφαρμογής πρέπει να χαρτογραφηθεί σε ένα FPGA με επαρκείς πόρους. Ενώ ο αριθμός των CLB / LAB και I / O που απαιτούνται προσδιορίζεται εύκολα από το σχεδιασμό, ο αριθμός των διαδρομών δρομολόγησης που απαιτείται μπορεί να διαφέρει σημαντικά ακόμη και μεταξύ των σχεδίων με το ίδιο ποσό λογικής. Για παράδειγμα, ένας διακόπτης εγκάρσιας γραμμής απαιτεί πολύ περισσότερη δρομολόγηση από έναν συστολικό πίνακα με τον ίδιο αριθμό πυλών. Δεδομένου ότι τα αχρησιμοποίητα κομμάτια δρομολόγησης αυξάνουν το κόστος (και μειώνουν την απόδοση) του τμήματος χωρίς να παρέχουν οφέλη, οι κατασκευαστές FPGA προσπαθούν να παρέχουν αρκετά κομμάτια έτσι ώστε τα περισσότερα σχέδια να ταιριάζουν με τους πίνακες αναζήτησης (LUTs), ώστε να δρομολογηθούν. Αυτό καθορίζεται από εκτιμήσεις όπως αυτές που προκύπτουν από τον κανόνα του Rent ή με πειράματα στα υπάρχοντα σχέδια.

Γενικά, ένα λογικό μπλοκ (CLB ή LAB) αποτελείται από μερικά λογικά κελιά (που ονομάζονται ALM, LE, slice κλπ.). Ένα τυπικό κελί αποτελείται από ένα 4-εισόδου LUT, έναν πλήρες αθροιστή (FA) και ένα flip-flop τύπου D , όπως φαίνεται παρακάτω.



Οι LUTs σε αυτό το σχήμα χωρίζονται σε δύο LUTs 3 εισόδων. Στην *κανονική λειτουργία* αυτές συνδυάζονται σε ένα LUT 4 εισόδων μέσω του αριστερού mux . Στην *αριθμητική*, οι εκροές τους τροφοδοτούνται στον FA. Η επιλογή της λειτουργίας προγραμματίζεται στο μεσαίο πολυπλέκτη. Η έξοδος μπορεί να είναι συγχρονισμένη ή ασύγχρονη, ανάλογα με τον προγραμματισμό του mux προς τα δεξιά, στο παράδειγμα της εικόνας. Στην πράξη, ολόκληρα ή τμήματα του FA τοποθετούνται ως λειτουργίες στα LUT προκειμένου να εξοικονομηθεί χώρος.

Σκληρά μπλοκ - Hardblocks

Οι σύγχρονες οικογένειες FPGA επεκτείνονται στις παραπάνω δυνατότητες ώστε να συμπεριλαμβάνουν λειτουργίες υψηλότερου επιπέδου που έχουν καθοριστεί στο πυρίτιο. Έχοντας αυτές τις κοινές λειτουργίες ενσωματωμένες στο πυρίτιο μειώνουν την απαιτούμενη περιοχή και δίνουν αυτές τις λειτουργίες με αυξημένη ταχύτητα σε σύγκριση με την κατασκευή τους από αρχέτυπα. Παραδείγματα αυτών περιλαμβάνουν τους πολλαπλασιαστές, τα γενικά μπλοκ DSP, τους ενσωματωμένους επεξεργαστές, την λογική I / O υψηλής ταχύτητας και τις ενσωματωμένες μνήμες.

Τα FPGA υψηλότερου σημείου μπορούν να περιέχουν πομποδέκτες υψηλής ταχύτητας πολλαπλών gigabit και *σκληρούς πυρήνες IP* όπως πυρήνες επεξεργαστών, Ethernet MAC , PCI / PCI Express ελεγκτές και εξωτερικούς ελεγκτές μνήμης. Αυτοί οι πυρήνες υπάρχουν παράλληλα με την προγραμματιζόμενη λογική, αλλά είναι κατασκευασμένα από τρανζίστορ αντί για LUT, έτσι ώστε να έχουν απόδοση επιπέδου ASIC και κατανάλωση ισχύος, ενώ δεν καταναλώνουν σημαντικό όγκο

υλικών, αφήνοντας περισσότερο από το υλοποιήσιμο χώρο ελεύθερο για τη συγκεκριμένη εφαρμογή λογικής. Οι πομποδέκτες πολλαπλών gigabit περιέχουν επίσης κυκλώματα αναλογικής εισόδου και εξόδου υψηλής απόδοσης μαζί με σειριοποιητές και αποσειριοποιητές υψηλής ταχύτητας, τα οποία δεν μπορούν να κατασκευαστούν από LUTs. Οι λειτουργίες επιπέδου PHY υψηλότερου επιπέδου, όπως η κωδικοποίηση γραμμής, μπορεί να είναι ή να μην εφαρμόζονται παράλληλα με τους σειριοποιητές και τους αποσειριοποιητές στην σκληρή λογική, ανάλογα με το FPGA.

Clocking

Το μεγαλύτερο μέρος του κυκλώματος που είναι ενσωματωμένο σε ένα FPGA είναι ένα σύγχρονο κύκλωμα που απαιτεί σήμα ρολογιού. Τα FPGA περιέχουν αποκλειστικά δίκτυα δρομολόγησης για το ρολόι και την επαναφορά, ώστε να μπορούν να παραδοθούν σε ελάχιστο χρόνο. Επίσης, τα FPGA περιέχουν γενικά αναλογικά εξαρτήματα PLL και / ή DLL για να συνθέσουν νέες συχνότητες ρολογιού καθώς και να εξασθενήσουν το jitter. Τα σύνθετα σχέδια μπορούν να χρησιμοποιούν πολλαπλά ρολόγια με διαφορετικές σχέσεις συχνότητας και φάσης, όπου κάθε μία χωρίζει ξεχωριστά πεδία ρολογιού. Αυτά τα σήματα ρολογιού μπορούν να παραχθούν τοπικά από έναν ταλαντωτή ή μπορούν να ανακτηθούν από ένα ρεύμα σειριακών δεδομένων υψηλής ταχύτητας. Πρέπει να λαμβάνεται μέριμνα κατά τη διέλευση του τομέα του ρολογιού για να αποφευχθεί η μεταστατοποίηση. Τα FPGAs γενικά περιέχουν μπλοκ μνήμης RAM που είναι ικανές να λειτουργούν ως RAM διπλής θύρας με διαφορετικά ρολόγια, βοηθώντας στην κατασκευή FIFOs και buffers διπλής θύρας που συνδέουν διαφορετικούς τομείς ρολογιών.

Τρισδιάστατες αρχιτεκτονικές - 3D architectures

Για να συρρικνωθεί το μέγεθος και η κατανάλωση ισχύος των FPGA, οι Tabula και Xilinx έχουν εισαγάγει αρχιτεκτονικές 3D ή στοίβα. Μετά την εισαγωγή των σειριακών FPGA των 28nm, η Xilinx δήλωσε ότι πολλά από τα τμήματα υψηλότερης πυκνότητας σε αυτές τις γραμμές προϊόντων FPGA θα κατασκευαστούν χρησιμοποιώντας πολλαπλές μήτρες σε ένα πακέτο, χρησιμοποιώντας τεχνολογία που αναπτύχθηκε για 3D κατασκευές.

Η προσέγγιση του Xilinx στοιβάζει αρκετά (τρία ή τέσσερα) ενεργά FPGA δίπλα-δίπλα σε μια παρεμβολή πυριτίου - ένα ενιαίο κομμάτι πυριτίου που φέρει παθητική διασύνδεση. Η κατασκευή πολλαπλών πετάλων επιτρέπει επίσης τη δημιουργία διαφορετικών τμημάτων του FPGA με διαφορετικές τεχνολογίες επεξεργασίας, καθώς οι απαιτήσεις διεργασίας διαφέρουν μεταξύ του ίδιου του FPGA και των υψηλής ταχύτητας σειριακών πομποδεκτών 28 Gbit / s. Ένα FPGA που χτίστηκε με αυτόν τον τρόπο ονομάζεται *ετερογενής FPGA*.

Η ετερογενής προσέγγιση της Altera προϋποθέτει τη χρήση μιας μονής μονολιθικής μήτρας FPGA και τη σύνδεση άλλων τεχνολογιών μήτρας με το FPGA χρησιμοποιώντας την ενσωματωμένη τεχνολογία γέφυρας διασύνδεσης πολλαπλών πετάλων (EMIB) της Intel.

Σχεδιασμός και προγραμματισμός

Για να καθορίσει τη συμπεριφορά του FPGA, ο χρήστης παρέχει ένα σχέδιο σε μια γλώσσα περιγραφής υλικού (HDL) ή ως σχηματική σχεδίαση. Η μορφή HDL είναι πιο κατάλληλη για να δουλέψει με μεγάλες δομές επειδή είναι δυνατόν να τις ορίσει μόνο αριθμητικά παρά να χρειαστεί να σχεδιάσει κάθε κομμάτι με το χέρι. Ωστόσο, η σχηματική εισαγωγή μπορεί να επιτρέψει την ευκολότερη απεικόνιση ενός σχεδίου.

Στη συνέχεια, χρησιμοποιώντας ένα εργαλείο αυτοματισμού ηλεκτρονικού σχεδιασμού , δημιουργείται μια τεχνολογία χαρτογραφημένη netlist. Το netlist μπορεί στη συνέχεια να προσαρμοστεί στην πραγματική αρχιτεκτονική FPGA χρησιμοποιώντας μια διαδικασία που ονομάζεται θέση-και-διαδρομή , που συνήθως εκτελείται από το λογισμικό τοποθεσίας και διαδρομής της εταιρείας FPGA. Ο χρήστης θα επικυρώσει τα αποτελέσματα στο χάρτη, τόπου και διαδρομής μέσω ανάλυσης χρονισμού , προσομοίωσης και άλλων μεθοδολογιών επαλήθευσης. Μόλις ολοκληρωθεί η διαδικασία σχεδιασμού και επικύρωσης, το δυαδικό αρχείο που δημιουργείται, χρησιμοποιείται για να επαναρυθμίσει το FPGA. Αυτό το αρχείο μεταφέρεται στο FPGA / CPLD μέσω μίας σειριακής διασύνδεσης (JTAG) ή σε μια εξωτερική συσκευή μνήμης όπως μια EEPROM.

Η πιο συνηθισμένη γλώσσα HDL είναι η VHDL και η Verilog, αν και σε μια προσπάθεια να μειωθεί η πολυπλοκότητα του σχεδιασμού σε HDL, τα οποία έχουν συγκριθεί με το ισοδύναμο των γλωσσών συναρμολόγησης, υπάρχουν κινήσεις για να αυξήσει το επίπεδο αφαίρεσης μέσω της εισαγωγής εναλλακτικών γλωσσών . Η γραφική γλώσσα προγραμματισμού LabVIEW της NationalInstruments (μερικές φορές αναφέρεται ως "G") διαθέτει μια πρόσθετη ενότητα FPGA που είναι διαθέσιμη για να στοχεύσει και να προγραμματίσει το υλικό FPGA.

Για να απλοποιηθεί ο σχεδιασμός σύνθετων συστημάτων σε FPGAs, υπάρχουν βιβλιοθήκες προκαθορισμένων πολύπλοκων λειτουργιών και κυκλωμάτων που έχουν δοκιμαστεί και βελτιστοποιηθεί για να επιταχύνουν τη διαδικασία σχεδιασμού. Αυτά τα προκαθορισμένα κυκλώματα ονομάζονται συνήθως *πυρήνες IP* και διατίθενται από προμηθευτές FPGA και προμηθευτές IP. Άλλα προκαθορισμένα κυκλώματα διατίθενται από κοινότητες προγραμματιστών, όπως το OpenCores (συνήθως κυκλοφορούν υπό άδειες ελεύθερης και ανοικτής πηγής όπως η άδεια GPL , BSD ή παρόμοια άδεια) και άλλες πηγές.

Σε μια τυπική ροή σχεδιασμού, ένας προγραμματιστής εφαρμογών FPGA θα προσομοιώσει τον σχεδιασμό σε πολλαπλά στάδια καθ 'όλη τη διάρκεια του σχεδιασμού. Αρχικά, η περιγραφή RTL σε VHDL ή Verilog προσομοιώνεται με τη δημιουργία πινάκων δοκιμών για την προσομοίωση του συστήματος και την παρακολούθηση των αποτελεσμάτων. Στη συνέχεια, αφού ο μηχανισμός σύνθεσης έχει χαρτογραφήσει τον σχεδιασμό σε μια netlist, η netlist μεταφράζεται σε μια περιγραφή στάθμης πύλης, όπου η προσομοίωση επαναλαμβάνεται για να επιβεβαιωθεί ότι η σύνθεση προχώρησε χωρίς σφάλματα. Τέλος, ο σχεδιασμός έχει καθοριστεί στο FPGA στο οποίο σημείο καθυστερήσεις διάδοσης μπορούν να προστεθεί και η προσομοίωση τρέχει και πάλι με αυτές τις τιμές πίσω στο netlist.

Πιο πρόσφατα, το OpenCL χρησιμοποιείται από τους προγραμματιστές για να εκμεταλλευτεί τις επιδόσεις και την αποδοτικότητα ισχύος που παρέχουν τα FPGAs.

Το OpenCL επιτρέπει στους προγραμματιστές να αναπτύξουν κώδικα στη γλώσσα προγραμματισμού C και να στοχεύσουν τις λειτουργίες FPGA ως πυρήνες OpenCL χρησιμοποιώντας δομές OpenCL.

Βασικοί τύποι τεχνολογίας διεργασιών

- SRAM - με βάση την τεχνολογία στατικής μνήμης. Είναι προγραμματιζόμενη και επαναπρογραμματιζόμενη εντός του συστήματος. Απαιτεί συσκευές εξωτερικής εκκίνησης - CMOS. Αξίζει να σημειωθεί ότι οι συσκευές flash ή EEPROM μπορεί συχνά να φορτώνουν τα περιεχόμενα σε εσωτερική SRAM που ελέγχει τη δρομολόγηση και τη λογική.
- Ασφάλεια - Προγραμματιζόμενη μία φορά. Διπολική. Απαρχαιωμένη.
- Antifuse - Προγραμματιζόμενη μία φορά. CMOS.
- PROM - Προγραμματιζόμενη τεχνολογία μνήμης μόνο για ανάγνωση. Εφάπαξ προγραμματιζόμενη λόγω πλαστικής συσκευασίας. Απαρχαιωμένη.
- EPROM - Διαγραφή προγραμματιζόμενης τεχνολογίας μνήμης μόνο για ανάγνωση. Ενίοτε προγραμματιζόμενη αλλά με παράθυρο, μπορεί να διαγραφεί με υπεριώδες (UV) φως. CMOS. Απαρχαιωμένη.
- EEPROM - Ηλεκτρικά διαγράψιμη προγραμματιζόμενη τεχνολογία μνήμης μόνο ανάγνωσης. Μπορεί να διαγραφεί, ακόμη και σε πλαστικές συσκευασίες. Ορισμένες αλλά όχι όλες οι συσκευές EEPROM μπορούν να προγραμματιστούν εντός του συστήματος. CMOS.
- Flash - Σβήνει την τεχνολογία EPROM. Μπορεί να διαγραφεί, ακόμη και σε πλαστικές συσκευασίες. Κάποιες αλλά όχι όλες οι συσκευές flash μπορούν να προγραμματιστούν εντός του συστήματος. Συνήθως, μια κυψέλη flash είναι μικρότερη από ένα ισοδύναμο κύτταρο EEPROM και συνεπώς είναι λιγότερο δαπανηρή στην κατασκευή. CMOS.

Σημαντικοί κατασκευαστές

Το 2016, οι μακροχρόνιοι ανταγωνιστές της βιομηχανίας Xilinx και Altera (τώρα Intel) ήταν οι ηγέτες της αγοράς FPGA. Την εποχή εκείνη, ελέγχονταν σχεδόν το 90% της αγοράς.

Τόσο η Xilinx όσο και η Altera παρέχουν ιδιωτικό λογισμικό σχεδιασμού σε Windows και Linux (ISE / Vivado και Quartus) το οποίο επιτρέπει στους μηχανικούς να σχεδιάζουν, να αναλύουν, να προσομοιώνουν και να συνθέτουν τα σχέδιά τους.

Άλλοι κατασκευαστές είναι:

- Microsemi (προηγουμένως Actel , αντιπληροφόρηση , αναμετάδοση flash, μικτό σήμα)
- Lattice Semiconductor (SRAM που βασίζεται σε ενσωματωμένο φλας, instant-on, χαμηλής ισχύος, ζωντανή αναδιάταξη)
  - SiliconBlue Technologies (FPGA με εξαιρετικά χαμηλή ισχύ SRAM με προαιρετική ενσωματωμένη μνήμη μη διαθεσίμων ρυθμίσεων, που αποκτήθηκε από το Lattice το 2011)

- QuickLogic (Πολύ χαμηλής ισχύος αισθητήρας Hubs, εξαιρετικά χαμηλής ισχύος, χαμηλής πυκνότητας FPGA με βάση SRAM, Γέφυρες οθόνης εισόδους MIPI & RGB, εξόδους MIPI, RGB και LVDS)
- Atmel (δεύτερη πηγή κάποιων συσκευών συμβατών με την Altera, επίσης FPSLIC που αναφέρθηκε παραπάνω, που αποκτήθηκε από τη Microchip το 2016))
- Achronix (βασισμένη σε SRAM, ταχύτητα υφασμάτων 1,5 GHz)

Τον Μάρτιο του 2010, η Tabula ανήγγειλε την τεχνολογία FPGA που χρησιμοποιεί πολυπλεξία λογικής χρόνου και διασύνδεσης που απαιτεί πιθανή εξοικονόμηση κόστους για εφαρμογές υψηλής πυκνότητας. Στις 24 Μαρτίου 2015, η Tabula έκλεισε επισήμως.

Την 1η Ιουνίου 2015, η Intel ανακοίνωσε ότι θα αποκτήσει την Altera για περίπου 16,7 δισεκατομμύρια δολάρια και ολοκλήρωσε την εξαγορά στις 30 Δεκεμβρίου 2015.

VHDL

Μια δήλωση οντότητας ή οντότητα, σε συνδυασμό με την αρχιτεκτονική ή το σώμα, αποτελεί ένα VHDL μοντέλο. Η VHDL ονομάζει το ζεύγος οντότητα-αρχιτεκτονική ως μία οντότητα σχεδιασμού. Περιγράφοντας την εναλλακτική λύση αρχιτεκτονικών για μια οντότητα, μπορούμε να διαμορφώσουμε ένα μοντέλο VHDL για ένα συγκεκριμένο επίπεδο έρευνας. Η οντότητα περιέχει την περιγραφική διεπαφή που είναι κοινή με την εναλλακτική λύση των αρχιτεκτονικών. Οι γενικές πληροφορίες ορίζουν μια οντότητα, καθορίζοντας στο περιβάλλον σταθερές όπως το μέγεθος ή η τιμή καθυστέρησης. Για παράδειγμα,

```
entityA is
        port (x, y: in real; z: out real);
        generic (delay: time);
endA;
```

Η αρχιτεκτονική περιλαμβάνει τις ενότητες δήλωσης. Δηλώσεις στη περιοχή πριν τη δεσμευμένη λέξη **begin** μπορούν να περιγράψουν τοπικά στοιχεία όπως σήματα και εξαρτήματα(components). Οι δηλώσεις εμφανίζονται μετά την λέξη **begin** και μπορούν να περιέχουν ταυτόχρονες δηλώσεις. Για παράδειγμα,

```
architecture B of A is
component M
port ( j : in real ; k : out real);
end component;
signala,b,c real := 0.0;
begin
"concurrent statements"
end B;
```

Η ποικιλία των ταυτόχρονων τύπων δήλωσης δίνει στη VHDL την περιγραφική δυνατότητα να δημιουργεί και να συνδυάζει μοντέλα όπως δομικό (structural), ροής δεδομένων (dataflows) και επίπεδα συμπεριφοράς (behanioral levels) σε ένα μοντέλο προσομοίωσης. Ο δομικός τύπος (structural) περιγραφής χρησιμοποιεί την παράσταση δομικών δηλώσεων και μπορεί να αναφέρει μοντέλα που περιγράφονται αλλού. Μετά τη δήλωση των στοιχείων (components), τα χρησιμοποιούμε στη δήλωση συστατικού συμβάντος, αναθέτοντας θύρες σε τοπικά σήματα ή σε άλλα θύρες και δίνοντας τιμές σε αυτά που δημιουργούνται. invert: M portmap (j => a ; **k** =>***c);***Μπορούμε να συνδυάσουμε τα στοιχεία (components) σε άλλες οντότητες σχεδιασμού μέσω των προδιαγραφών διαμόρφωσης στο τμήμα δήλωσης αρχιτεκτονικής της VHDL ή μέσω χωριστών δηλώσεων διαμόρφωσης. Η ροή δεδομένων (dataflow) χρησιμοποιεί ευρέως έναν αριθμό από τύπους δηλώσεων ταυτόχρονης εκχώρησης σήματος, οι οποίες συνδέουν ένα σήμα στόχου με μια έκφραση (expression) και μια καθυστέρηση. Η λίστα των σημάτων που εμφανίζεται στην έκφραση είναι η λίστα ευαισθησίας (sensitivitylist), όπου η έκφραση πρέπει να αξιολογηθεί για οποιαδήποτε αλλαγή σε οποιοδήποτε από αυτά τα σήματα. Τα σήματα στόχου (target signals) αποκτούν νέες τιμές μετά τη καθυστέρηση που καθορίζεται στη δήλωση καταχώρησης των σημάτων. Εάν δεν έχει καθοριστεί

καθυστέρηση, η ανάθεση του σήματος πραγματοποιείται κατά τον επόμενο κύκλο προσομοίωσης:

```
c <= a + b afterdelay
```

Η VHDL περιλαμβάνει επίσης δηλώσεις καταχώρησης σήματος conditional και selected. Χρησιμοποιεί ένα μπλοκ δηλώσεων για την ομαδοποίηση εντολών καταχώρησης σήματος και το καθιστά σύγχρονο με μία προστατευμένη κατάσταση. Οι εντολές μπλοκ μπορούν επίσης να περιέχουν θύρες και δηλώσεις για να παρέχουν περισσότερα στοιχεία στις περιγραφές. Συνήθως χρησιμοποιούμε παράλληλες δηλώσεις διαδικασιών όταν θέλουμε να περιγράψουμε το υλικό σε ένα αφαιρετικό επίπεδο συμπεριφοράς. Η διαδικασία της δήλωσης αποτελείται από δηλώσεις και τύπους δηλώσεων που συνθέτουν το διαδοχικό πρόγραμμα. Δηλώσεις τύπου waitκαι assertπροσθέτουν στην περιγραφή δηλώσεις διαδικασίας για μοντελοποίηση ταυτόχρονων ενεργειών:

```
process
begin
variablei : real := 1.0;
wait on a;
i = b * 3.0;
c <= i after delay;
end process;
```

Άλλες παράλληλες δηλώσεις περιλαμβάνουν την ταυτόχρονη δήλωση ισχυρισμού (concurrent assertion statement), ταυτόχρονη διαδικασία κλήσης (concurrent procedure call), και δημιουργία δηλώσεων (generate statement). Τα πακέτα είναι μονάδες σχεδιασμού που επιτρέπουν τύπους και αντικείμενα προς κοινή χρήση. Οι αριθμητικές λειτουργίες κυριαρχούν στον χρόνο εκτέλεσης των περισσότερων ψηφιακών Αλγορίθμων Επεξεργασίας Ψηφιακών Σημάτων (DSP) και προς το παρόν ο χρόνος που απαιτείται για την εκτέλεση ενός πολλαπλασιασμού παραμένει ο κυρίαρχος παράγοντας στον καθορισμό του κύκλου χρόνου ενός τσιπ DSP και των υπολογιστών μειωμένων συνόλων εντολών (RISC – Reduced Instruction Set Computers). Μεταξύ των πολλών μεθόδων εφαρμογής παράλληλων πολλαπλασιαστών υψηλής ταχύτητας, υπάρχει μια βασική προσέγγιση, δηλαδή τον αλγόριθμο Booth.

Η κατανάλωση ενέργειας σε ένα DSP VLSI έχει αποκτήσει ιδιαίτερη προσοχή λόγω της εξάπλωσης του σε υψηλής απόδοσης φορητές ηλεκτρονικές συσκευές που λειτουργούν με μπαταρία, όπως κινητά τηλέφωνα, φορητοί υπολογιστές, κ.λπ. Οι εφαρμογές DSP απαιτούν υψηλή υπολογιστική ταχύτητα και την ίδια στιγμή, υποφέρουν από αυστηρούς περιορισμούς ισχύος.
Οι μονάδες πολλαπλασιασμού είναι κοινές σε πολλές εφαρμογές DSP. Οι ταχύτεροι τύποι πολλαπλασιαστών είναι οι παράλληλοι πολλαπλασιαστές. Μεταξύ αυτών, ο πολλαπλασιαστής Wallace είναι ο ταχύτερος. Ωστόσο, υποφέρει από κακή τακτικότητα. Ως εκ τούτου, όταν οι επιδόσεις και η χαμηλή ισχύς είναι πρωταρχικές σκέψεις, οι πολλαπλασιαστές τύπου Booth τείνουν να είναι η πρωταρχική επιλογή.

Οι Boothπολλαπλασιαστές επιτρέπουν τη λειτουργία σε υπογεγραμμένους τελεστές στο 2-συμπληρωματικό. Προέρχονται από πολλαπλασιαστές συστοιχιών (array multipliers) όπου, για κάθε bit σε μία μερική σειρά, υπάρχει ένα σχήμα κωδικοποίησης που χρησιμοποιείται για να προσδιορίσει εάν αυτό το δυαδικό ψηφίο είναι θετικό, αρνητικό ή μηδέν. Ο τροποποιημένος αλγόριθμος Booth επιτυγχάνει σημαντική βελτίωση απόδοσης μέσω της κωδικοποίησης radix-4. Σε αυτόν τον αλγόριθμο κάθε μερική σειρά λειτουργεί με 2 bits κάθε φορά, μειώνοντας έτσι τον συνολικό αριθμό των υπολειπόμενων bit. Αυτό ισχύει ιδιαίτερα για τελεστές που χρησιμοποιούν 16 bits ή περισσότερα.

Xilinx ISE

Το **Xilinx ISE** (**I**ntegrated **S** ynthesis **E** nvironment) είναι ένα εργαλείο λογισμικού που παράγεται από την Xilinx για τη σύνθεση και την ανάλυση των HDL σχεδίων, επιτρέποντας στον προγραμματιστή να συνθέσει («μεταγλώττισει») τα σχέδιά του, να εκτελεί την ανάλυση χρονισμού , να εξετάζει τα RTL διαγράμματα, να προσομοιώνει την αντίδραση ενός σχεδίου σε διαφορετικά ερεθίσματα και να ρυθμίζει τη συσκευή προορισμού με τον προγραμματιστή.

Το Xilinx ISE είναι ένα περιβάλλον σχεδιασμού για προϊόντα FPGA από την Xilinx και είναι στενά συνδεδεμένο με την αρχιτεκτονική τέτοιων τσιπ και δεν μπορεί να χρησιμοποιηθεί με προϊόντα FPGA άλλων προμηθευτών. Το Xilinx ISE χρησιμοποιείται κυρίως για τη σύνθεση και το σχεδιασμό κυκλωμάτων, ενώ το ISIM ή ο εξομοιωτής προσομοίωσης ModelSim χρησιμοποιείται για δοκιμές σε επίπεδο συστήματος.Άλλα στοιχεία που συνοδεύουν το Xilinx ISE περιλαμβάνουν το Embedded Development Kit (EDK), ένα κιτ ανάπτυξης λογισμικού (SDK) και το ChipScopePro.

Από το 2012, το Xilinx ISE έχει αντικατασταθεί από το Vivado Design Suite , το οποίο εξυπηρετεί τους ίδιους ρόλους με το ISE με πρόσθετα χαρακτηριστικά για το σύστημα σε ανάπτυξη τσιπ . Το Xilinx κυκλοφόρησε την τελευταία έκδοση του ISE τον Οκτώβριο του 2013 (έκδοση 14.7) και δηλώνει ότι "το ISE έχει μεταφερθεί στη φάση συντήρησης του κύκλου ζωής του προϊόντος και δεν υπάρχουν πλέον προγραμματισμένες εκδόσεις ISE".

User Interface

Το πρωτεύον περιβάλλον εργασίας ενός χρήστη του ISE είναι το Project Navigator, το οποίο περιλαμβάνει την ιεραρχία σχεδιασμού (Πηγές), έναν επεξεργαστή πηγαίου κώδικα (Workplace), μια κονσόλα εξόδου (Transcript) και ένα δέντρο διεργασιών (Processes).

Η ιεραρχία σχεδιασμού αποτελείται από αρχεία σχεδιασμού (modules), των οποίων οι εξάρσεις ερμηνεύονται από το ISE και εμφανίζονται ως μία δομή δέντρου. Για το σχέδιο ενός τσιπ μπορεί να υπάρχει μία κύρια μονάδα, με άλλες ενότητες που περιλαμβάνονται στην κύρια μονάδα, παρόμοια με την main() υπορουτίνα στα προγράμματα C ++. Οι περιορισμοί σχεδιασμού καθορίζονται σε ενότητες, οι οποίες περιλαμβάνουν τη διαμόρφωση και την αντιστοίχιση συνδέσεων (pins).

Η ιεραρχία των διεργασιών περιγράφει τις λειτουργίες που θα εκτελέσει το ISE στην τρέχουσα ενεργή ενότητα. Η ιεραρχία περιλαμβάνει τις λειτουργίες συμπίεσης, τις λειτουργίες εξάρτησης και άλλες υπηρεσίες κοινής ωφέλειας. Το παράθυρο υποδεικνύει επίσης ζητήματα ή σφάλματα που προκύπτουν με κάθε λειτουργία.

Το παράθυρο "Transcript" παρέχει κατάσταση των λειτουργιών που εκτελούνται αυτήν τη στιγμή και ενημερώνει τους μηχανικούς για θέματα σχεδίασης. Αυτά τα θέματα μπορούν να φιλτραριστούν για να εμφανίσουν Προειδοποιήσεις, Λάθη ή και τα δύο.

Προσομοίωση

Οι δοκιμές σε επίπεδο συστήματος μπορούν να πραγματοποιηθούν με ISIM ή τον προσομοιωτή λογικής ModelSim και προγράμματα τέτοιου είδος δοκιμών πρέπει επίσης να είναι γραμμένα σε γλώσσες HDL. Τα προγράμματα δοκιμών βάσης μπορούν να περιλαμβάνουν προσομοιωμένες κυματομορφές σήματος εισόδου ή οθόνες που παρατηρούν και επαληθεύουν τις εξόδους της υπό δοκιμή της συσκευής .

Το ModelSim ή το ISIM μπορεί να χρησιμοποιηθεί για την εκτέλεση των ακόλουθων προσομοιώσεων:

- Λογική επαλήθευση, για να διασφαλιστεί ότι η ενότητα θα παράγει τα αναμενόμενα αποτελέσματα
- Επαλήθευση συμπεριφοράς, για να επαληθεύσετε τα θέματα λογικής και χρονισμού
- Μετά την τοποθέτηση και τη προσομοίωση διαδρομής, για να επαληθεύσετε τη συμπεριφορά μετά την τοποθέτηση της μονάδας μέσα στην αναμορφώσιμη λογική του FPGA

Σύνθεση

Οι κατοχυρωμένοι με δίπλωμα ευρεσιτεχνίας αλγόριθμοι της Xilinx για τη σύνθεση επιτρέπουν στα σχέδια να τρέχουν έως και 30% ταχύτερα από τα ανταγωνιστικά προγράμματα και επιτρέπουν μεγαλύτερη λογική πυκνότητα η οποία μειώνει το χρόνο και το κόστος του έργου.

Επίσης, εξαιτίας της αυξανόμενης πολυπλοκότητας του FPGA, συμπεριλαμβανομένων των μπλοκ μνήμης και των μπλοκ I / O, αναπτύχθηκαν πιο περίπλοκοι αλγόριθμοι σύνθεσης που διαχωρίζουν μη σχετιζόμενες ενότητες σε *φέτες (slices)* , μειώνοντας τα λάθη μετά την τοποθέτηση.

Οι πυρήνες IP προσφέρονται από την Xilinx και από άλλους προμηθευτές τρίτων μερών, για την υλοποίηση λειτουργιών σε επίπεδο συστήματος όπως ψηφιακή επεξεργασία σήματος(DSP), διεπαφές διαύλου, πρωτόκολλα δικτύωσης, επεξεργασία εικόνας , ενσωματωμένοι επεξεργαστές και περιφερειακά συστήματα. Το Xilinx έχει συμβάλει στην αλλαγή των σχεδίων από την υλοποίηση ASIC με βάση την εφαρμογή FPGA.

Αθροιστές

Στα ηλεκτρονικά, ένας αθροιστής είναι ένα ψηφιακό κύκλωμα που εκτελεί την πρόσθεση αριθμών. Στους σύγχρονους υπολογιστές οι αθροιστές βρίσκονται στην αριθμητική λογική μονάδα (ALU) όπου οι πράξεις εκτελούνται. Παρόλο που οι αθροιστές μπορούν να κατασκευαστούν για πολλές αριθμητικές αναπαραστάσεις, όπως δυαδικά κωδικοποιημένα δεκαδικά ψηφία ή υπέρβαση-3 (excess-3), οι συνηθέστεροι αθροιστές λειτουργούν με δυαδικούς αριθμούς. Σε περιπτώσεις όπου το συμπλήρωμα ως προς δύο χρησιμοποιείται για να εκπροσωπεί τους αρνητικούς αριθμούς είναι ασήμαντο να τροποποιήσουμε έναν αθροιστή σε έναν αθροιστή-αφαιρέτη.

Τύποι αθροιστών

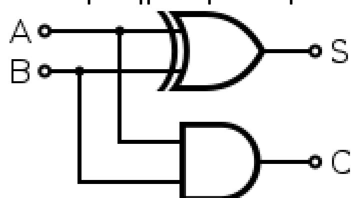
Για ενός δυαδικού ψηφίου αθροιστές, υπάρχουν δύο γενικοί τύποι.

Ο πρώτος τύπος είναι ο ημιαθροιστής (half-adder) έχει δύο εισόδους, γενικά τις *A* και *B* , και δύο εξόδους, το άθροισμα *S* και το κρατούμενο *C*. *Το S* είναι το XOR δύο-bit των *A* και *B* , και το *C* είναι το AND του *A* και *B* . Ουσιαστικά η έξοδος ενός ημιαθροιστή είναι το άθροισμα δύο αριθμών ενός δυαδικού ψηφίου, όπου το *C* είναι σημαντικότερο και από τις δύο εξόδους.

Ο πίνακας αληθείας τού ημιαθροιστή έχει ως εξής:

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

Κύκλωμα ημιαθροιστή



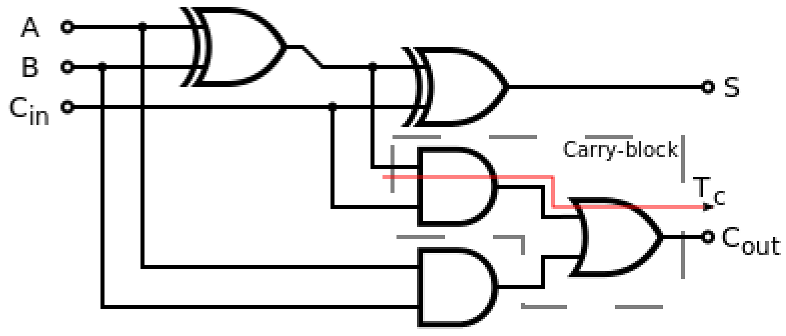
Ο δεύτερος τύπος αθροιστή ενός bit είναι ο πλήρης αθροιστής(fulladder). Ο πλήρης αθροιστής είναι το συνδυαστικό κύκλωμα που εκτελεί την πρόσθεση τριών δυαδικών αριθμών και, συγκεκριμένα, δύο σημαντικών και ενός κρατουμένου. Το κρατούμενο ενδέχεται να έχει παραχθεί από προηγούμενη άθροιση.

Έχει τρεις εισόδους A, B, Ci, που αποτελούν τους δυο προσθετέους και το προηγούμενο κρατούμενο. Οι δυο έξοδοι S, Co συμβολίζουν το άθροισμα και το νέο κρατούμενο.

Ο πίνακας αληθείας του πλήρη αθροιστή έχει ως εξής:

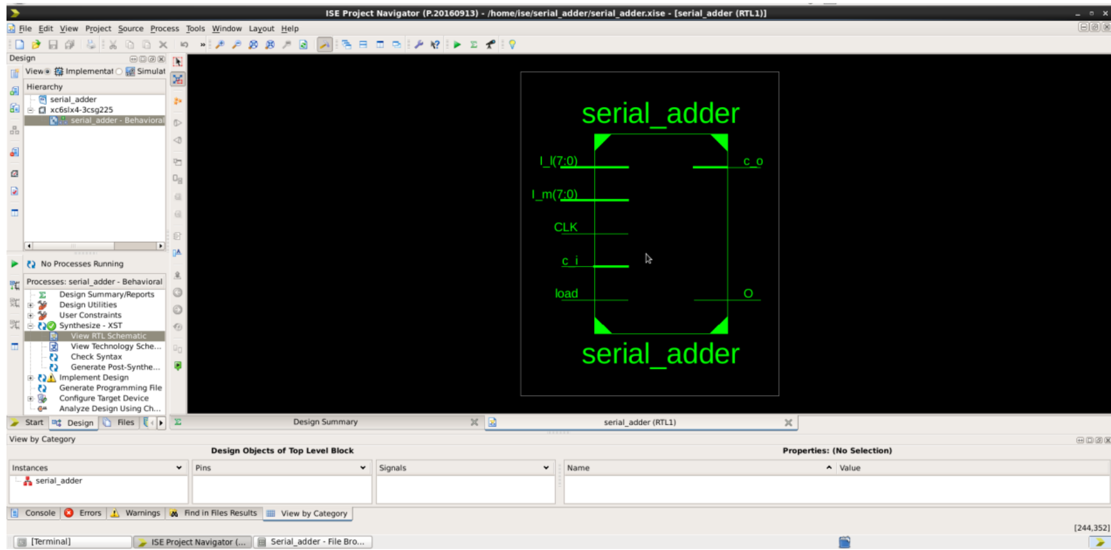
Input			Output	
A	B	C_i	C_o	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Κύκλωμα πλήρη αθροιστή

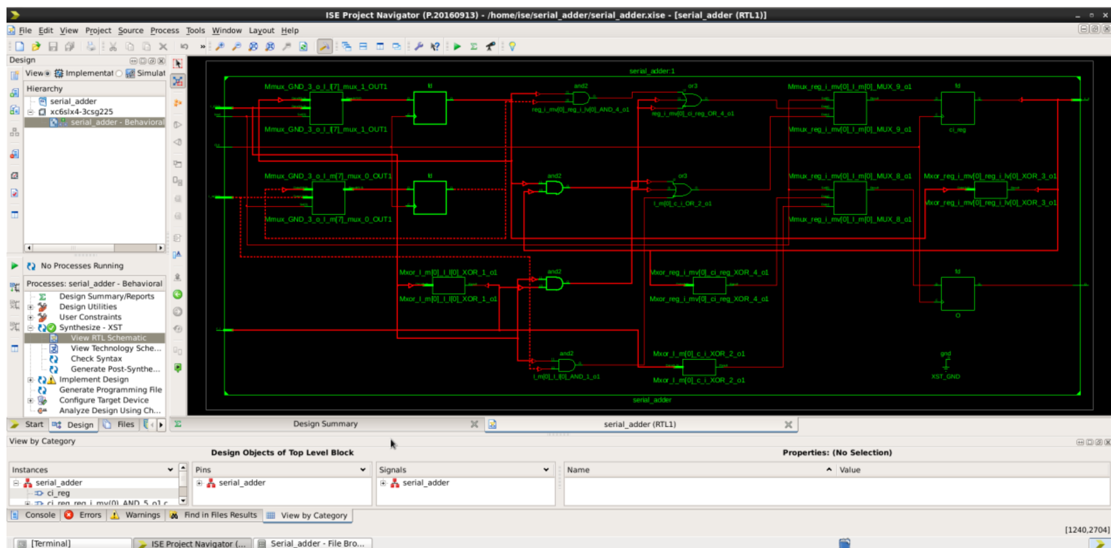


Εφαρμογή και Αποτελέσματα

Σειριακός αθροιστής (BitSerialAdder)



Εικόνα 1: RTL σχηματική (top-level block) - Bit Serial Adder



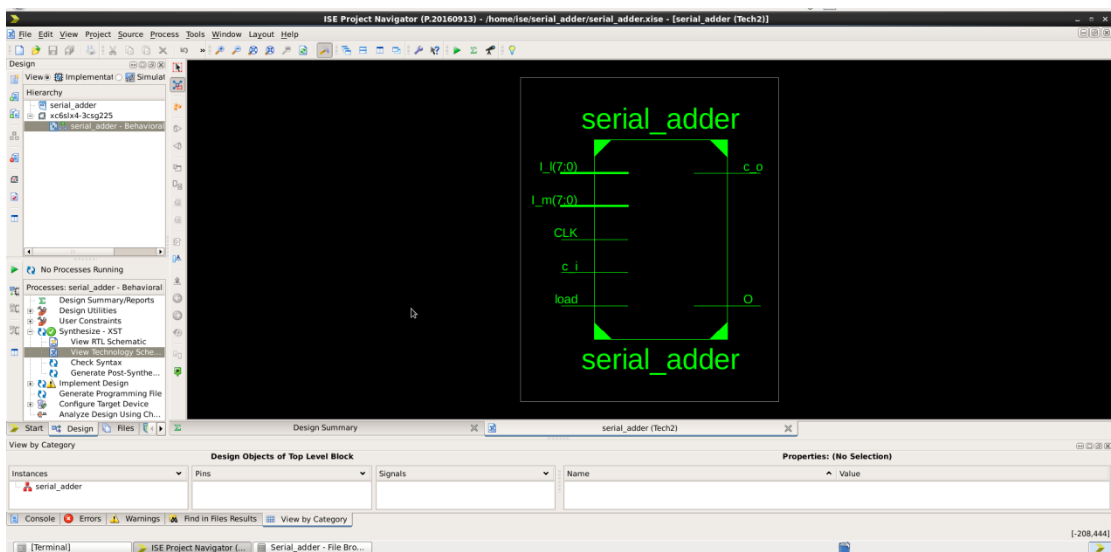
Εικόνα 2: RTL σχηματική - Bit Serial Adder

serial_adder Project Status (05/29/2018 - 21:14:38)			
Project File:	serial_adder.xise	**Parser Errors:**	No Errors
Module Name:	serial_adder	**Implementation State:**	Placed and Routed
Target Device:	xc6slx4-3csg225	**• Errors:**	No Errors
Product Version:	ISE 14.7	**• Warnings:**	No Warnings
Design Goal:	Balanced	**• Routing Results:**	All Signals Completely Routed
Design Strategy:	Xilinx Default (unlocked)	**• Timing Constraints:**	All Constraints Met
Environment:	System Settings	**• Final Timing Score:**	0 (Timing Report)

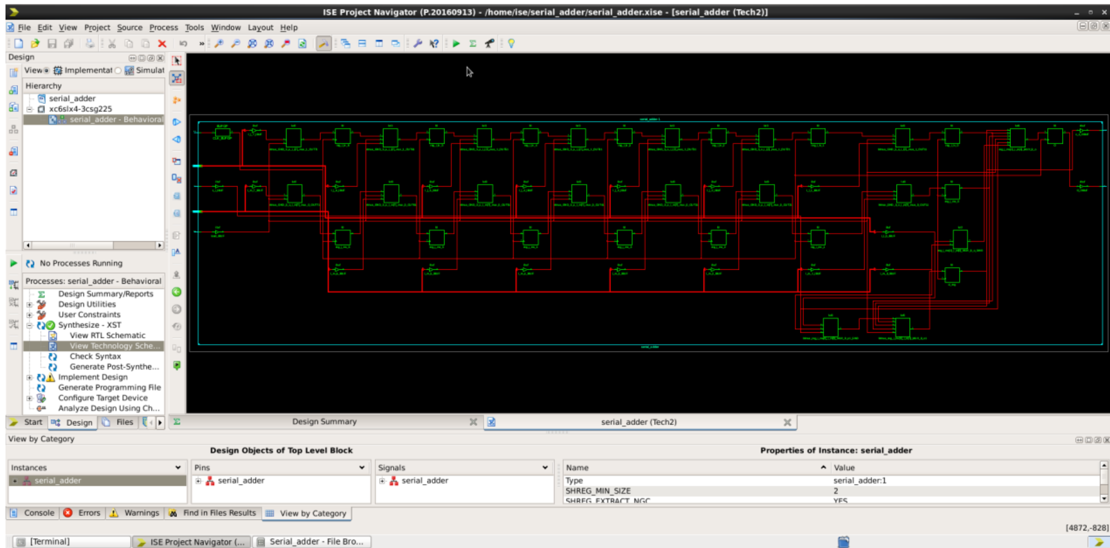
Device Utilization Summary

Slice Logic Utilization	Used	Available	Utilization	Note(s)
Number of Slice Registers	25	4,800	1%	
Number used as Flip Flops	25			
Number used as Latches	0			
Number used as Latch-thrus	0			
Number used as AND/OR logics	0			
Number of Slice LUTs	19	2,400	1%	
Number used as logic	16	2,400	1%	
Number using O6 output only	14			
Number using O5 output only	0			
Number using O5 and O6	2			
Number used as ROM	0			
Number used as Memory	0	1,200	0%	

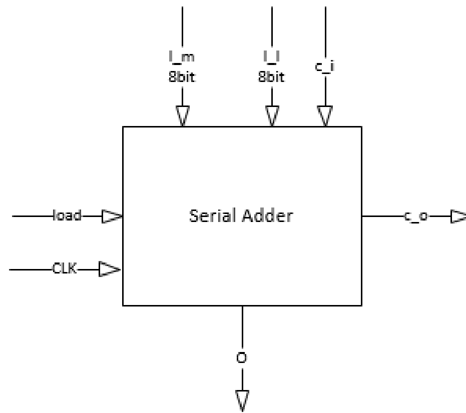
Εικόνα 3: Περίληψη σχεδίασης - Bit Serial Adder



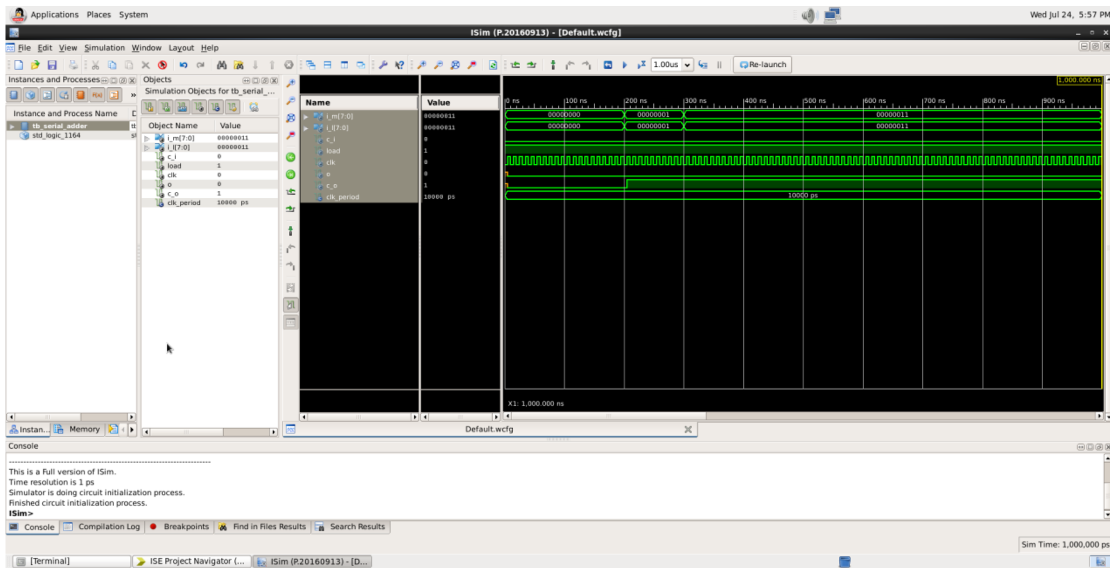
Εικόνα 4: Προβολή τεχνολογικών σχημάτων (top-level block) - Bit Serial Adder



Εικόνα 5: Προβολή τεχνολογικών σχημάτων - Bit Serial Adder

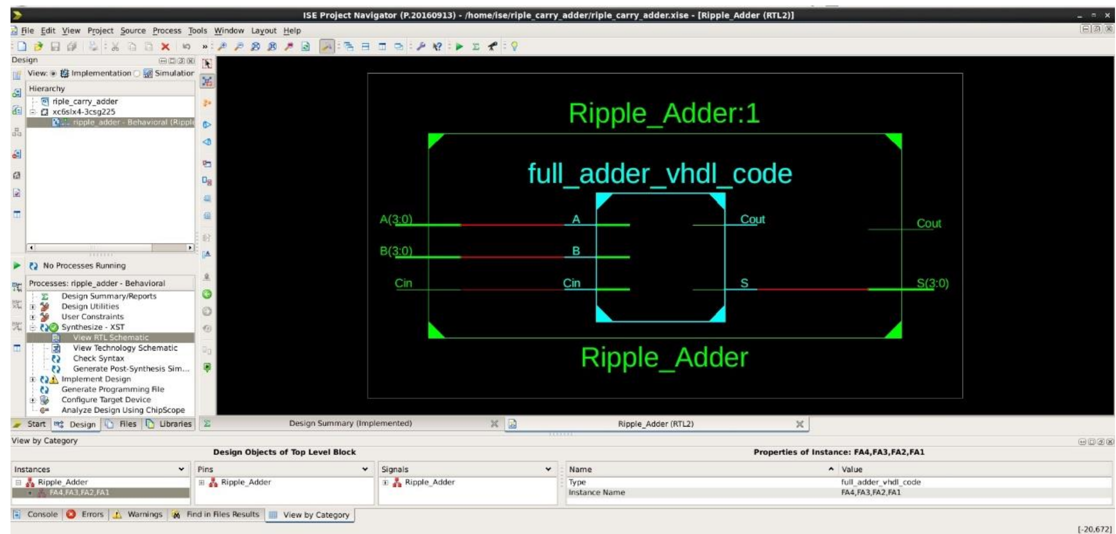


Εικόνα 6: Bit Serial Adder

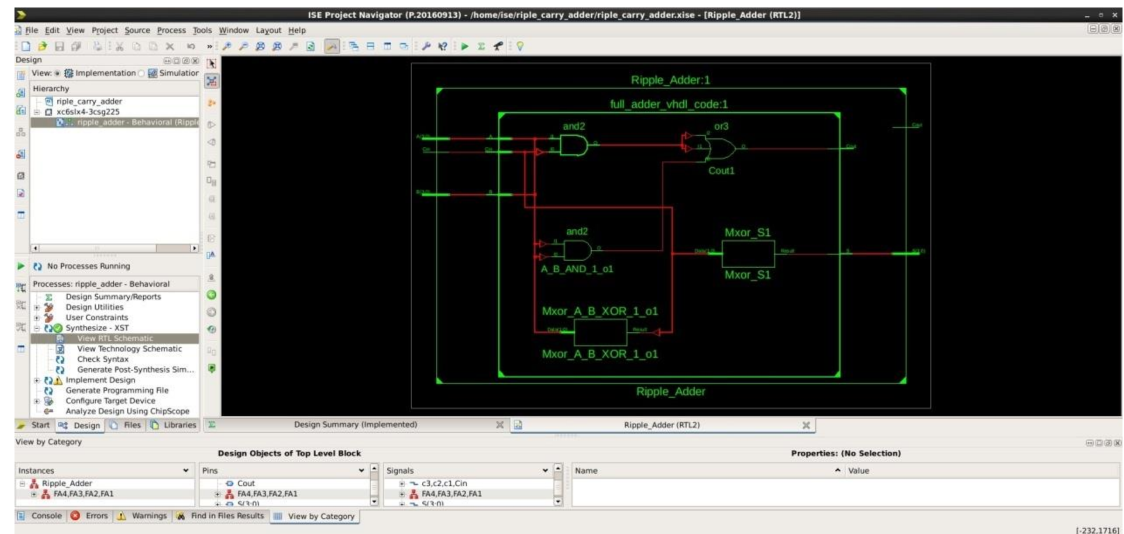


Εικόνα 7: TestBench Serial Adder

Αθροιστής μετάδοσης κρατουμένου (RCA – RippleCarryAdder)



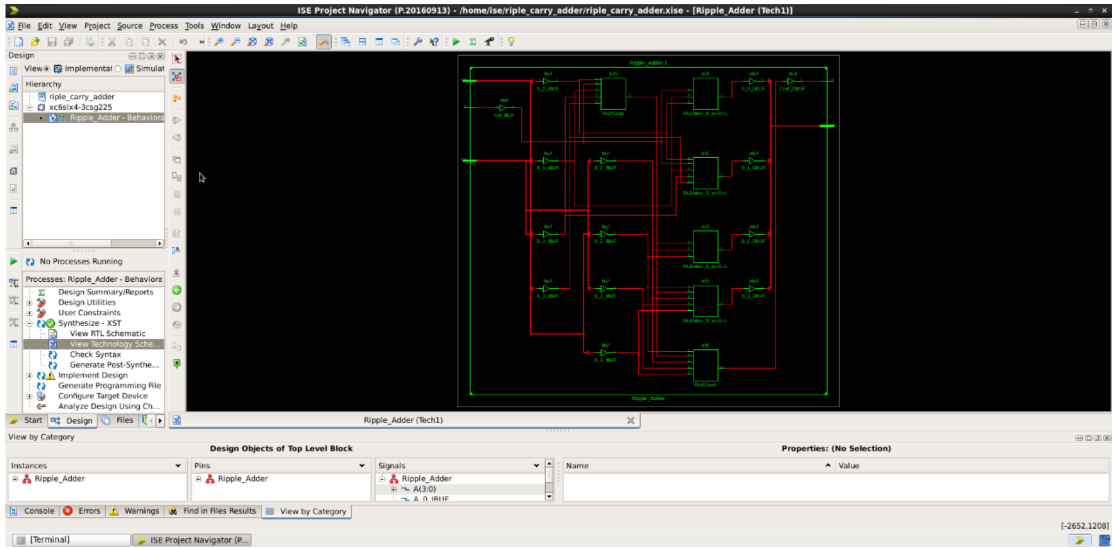
Εικόνα 8: RTL σχηματική (top-level block) - Ripple Carry Adder



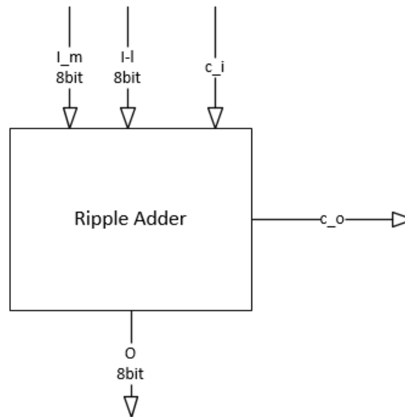
Εικόνα 9: RTL σχηματική - Ripple Carry Adder

Εικόνα 10: Περίληψη σχεδίασης - Ripple Carry Adder

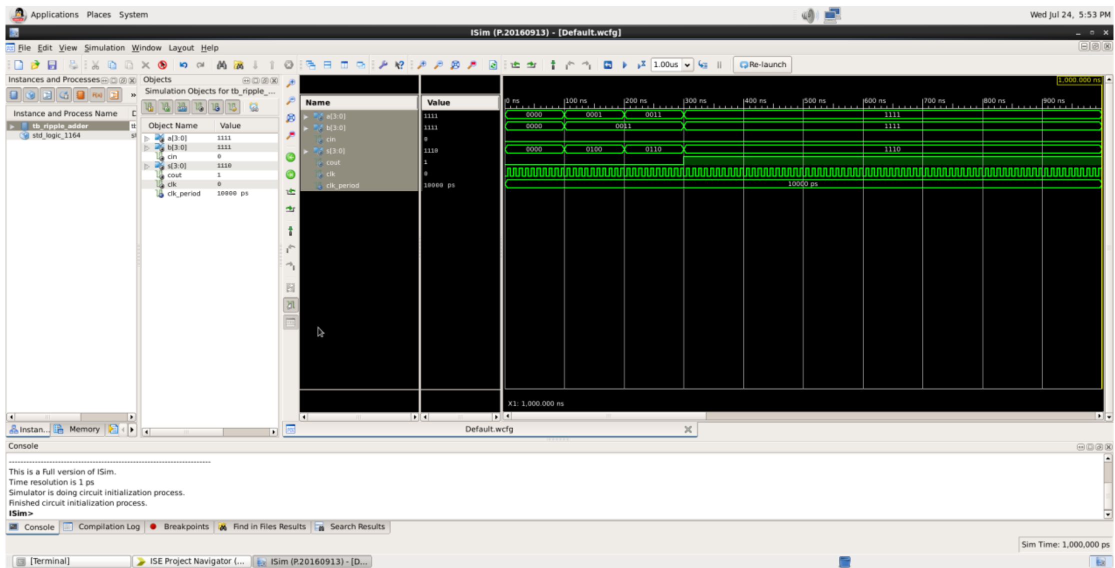
Εικόνα 11: Προβολή τεχνολογικών σχημάτων (top-level block) - Ripple Carry Adder



Εικόνα 12: Προβολή τεχνολογικών σχημάτων - Ripple Carry Adder

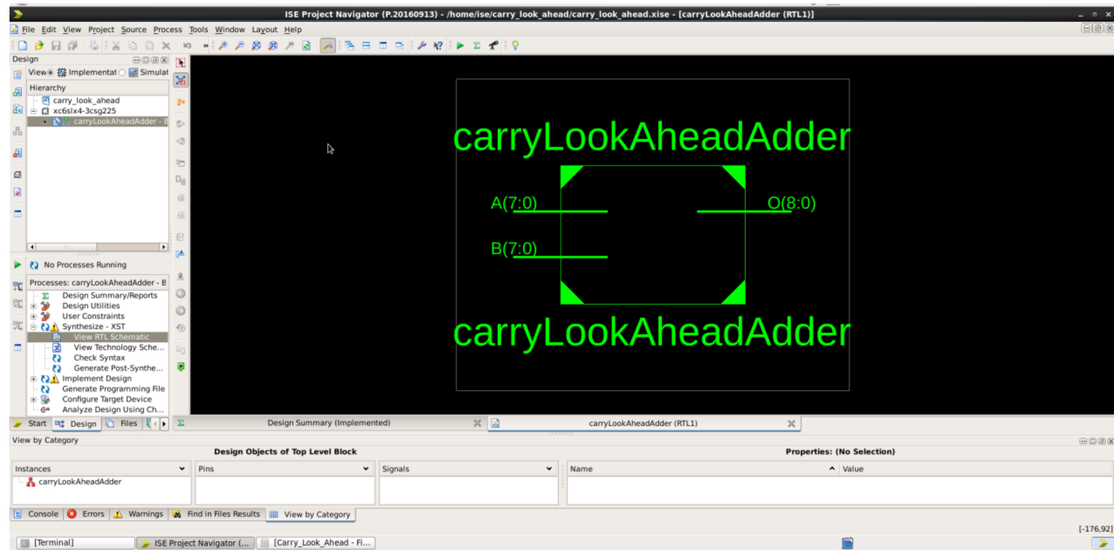


Εικόνα 13: Ripple Carry Adder

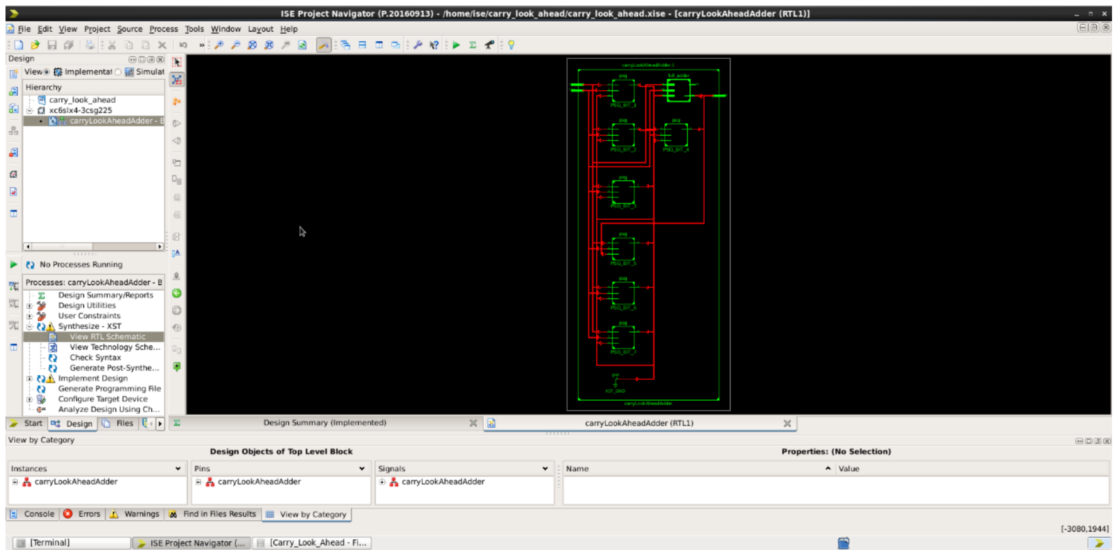


Εικόνα 14: TestBench Ripple Carry Adder

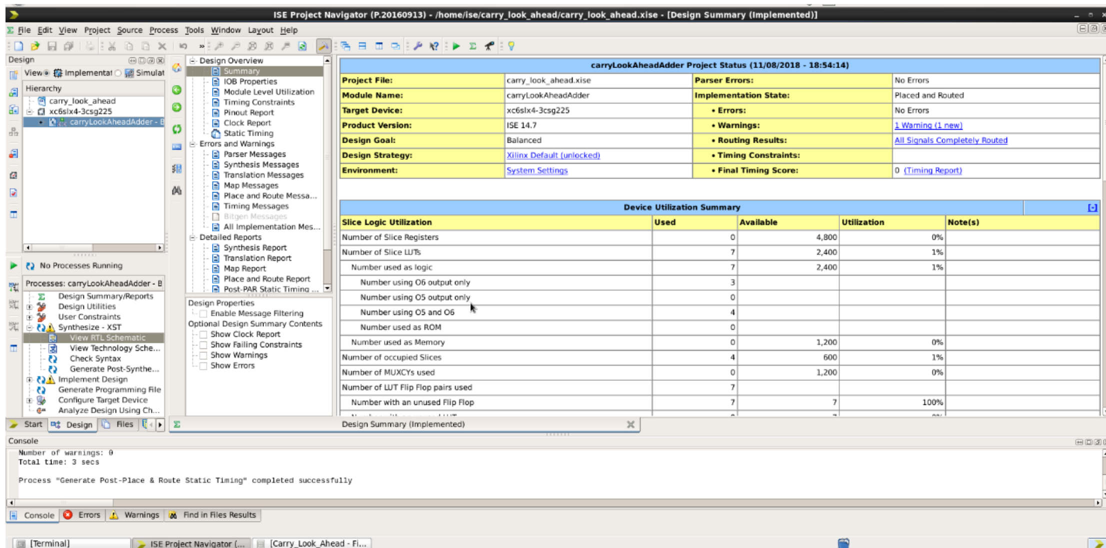
Αθροιστής πρόβλεψης κρατουμένου (CLA – CarryLookaheadAdder)



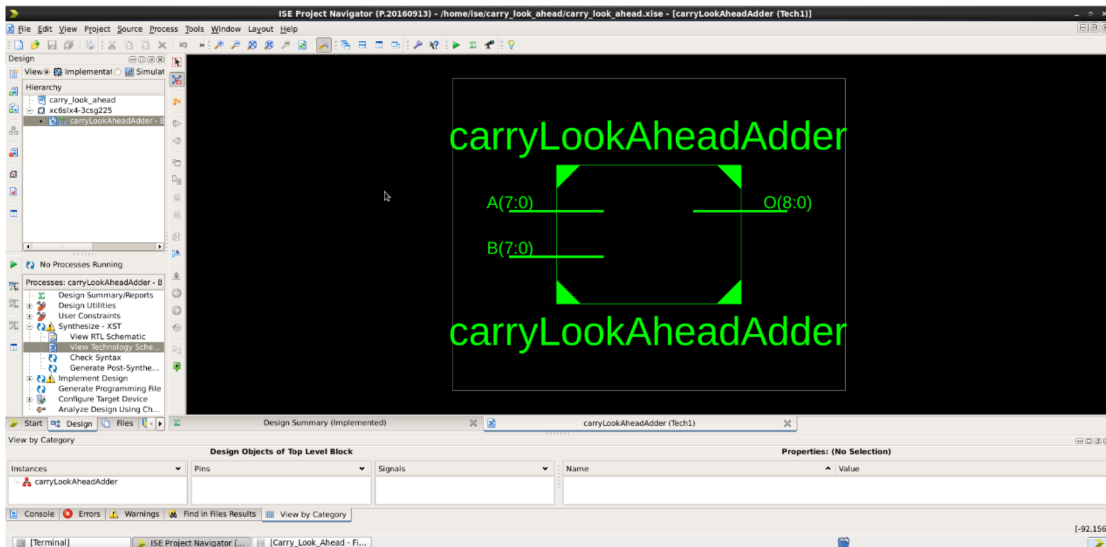
Εικόνα 15: RTL σχηματική (top-level block) - Carry Look Ahead Adder



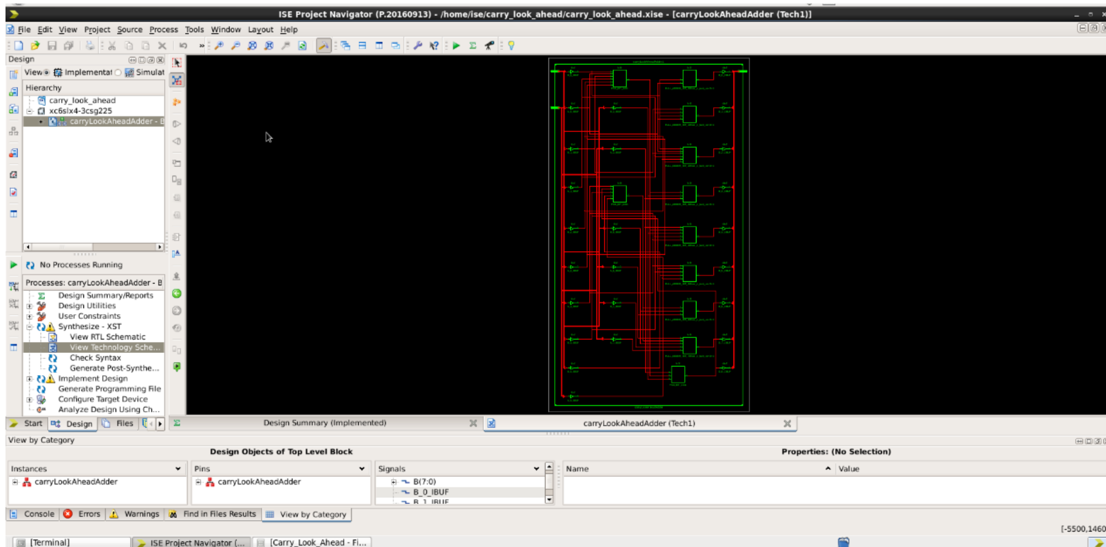
Εικόνα 16: RTL σχηματική - Carry Look Ahead Adder



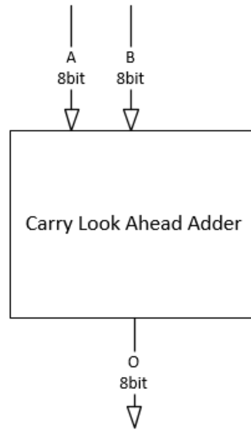
Εικόνα 17: Περίληψη σχεδίασης - Carry Look Ahead Adder



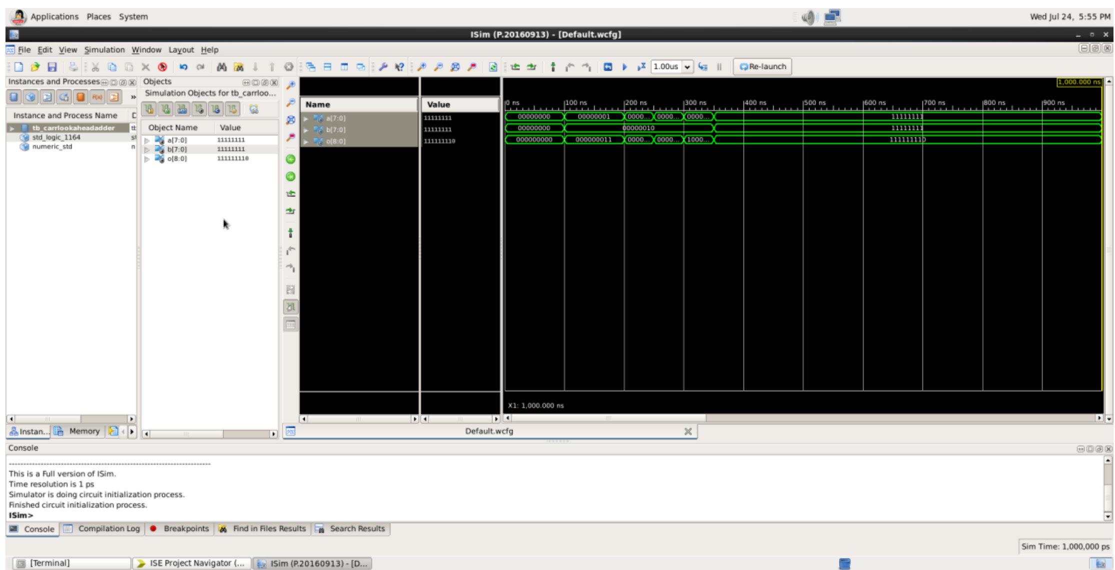
Εικόνα 18: Προβολή τεχνολογικών σχημάτων (top-level block) - Carry Look Ahead Adder



Εικόνα 19: Προβολή τεχνολογικών σχημάτων - Carry Look Ahead Adder

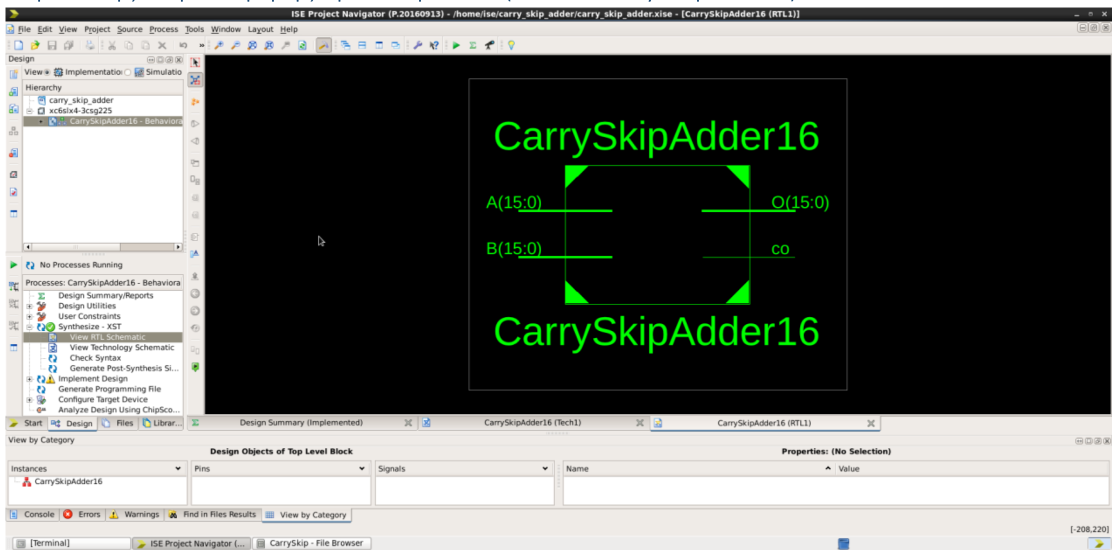


Εικόνα 20: Carry Look Ahead Adder

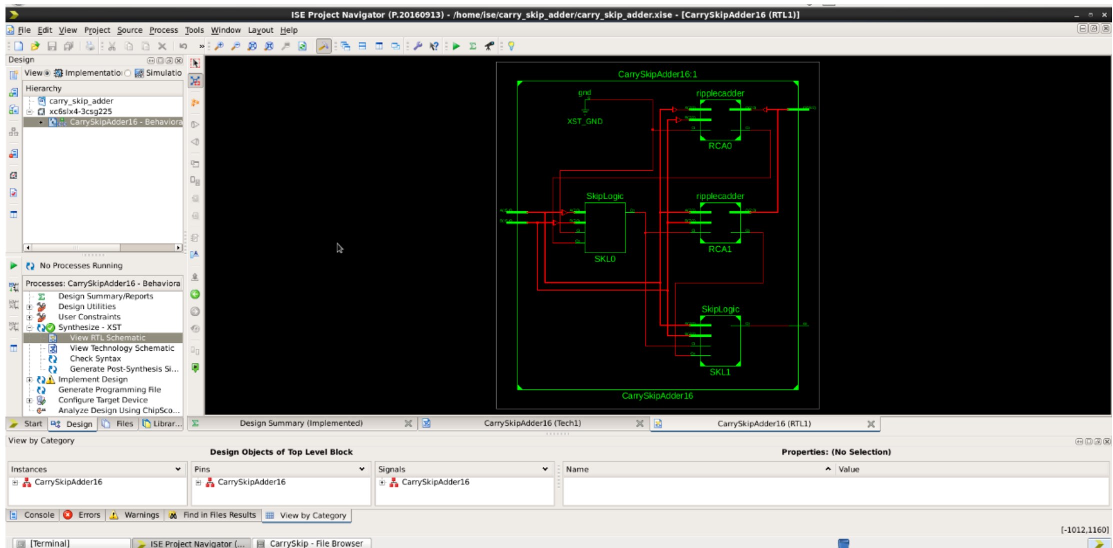


Εικόνα 21: TestBench Carry Look Ahead Adder

Αθροιστής παράκαμψης κρατουμένου (CSK – CarrySkipAdder)



Εικόνα 22: RTL σχηματική (top-level block) - Carry Skip Adder

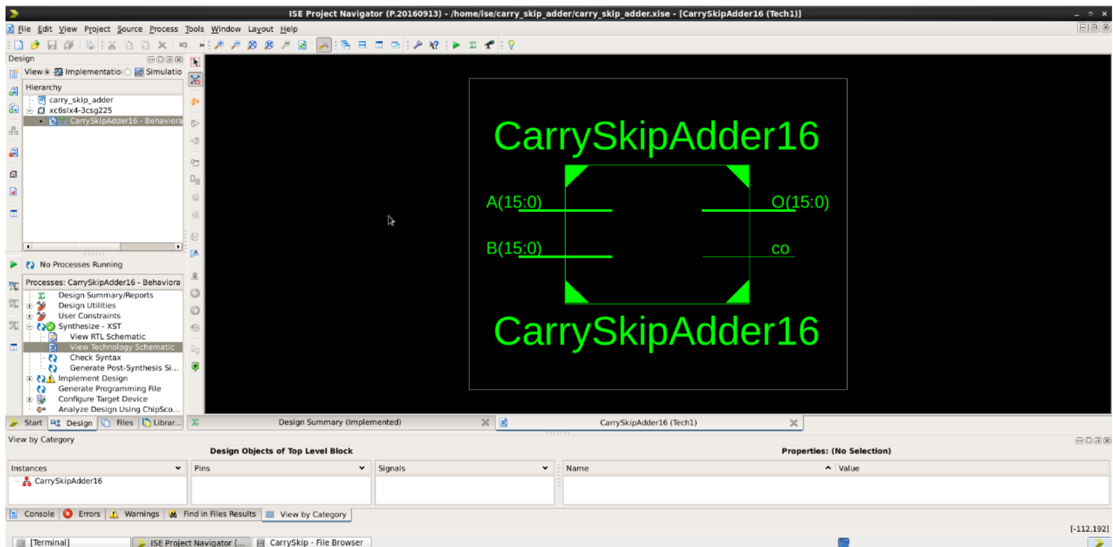


Εικόνα 23: RTL σχηματική - Carry Skip Adder

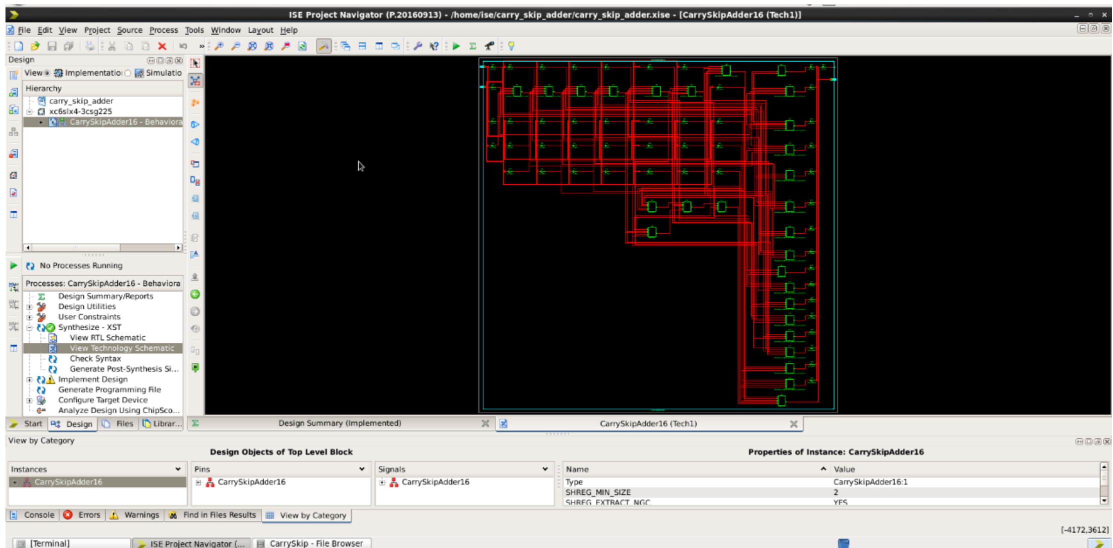
CarrySkipAdder16 Project Status (11/16/2018 - 20:19:38)			
Project File:	carry_skip_adder.xise	**Parser Errors:**	No Errors
Module Name:	CarrySkipAdder16	**Implementation State:**	Placed and Routed
Target Device:	xc6slx4-3csg225	**• Errors:**	No Errors
Product Version:	ISE 14.7	**• Warnings:**	No Warnings
Design Goal:	Balanced	**• Routing Results:**	All Signals Completely Routed
Design Strategy:	Xilinx Default (unlocked)	**• Timing Constraints:**	
Environment:	System Settings	**• Final Timing Score:**	0 (Timing Report)

Device Utilization Summary				
Slice Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Registers	0	4,800	0%	
Number of Slice LUTs	19	2,400	1%	
Number used as logic	19	2,400	1%	
Number using O6 output only	10			
Number using O5 output only	0			
Number using O5 and O6	9			
Number used as ROM	0			
Number used as Memory	0	1,200	0%	
Number of occupied Slices	8	600	1%	
Number of MUXCYs used	0	1,200	0%	
Number of LUT Flip Flop pairs used	19			
Number with an unused Flip Flop	19	19	100%	

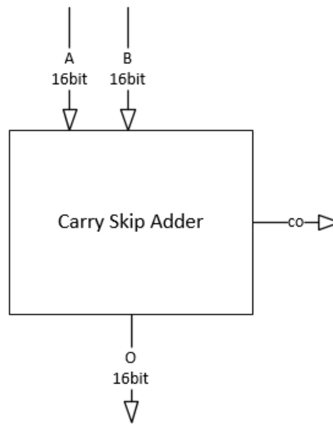
Εικόνα 24: Περίληψη σχεδίασης - Carry Skip Adder



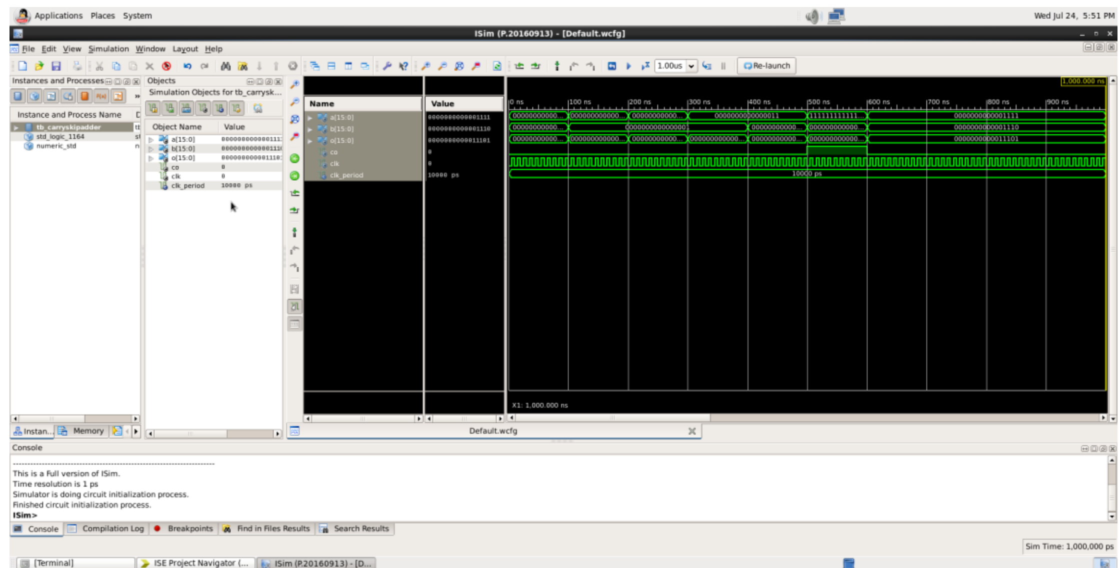
Εικόνα 25: Προβολή τεχνολογικών σχημάτων (top-level block) - Carry Skip Adder



Εικόνα 26: Προβολή τεχνολογικών σχημάτων - Carry Skip Adder

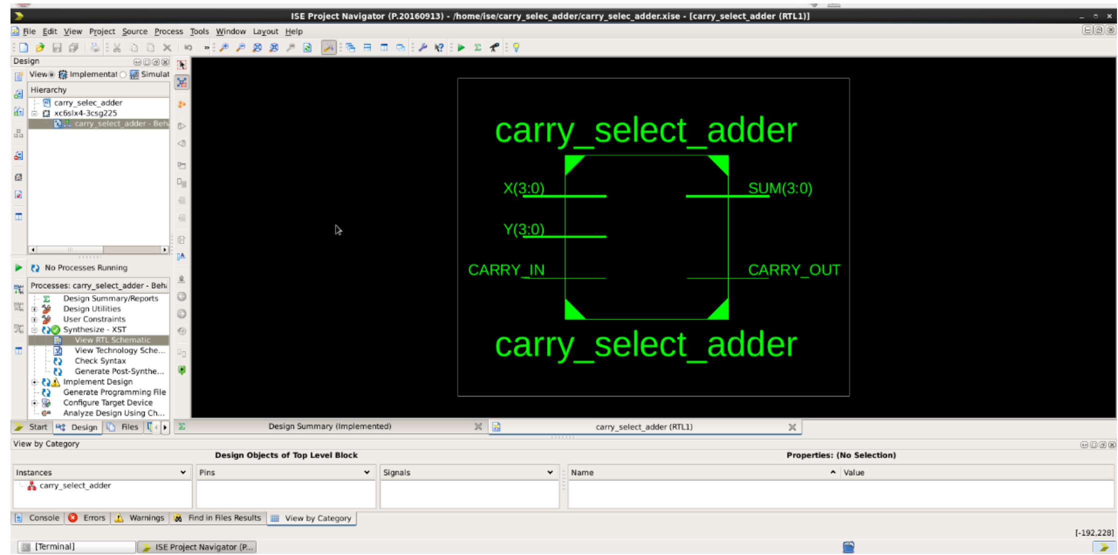


Εικόνα 27: Carry Skip Adder

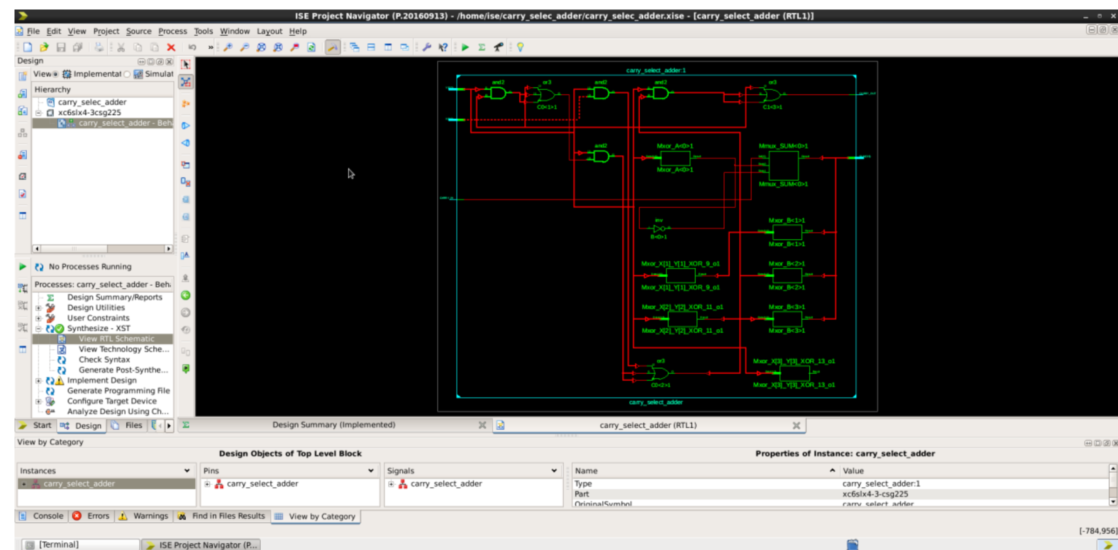


Εικόνα 28: TestBench Carry Skip Adder

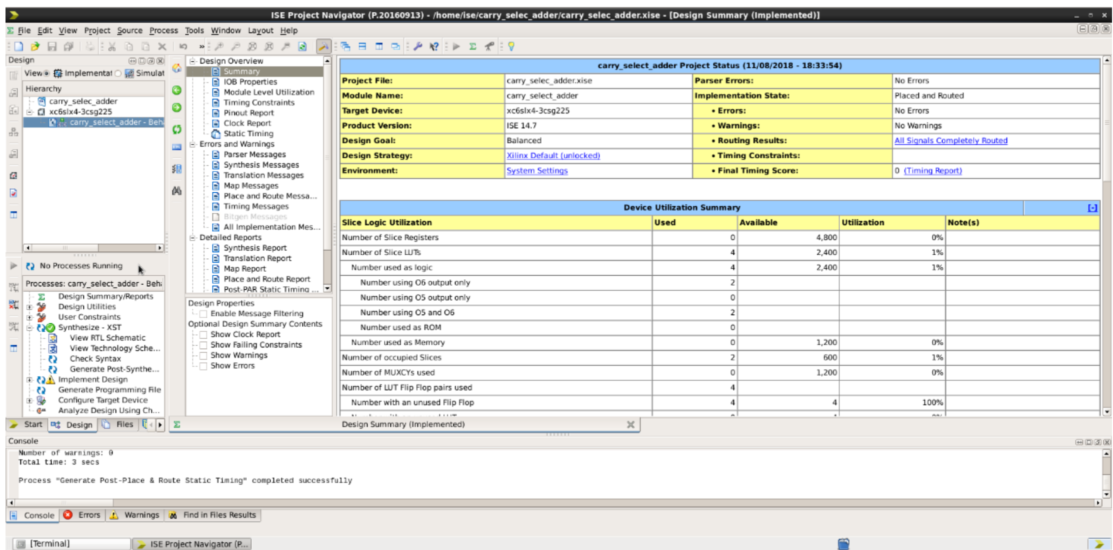
Αθροιστής επιλογής κρατουμένου (CSL – CarrySelectAdder)



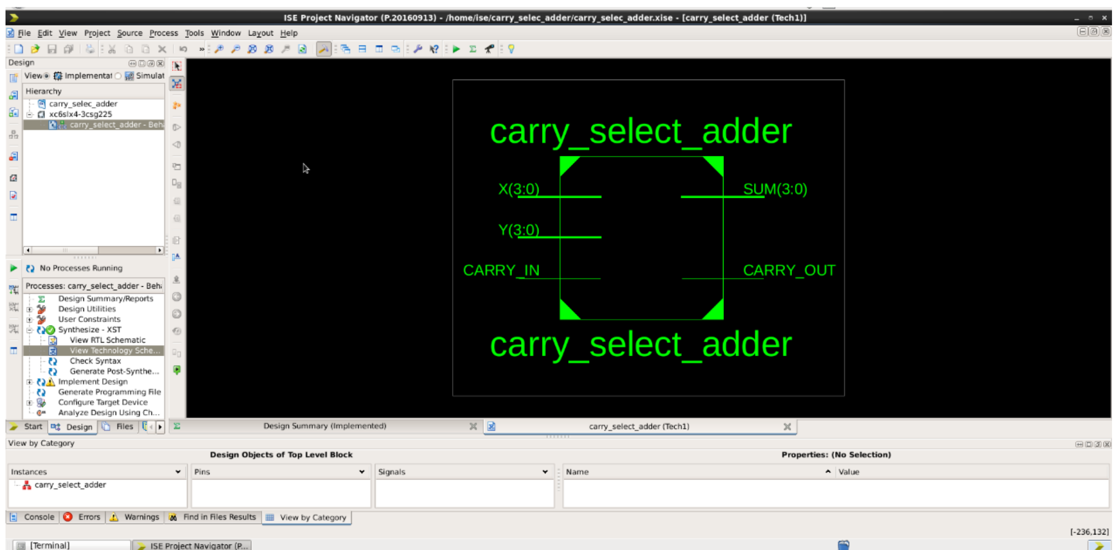
Εικόνα 29: RTL σχηματική (top-level block) - Carry Select Adder



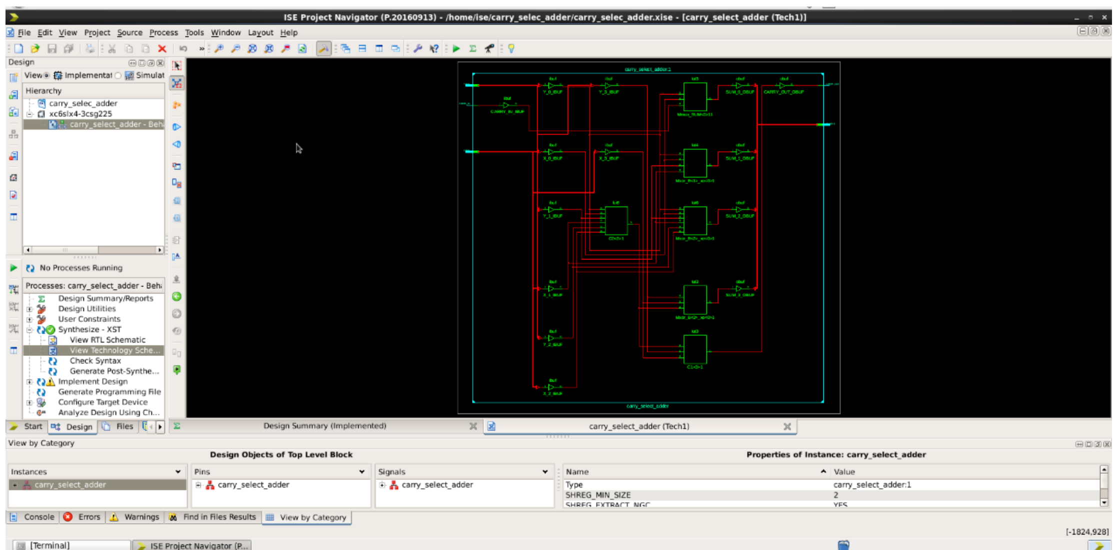
Εικόνα 30: RTL σχηματική - Carry Select Adder



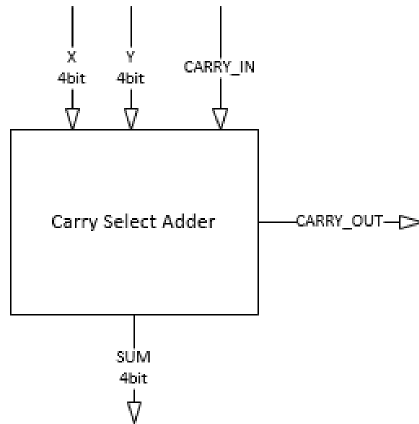
Εικόνα 31: Περίληψη σχεδίασης - Carry Select Adder



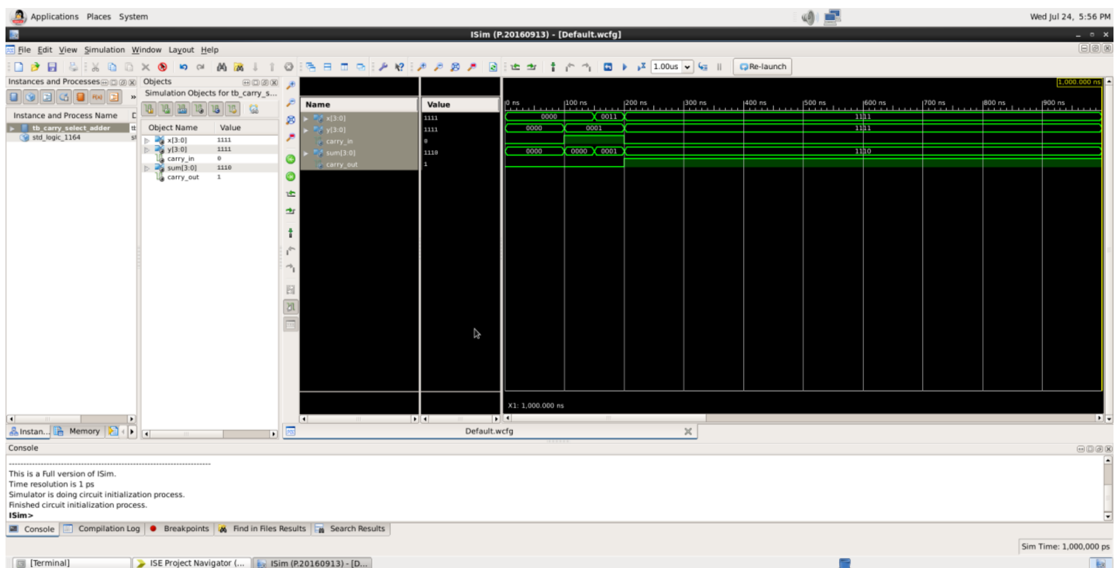
Εικόνα 32: Προβολή τεχνολογικών σχημάτων (top-level block) - Carry Select Adder



Εικόνα 33: Προβολή τεχνολογικών σχημάτων - Carry Select Adder

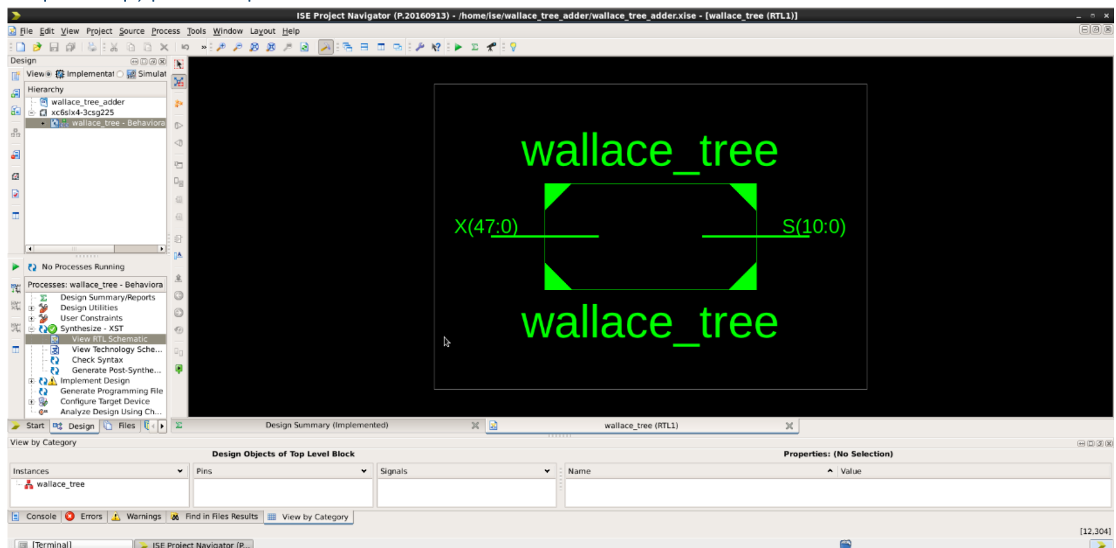


Εικόνα 34: Carry Select Adder

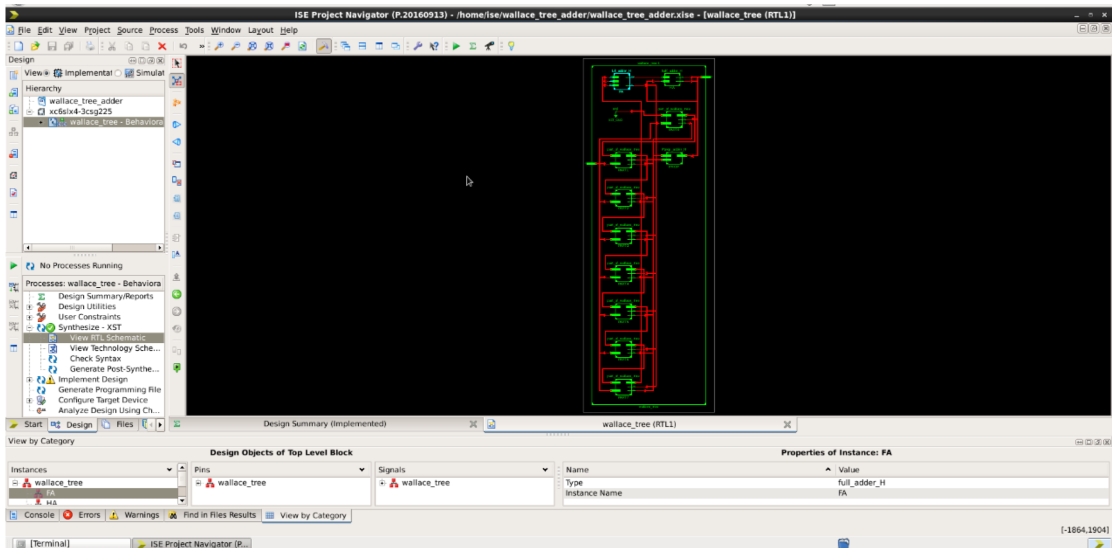


Εικόνα 35: TestBench Carry Select Adder

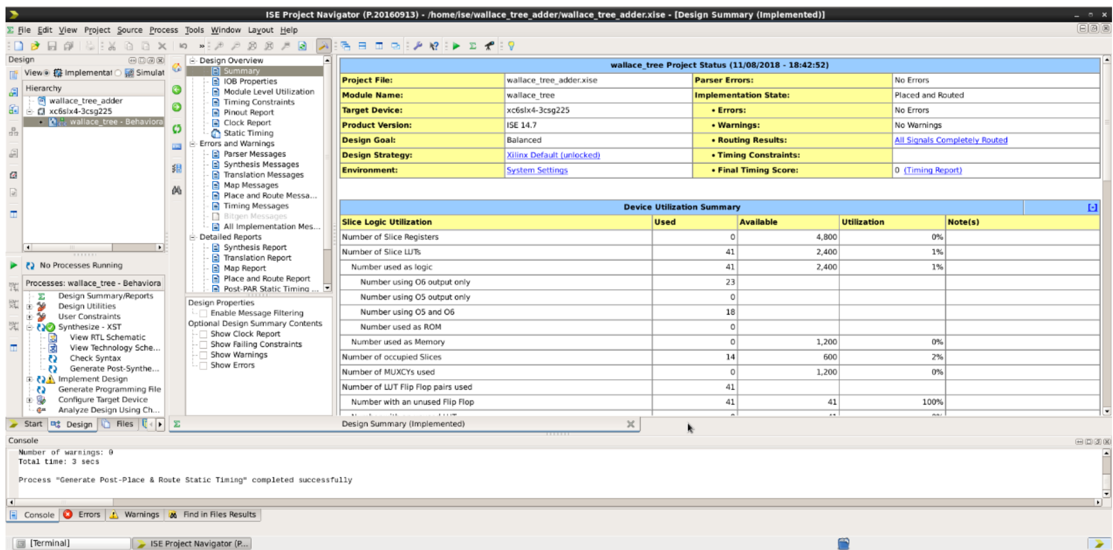
Αθροιστής με δέντρο Wallace



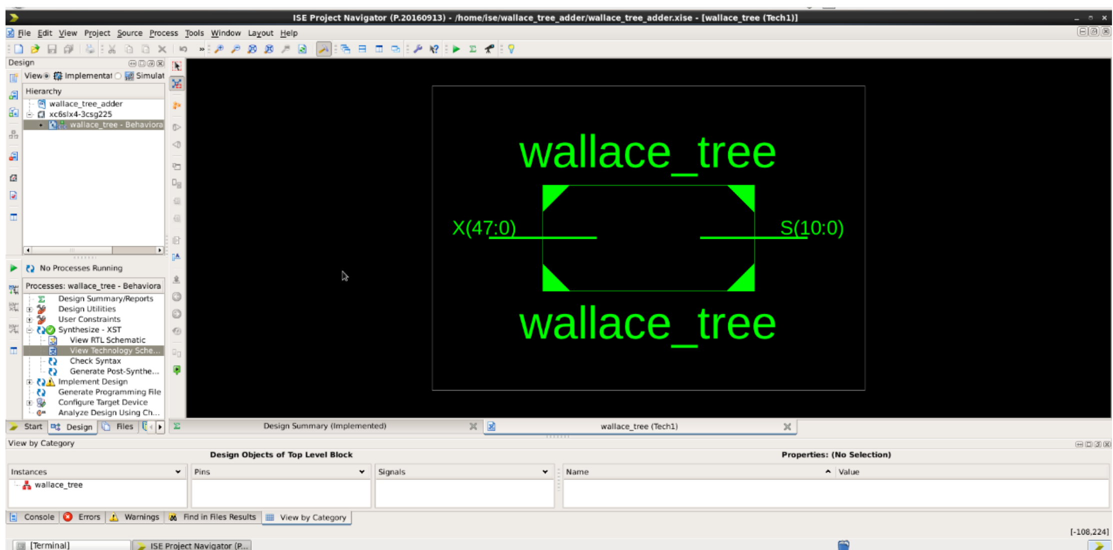
Εικόνα 36: RTL σχηματική (top-level block) - Wallace Tree Adder



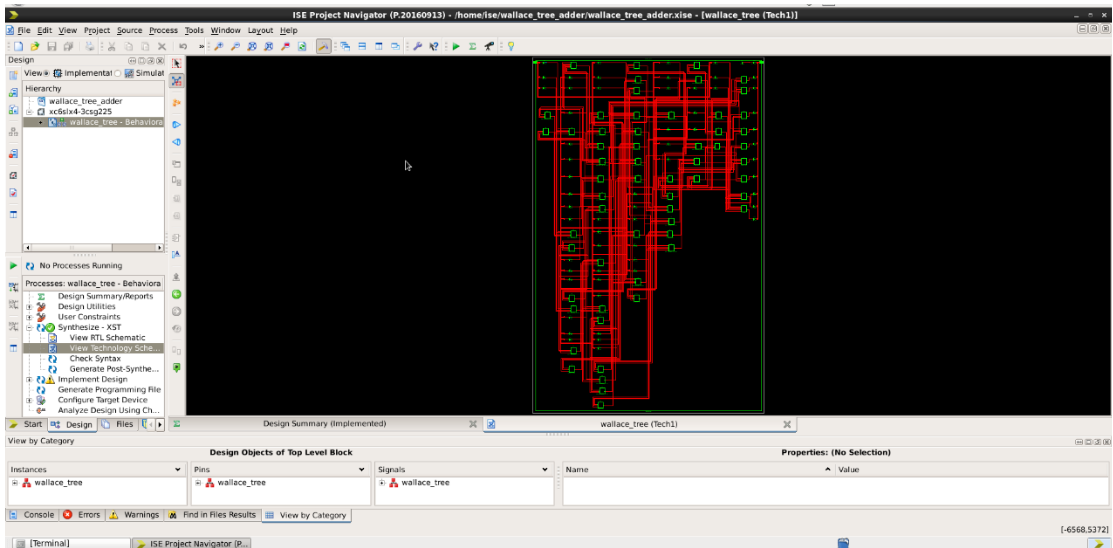
Εικόνα 37: RTL σχηματική - Wallace Tree Adder



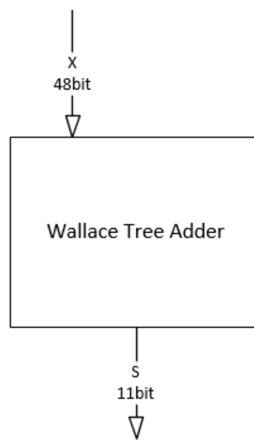
Εικόνα 38: Περίληψη σχεδίασης - Wallace Tree Adder



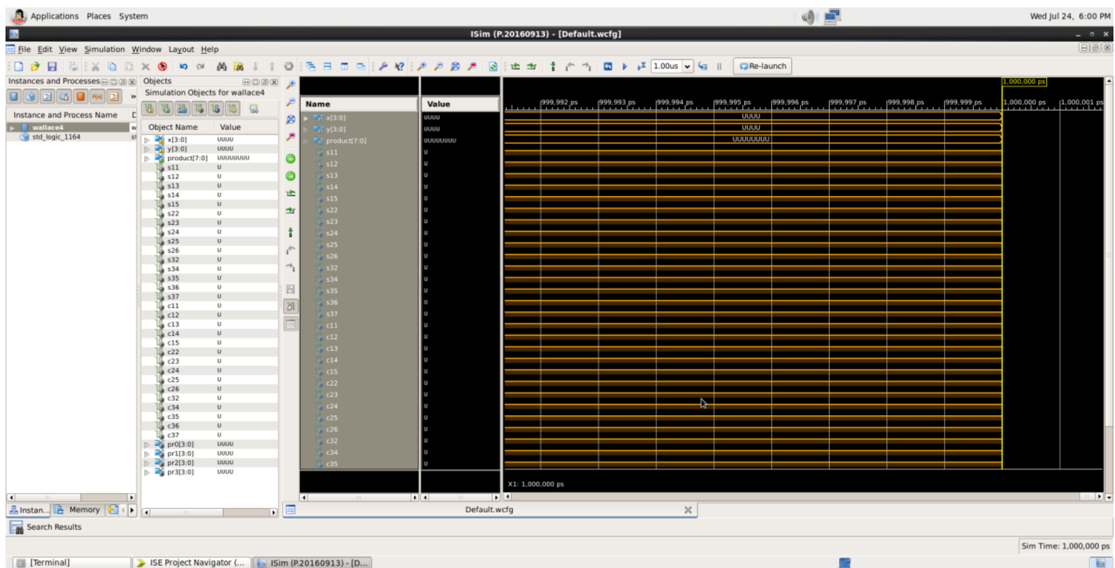
Εικόνα 39: Προβολή τεχνολογικών σχημάτων (top-level block) - Wallace Tree Adder



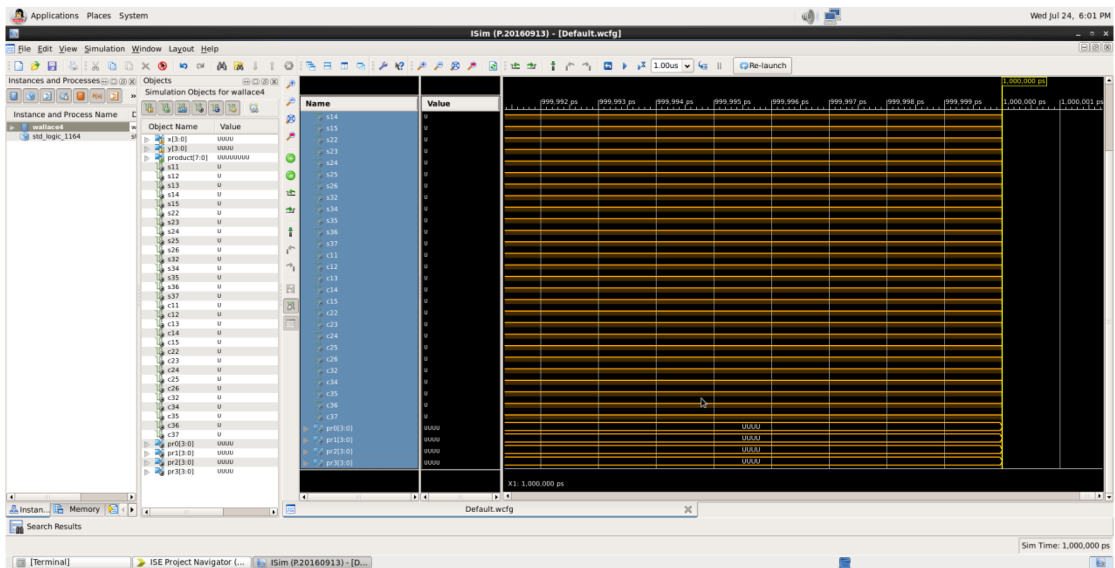
Εικόνα 40: Προβολή τεχνολογικών σχημάτων - Wallace Tree Adder



Εικόνα 41: Wallace Tree Adder



Εικόνα 42: TestBench Wallace Tree Adder (1)



Εικόνα 43: TestBench Wallace Tree Adder (2)

Πολλαπλασιαστές

Οι πολλαπλασιαστές έχουν ένα σημαντικό ρόλο σήμερα στην επεξεργασία ψηφιακού σήματος και σε διάφορες άλλες εφαρμογές. Με πλεονέκτημα τη τεχνολογία πολλοί ερευνητές έχουν προσπαθήσει και προσπαθούν να σχεδιάσουν πολλαπλασιαστές οι οποίοι ακολουθούν κάποιους από τους ακόλουθους σχεδιαστικούς στόχους – υψηλή ταχύτητα, χαμηλή κατανάλωση ενέργειας, λιγότερο χώρο στο κύκλωμα ή ακόμα και έναν συνδυασμό από τα παραπάνω σε έναν πολλαπλασιαστή προκειμένου να ταιριάζει καλύτερα στην εφαρμογή του στο VLSI.

Ο αλγόριθμος «πρόσθεση και μετατόπιση» (“add and shift”)είναι μία κοινή μέθοδο πολλαπλασιασμού. Στους παράλληλους πολλαπλασιαστές ο αριθμός των μερικών γινομένων που πρέπει να προστεθούν είναι από τις κύριες παραμέτρους που καθορίζουν την απόδοση του πολλαπλασιαστή. Για να μειωθεί ο αριθμός των μερικών γινομένων που πρέπει να προστεθούν έχει προκύψει ο τροποποιημένος αλγόριθμος Booth (Modified Booth Algorithm)που είναι αρκετά δημοφιλής. Όσο αναφορά την επίτευξη της ταχύτητας ο αλγόριθμος Wallace Treeμπορεί να χρησιμοποιηθεί για να μειώσει τον αριθμό διαδοχικών προσθέσεων. Ως πλεονέκτημα ενός πολλαπλασιαστή αποτελεί η δημιουργία του από τον συνδυασμό των αλγορίθμων Modified Booth και Wallace Tree.

Παρόλα αυτά καθώς αυξάνουμε τον παραλληλισμό, το πλήθος των μετατοπίσεων μεταξύ των μερικών γινομένων και ενδιάμεσων αθροίσεων αυξάνεται και έχει σαν αποτέλεσμα την μείωση της ταχύτητας, αύξηση στη χρήση περιοχής από το κύκλωμα και αύξηση της κατανάλωσης ενέργειας. Από την άλλη πλευρά έχουμε τους «σειριακούς-παράλληλους» πολλαπλασιαστές που πετυχαίνουν καλύτερη απόδοση όσο αναφορά την κατανάλωση της περιοχής και της ενέργειας. Η επιλογή του παράλληλου ή σειριακού πολλαπλασιαστή εξαρτάται από τη φύση της εφαρμογής.

Οι κανόνες για τον δυαδικό πολλαπλασιασμό μπορούν να δηλωθούν ως εξής:

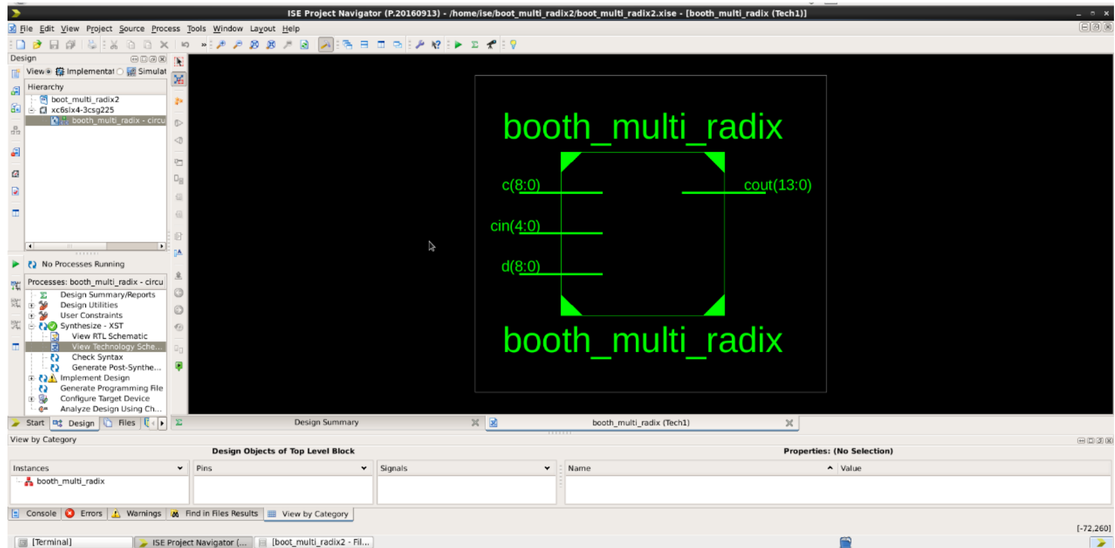
1. Εάν το ψηφίο που πολλαπλασιάζεται είναι το 1, ο πολλαπλασιαστής απλά το αντιγράφει προς τα κάτω την είσοδο και αναπαριστά το γινόμενο.
2. Εάν το ψηφίο που πολλαπλασιάζεται είναι το 0, το γινόμενο είναι επίσης 0.

Για το σχεδιασμό ενός κυκλώματος πολλαπλασιαστή θα πρέπει να έχουμε κυκλώματα για να κάνουμε τα εξής τρία πράγματα:

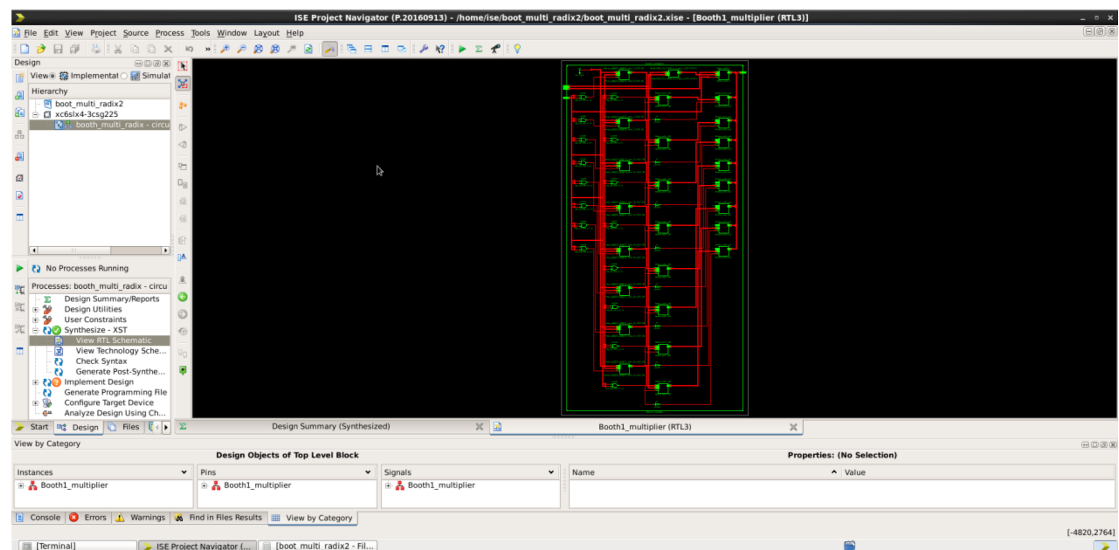
1. Θα πρέπει να είναι σε θέση να προσδιορίσει αν το ένα bit είναι 0 ή 1.
2. Θα πρέπει να μπορεί να μετατοπίζει προς τα αριστερά τα μερικά γινόμενα.
3. Θα πρέπει να είναι σε θέση να προσθέσει όλα τα μερικά γινόμενα για να δώσει το τελικό γινόμενο ως άθροισμα των μερικών γινομένων.

Εφαρμογή και Αποτελέσματα

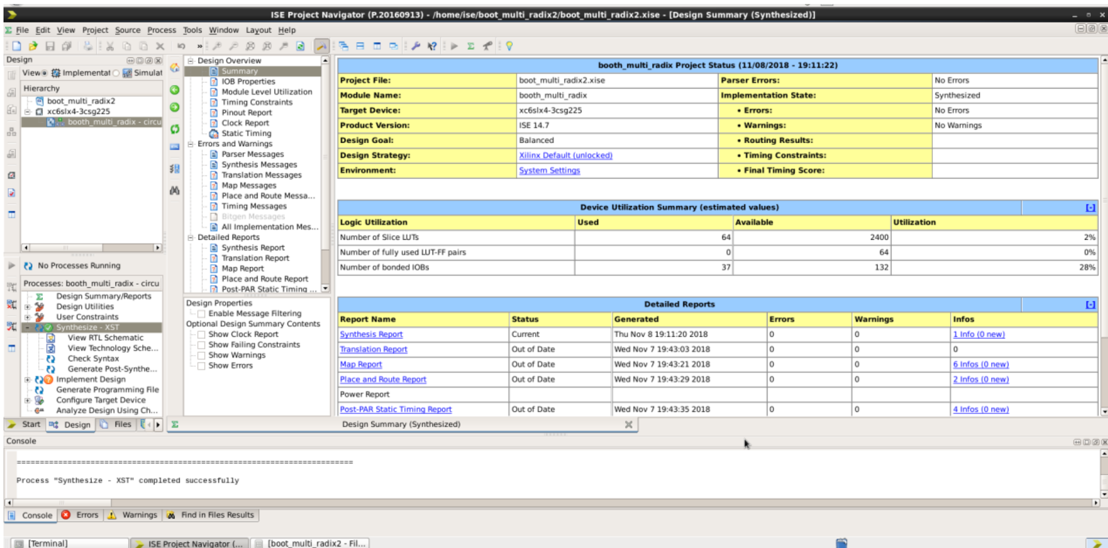
Πολλαπλασιαστής Booth των n·mbits βάσης-2



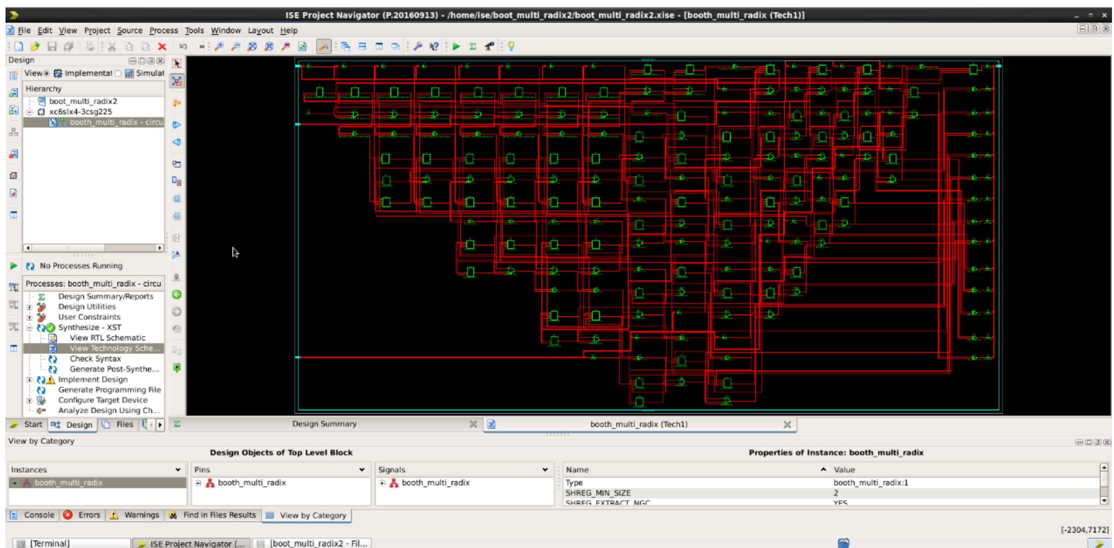
Εικόνα 44: Προβολή τεχνολογικών σχημάτων / RTL σχηματική (top-level block) - Booth βάσης-2 Multiplier



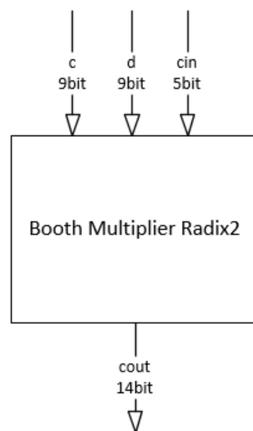
Εικόνα 45: RTL σχηματική - Booth βάσης-2 Multiplier



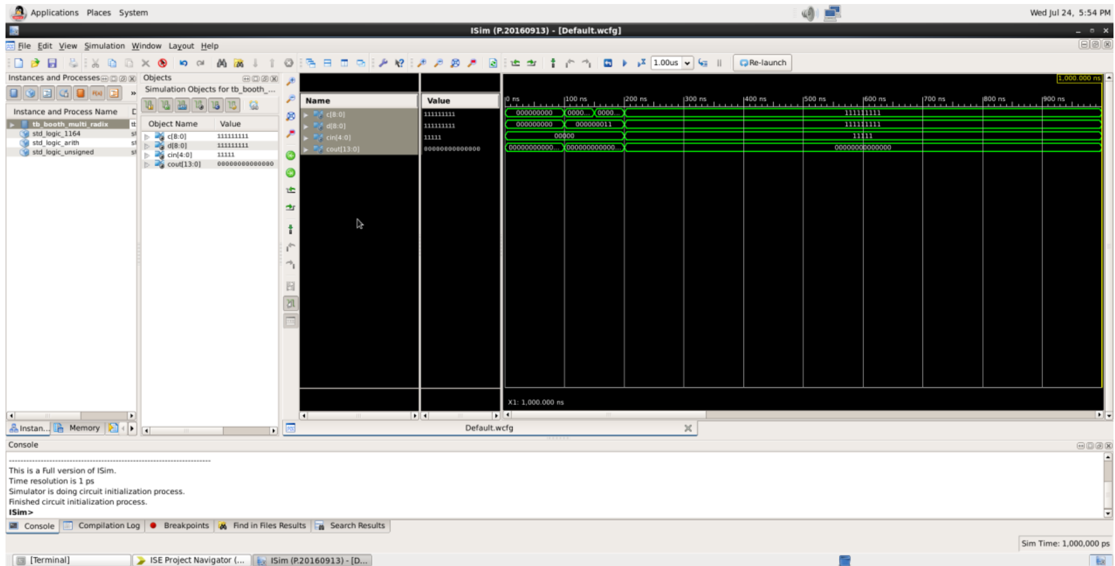
Εικόνα 46: Περίληψη σχεδίασης - Booth βάσης-2 Multiplier



Εικόνα 47: Προβολή τεχνολογικών σχημάτων - Booth βάσης-2 Multiplier

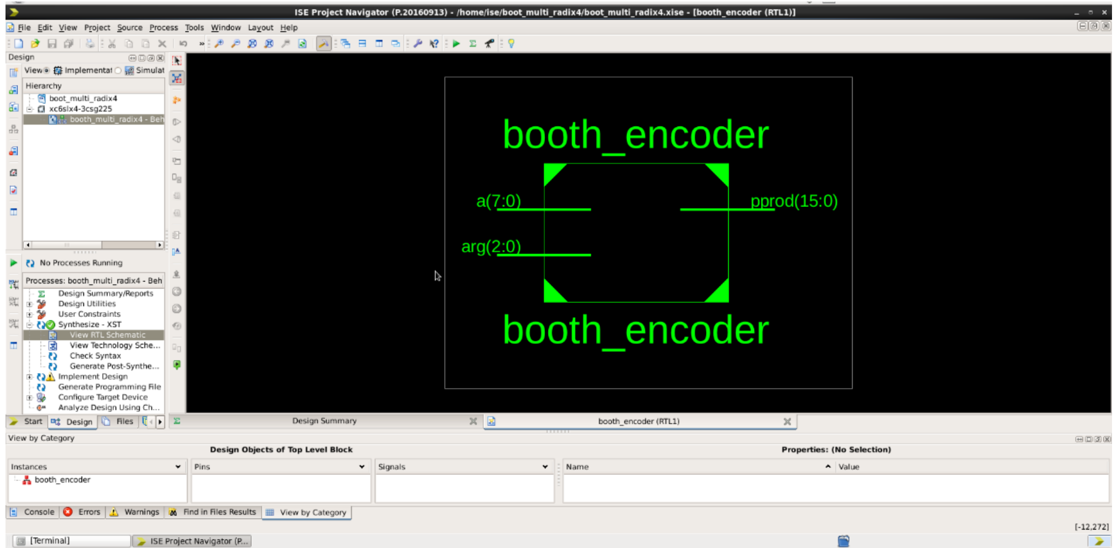


Εικόνα 48: Booth βάσης-2 Multiplier

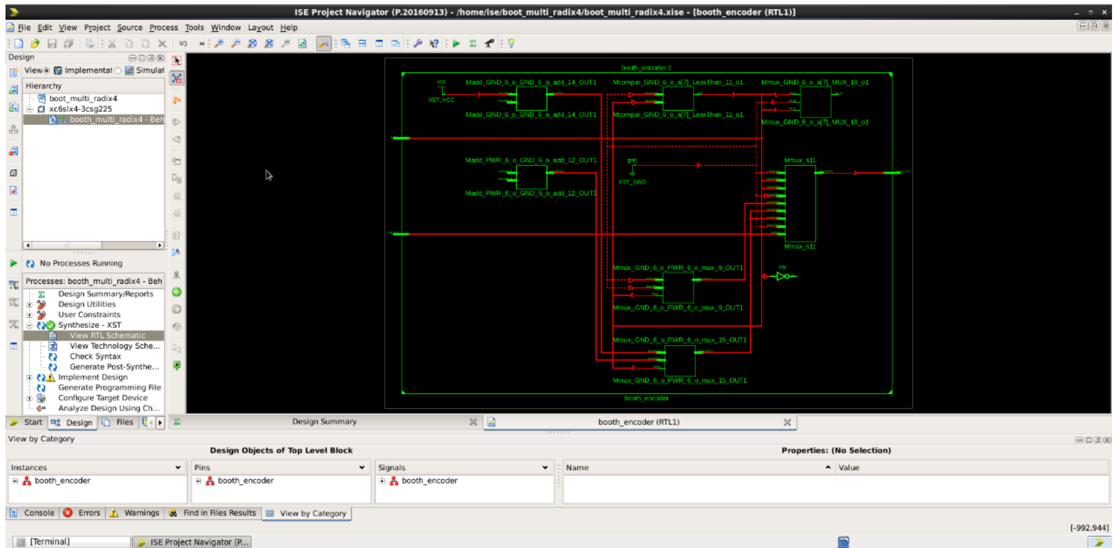


Εικόνα 49: TestBench Booth βάσης-2 Multiplier

Πολλαπλασιαστής Booth των n·mbits βάσης-4



Εικόνα 50: RTL σχηματική (top-level block) - Booth βάσης-4 Multiplier



Εικόνα 51: RTL σχηματική - Booth βάσης-4 Multiplier

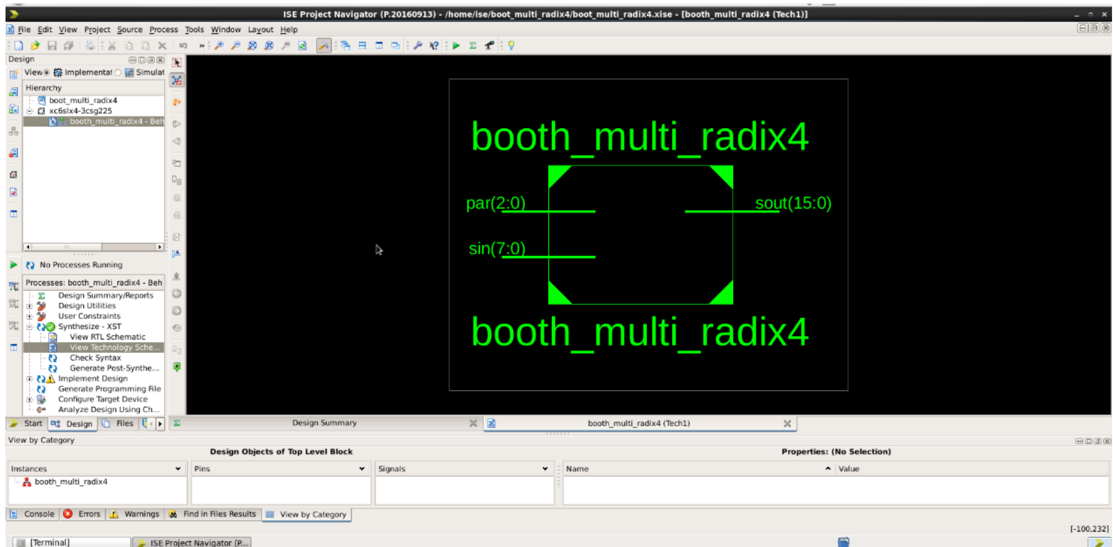
ISE Project Navigator (P.20160913) - /home/ise/boot_multi_radix4/boot_multi_radix4.xise - [Design Summary]

booth_multi_radix4 Project Status			
Project File:	boot_multi_radix4.xise	**Parser Errors:**	No Errors
Module Name:	booth_multi_radix4	**Implementation State:**	Placed and Routed
Target Device:	xc6slx4-3csg225	**• Errors:**	No Errors
Product Version:	ISE 14.7	**• Warnings:**	No Warnings
Design Goal:	Balanced	**• Routing Results:**	All Signals Completely Routed
Design Strategy:	Xilinx Default (unlocked)	**• Timing Constraints:**	
Environment:	System Settings	**• Final Timing Score:**	0 (Timing Report)

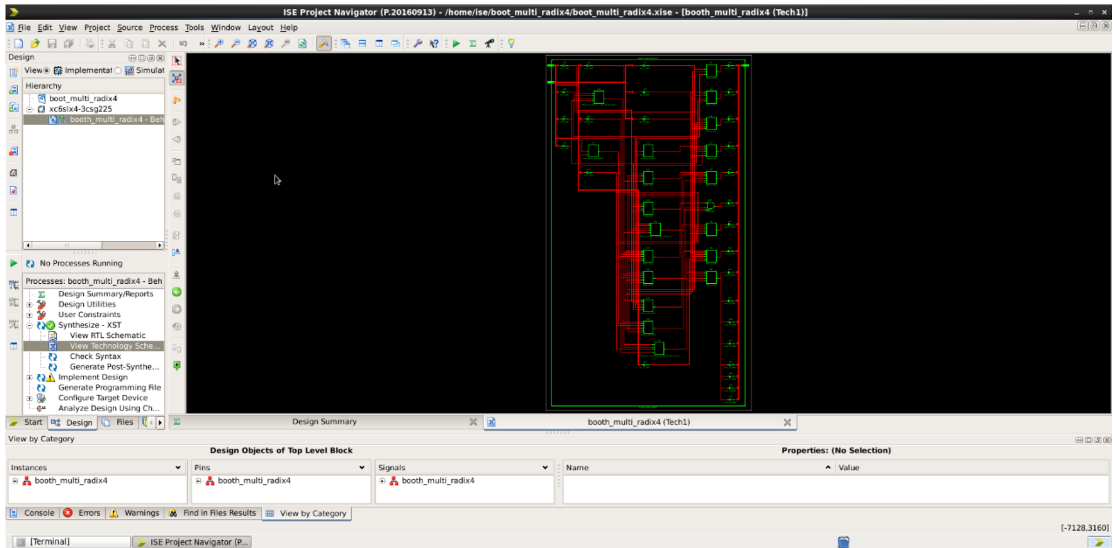
Device Utilization Summary

Slice Logic Utilization	Used	Available	Utilization	Note(s)
Number of Slice Registers	0	4,800	0%	
Number of Slice LUTs	15	2,400	1%	
Number used as logic	15	2,400	1%	
Number using O6 output only	11			
Number using O5 output only	0			
Number using O5 and O6	4			
Number used as ROM	0			
Number used as Memory	0	1,200	0%	
Number of occupied Slices	7	600	1%	
Number of MUXCYs used	0	1,200	0%	
Number of LUT Flip Flop pairs used	15			
Number with an unused Flip Flop	15	15	100%	

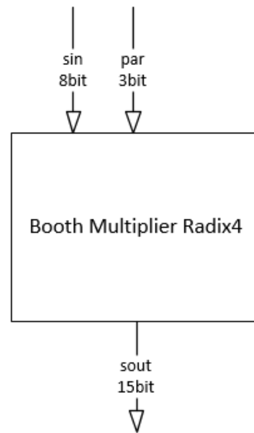
Εικόνα 52: Περίληψη σχεδίασης - Booth βάσης-4 Multiplier



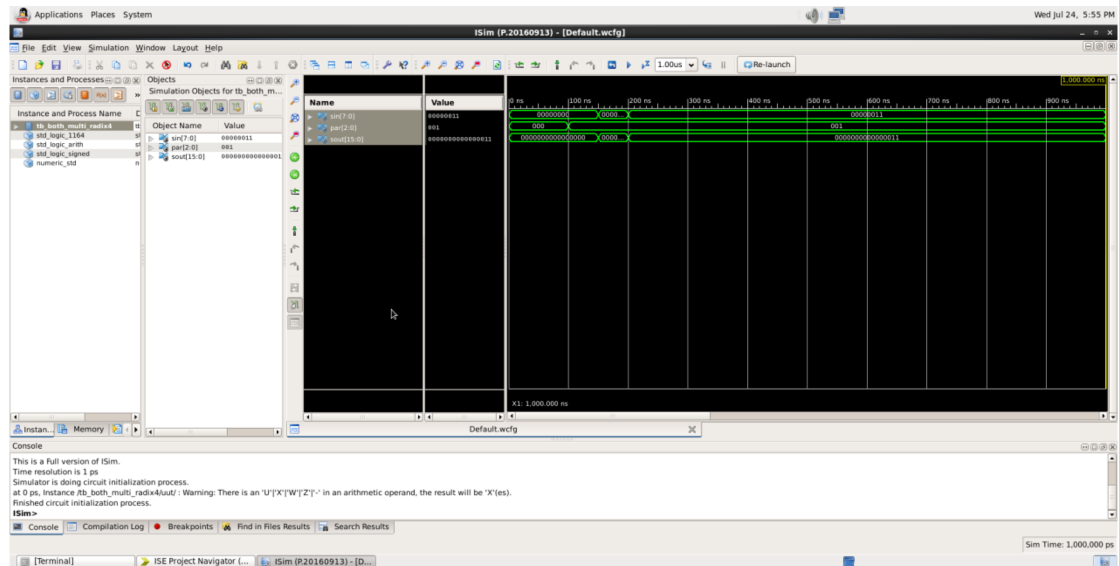
Εικόνα 53: Προβολή τεχνολογικών σχημάτων (top-level block) - Booth βάσης-4 Multiplier



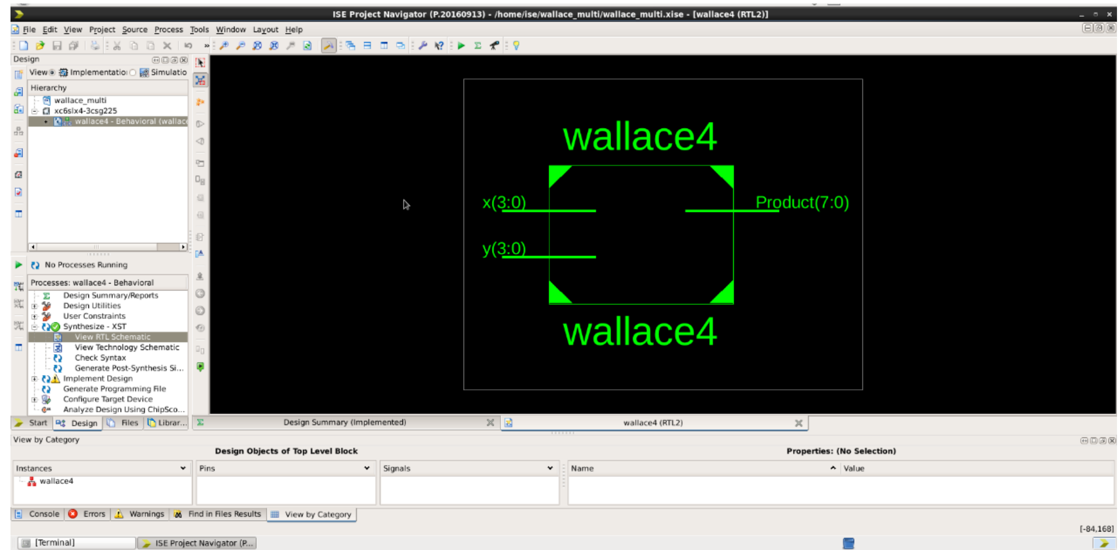
Εικόνα 54: Προβολή τεχνολογικών σχημάτων - Booth βάσης-4 Multiplier



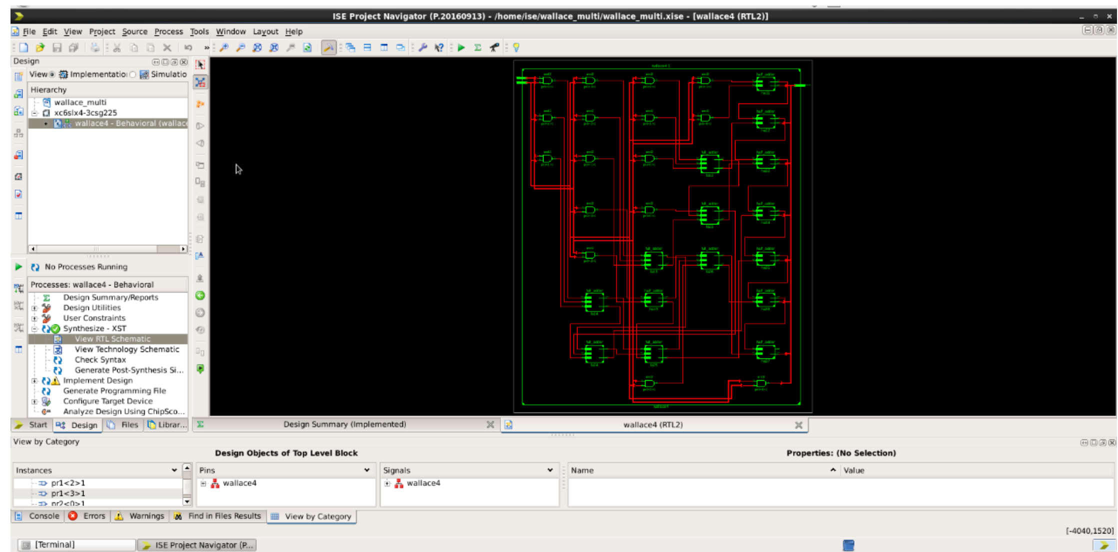
Εικόνα 55: Booth βάσης-4 Multiplier



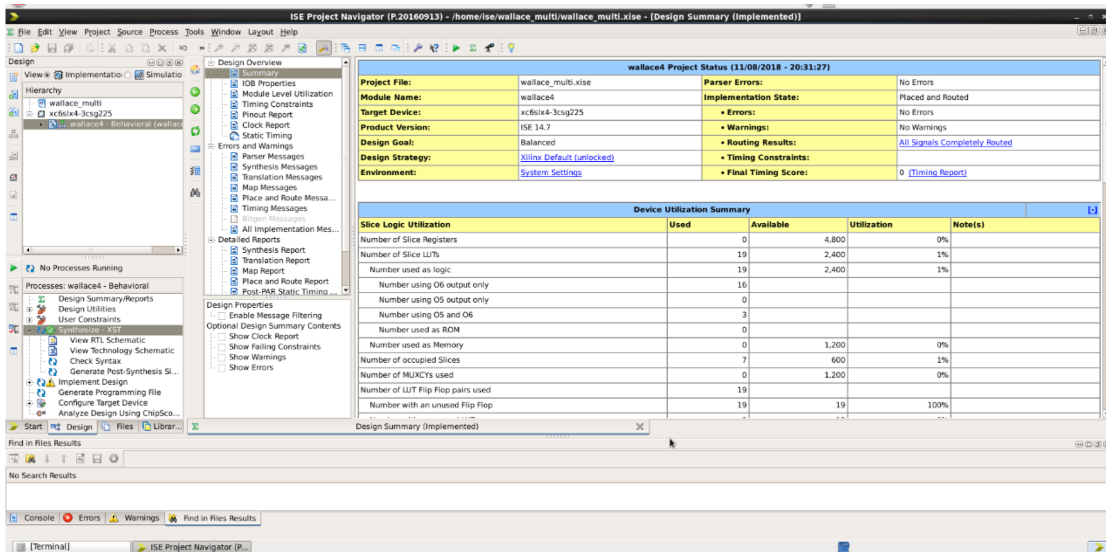
Εικόνα 56: TestBench Booth βάσης-4 Multiplier



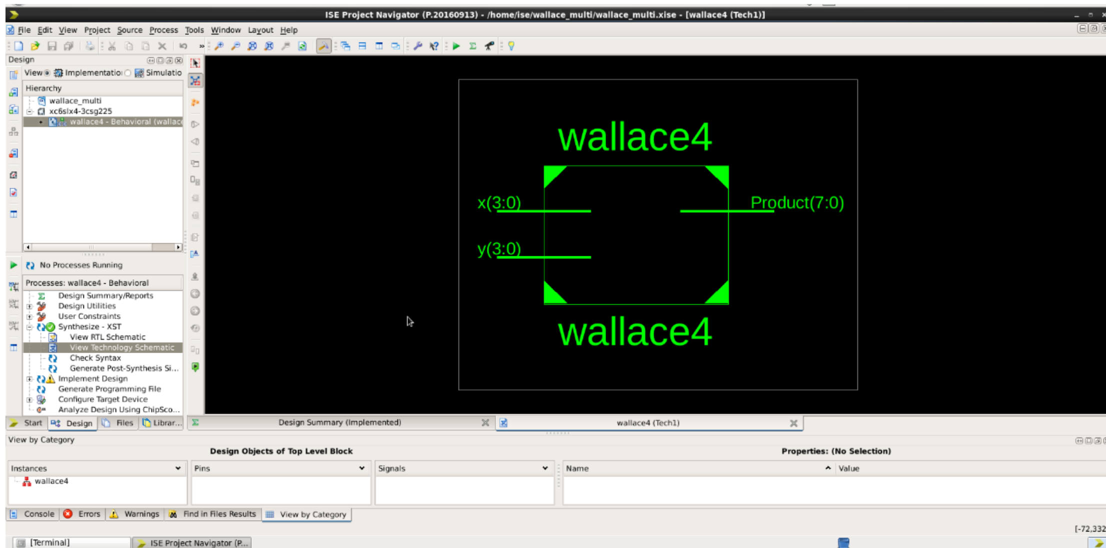
Εικόνα 57: RTL σχηματική (top-level block) - Wallace Multiplier



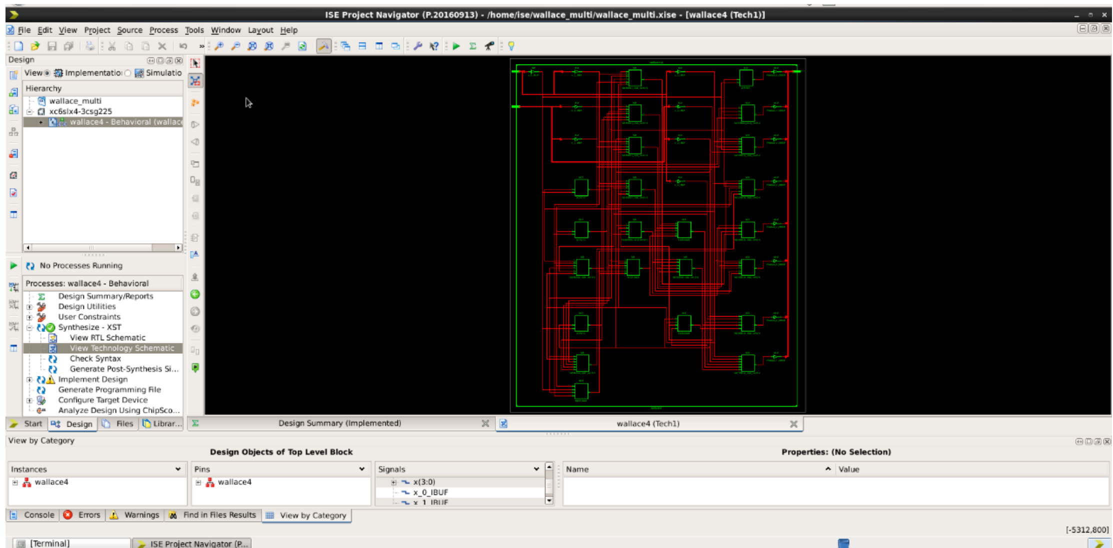
Εικόνα 58: RTL σχηματική - Wallace Multiplier



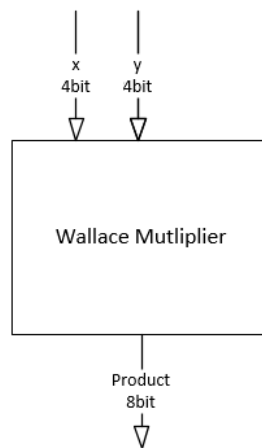
Εικόνα 59: Περίληψη Σχεδίασης - Wallace Multiplier



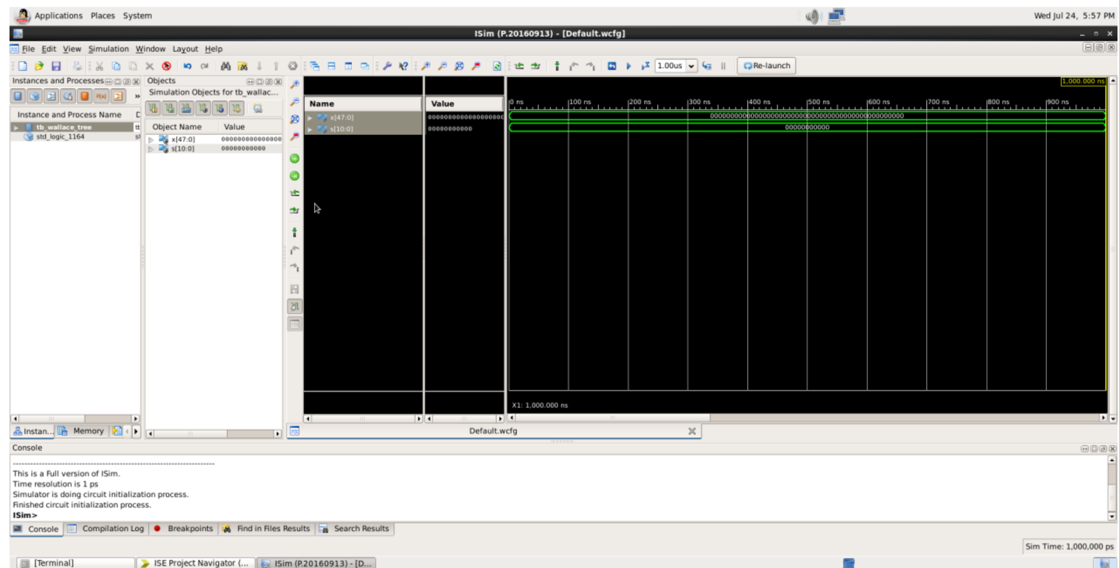
Εικόνα 60: Προβολή τεχνολογικών σχημάτων (top-level block) - Wallace Multiplier



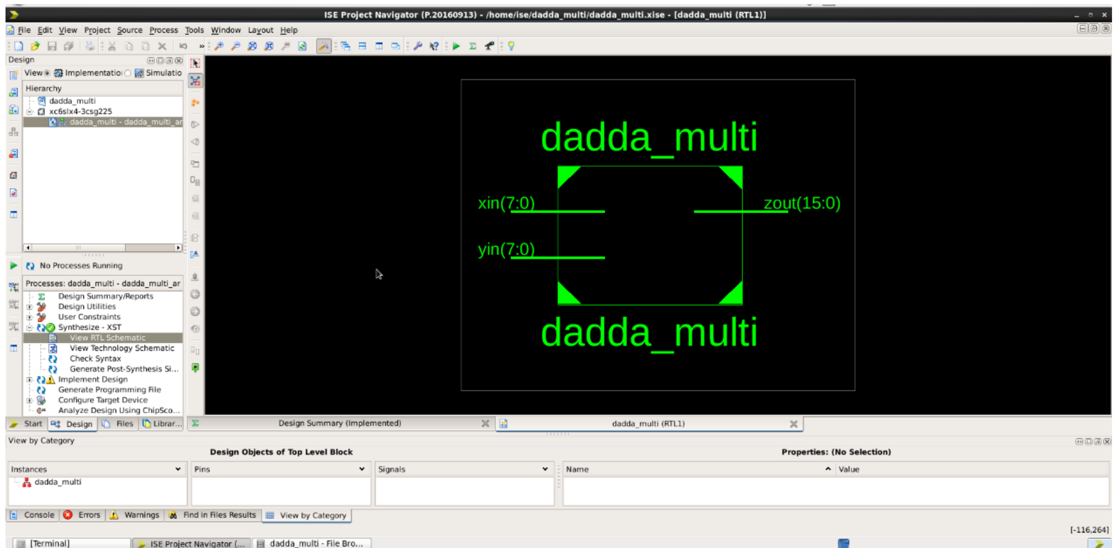
Εικόνα 61: Προβολή τεχνολογικών σχημάτων - Wallace Multiplier



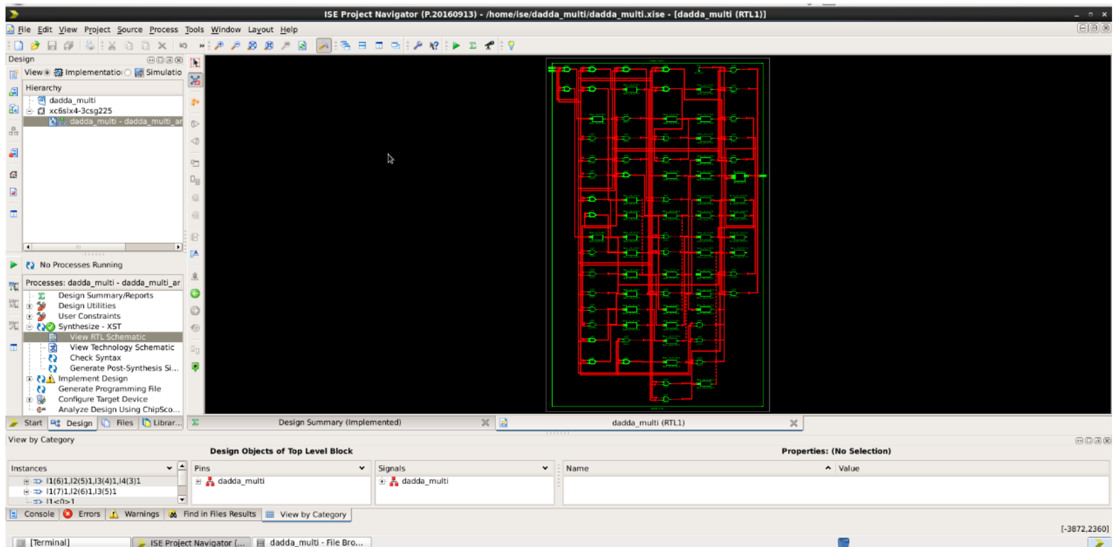
Εικόνα 62: Wallace Multiplier



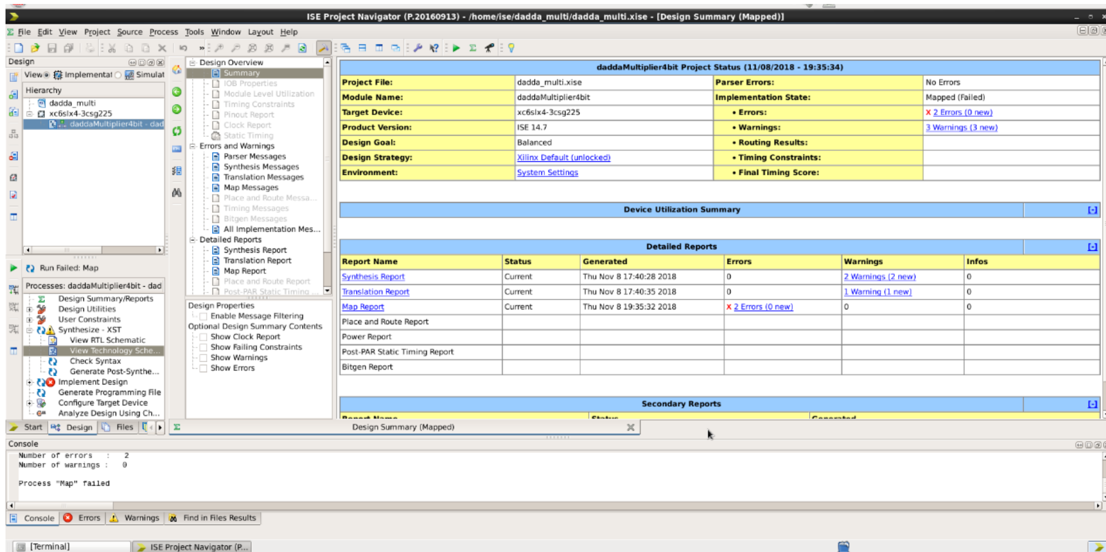
Εικόνα 63: TestBench Wallace Multiplier



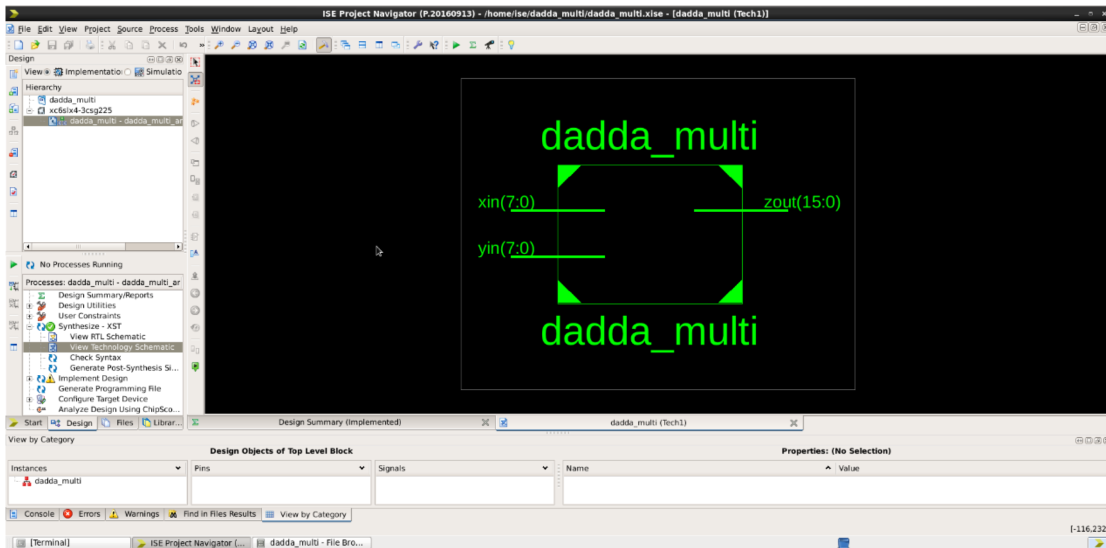
Εικόνα 64: RTL σχηματική (top-level block) - Dadda Multiplier



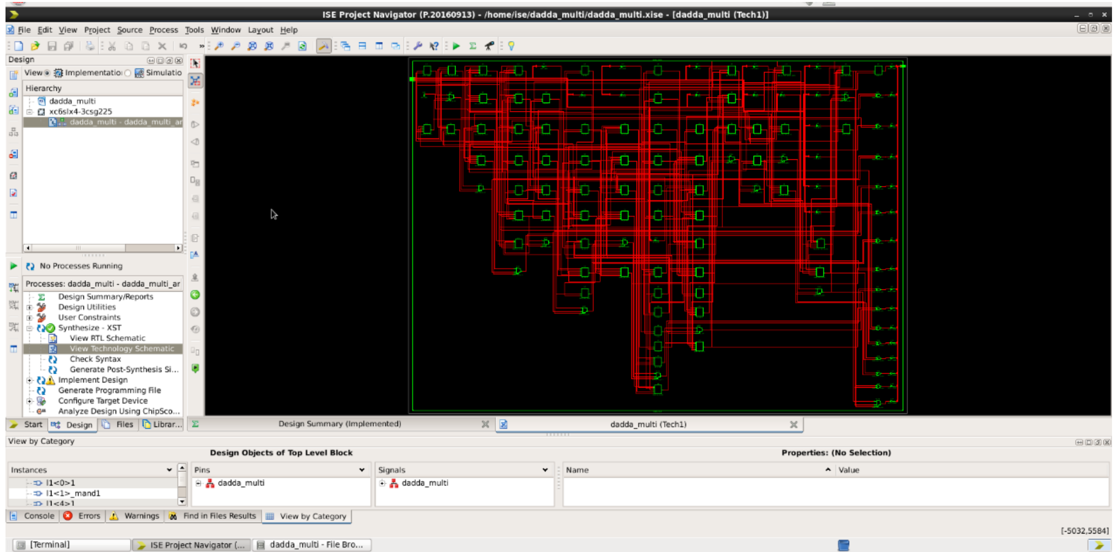
Εικόνα 65: RTL σχηματική - Dadda Multiplier



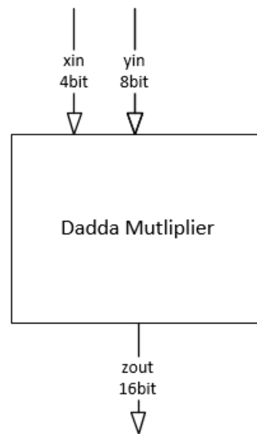
Εικόνα 66: Περίληψη σχεδίασης - Dadda Multiplier



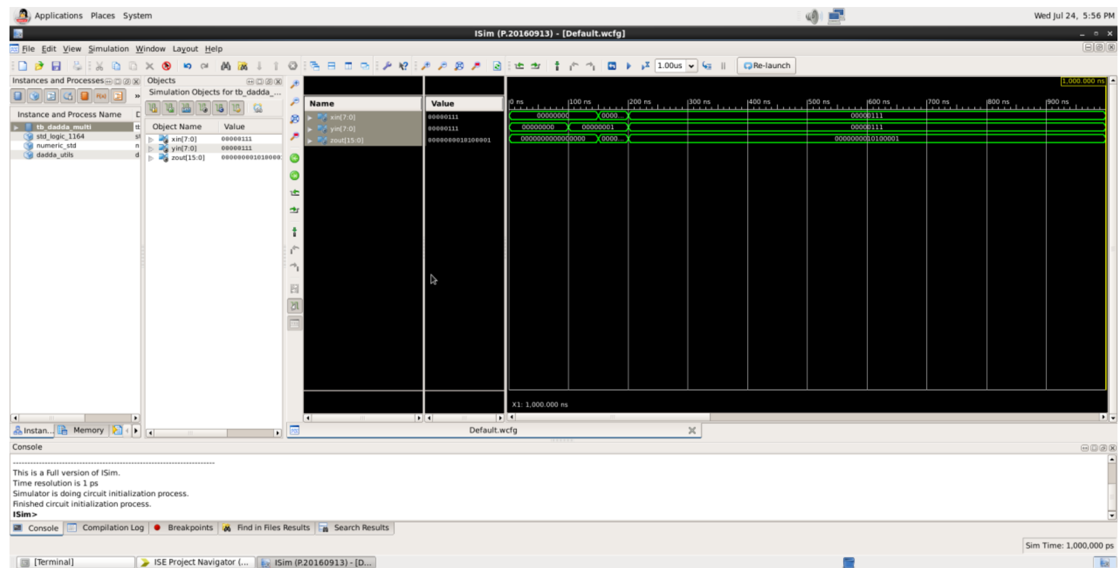
Εικόνα 67: Προβολή τεχνολογικών σχημάτων (top-level block) - Dadda Multiplier



Εικόνα 68: Προβολή τεχνολογικών σχημάτων - Dadda Multiplier

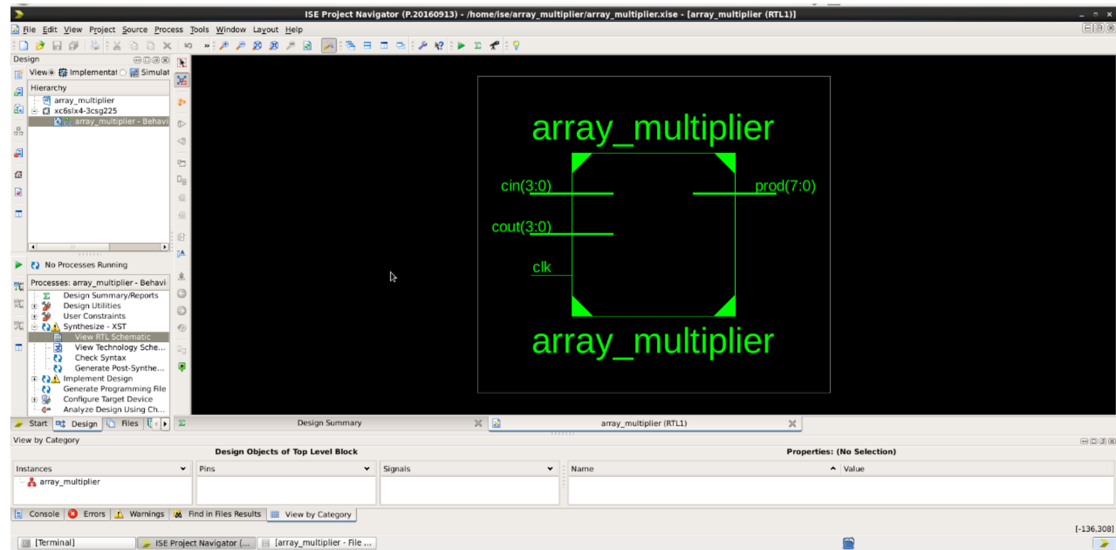


Εικόνα 69: Dadda Multiplier

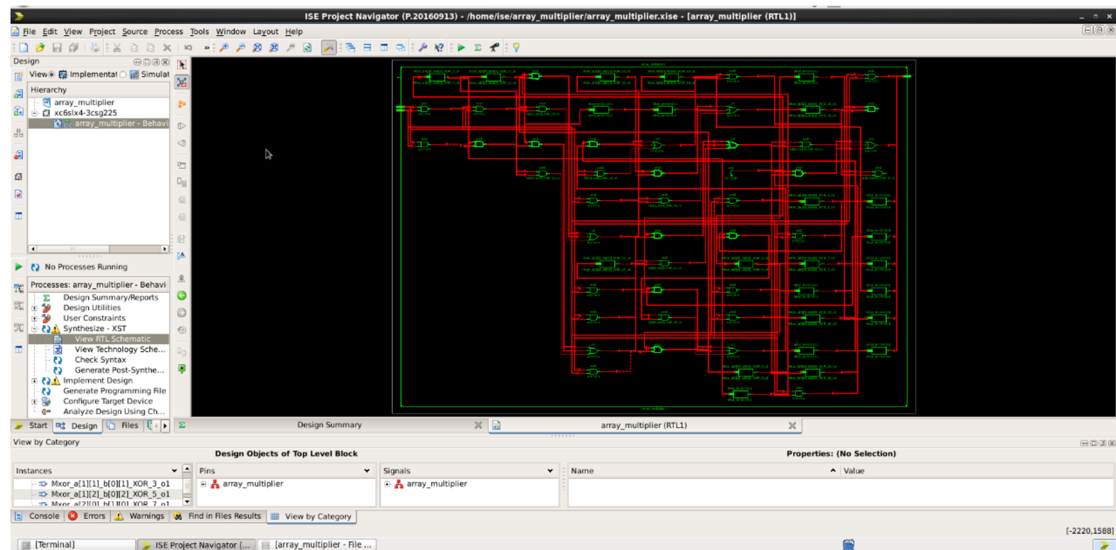


Εικόνα 70: TestBench Dadda Multiplier

Πολλαπλασιαστής με διάταξη πίνακα (ArrayMultiplier)



Εικόνα 71: RTL σχηματική (top-level block) - Array Multiplier

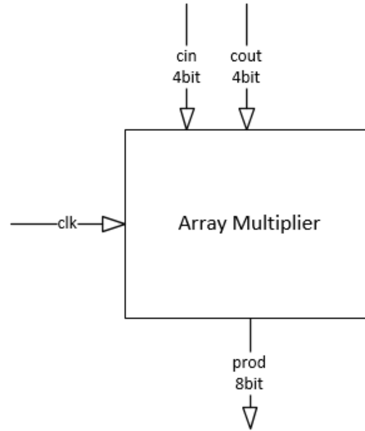


Εικόνα 72: RTL σχηματική - Array Multiplier

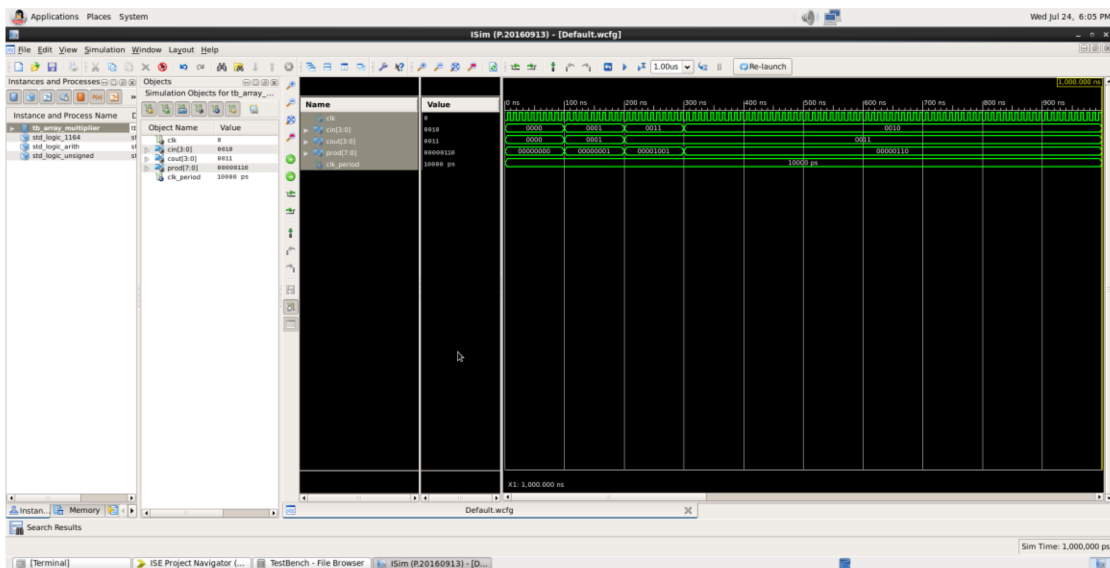
Εικόνα 73: Περίληψη σχεδίασης - Array Multiplier

Εικόνα 74: Προβολή τεχνολογικών σχημάτων (top-level block) - Array Multiplier

Εικόνα 75: Προβολή τεχνολογικών σχημάτων - Array Multiplier

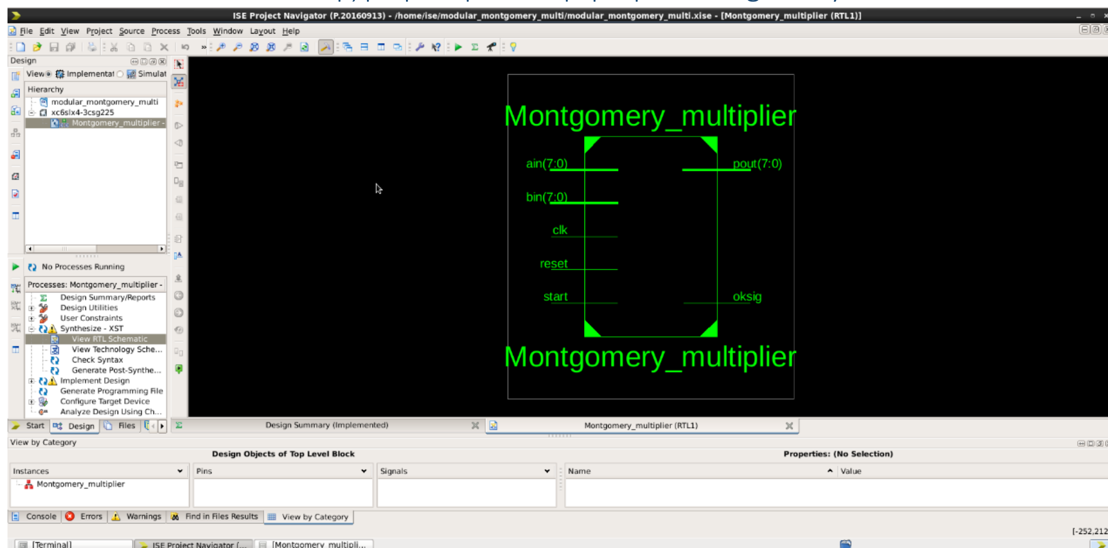


Εικόνα 76: Array Multiplier

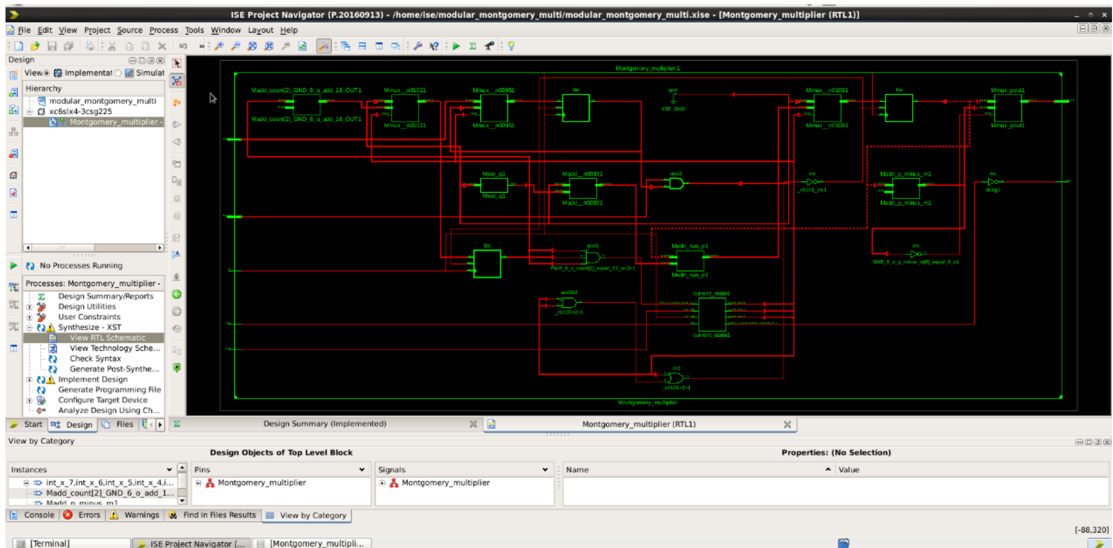


Εικόνα 77: TestBench Array Multiplier

Modular πολλαπλασιαστής με βάση τον αλγόριθμο Montgomery



Εικόνα 78: RTL σχηματική (top-level block) - Montgomery Multiplier



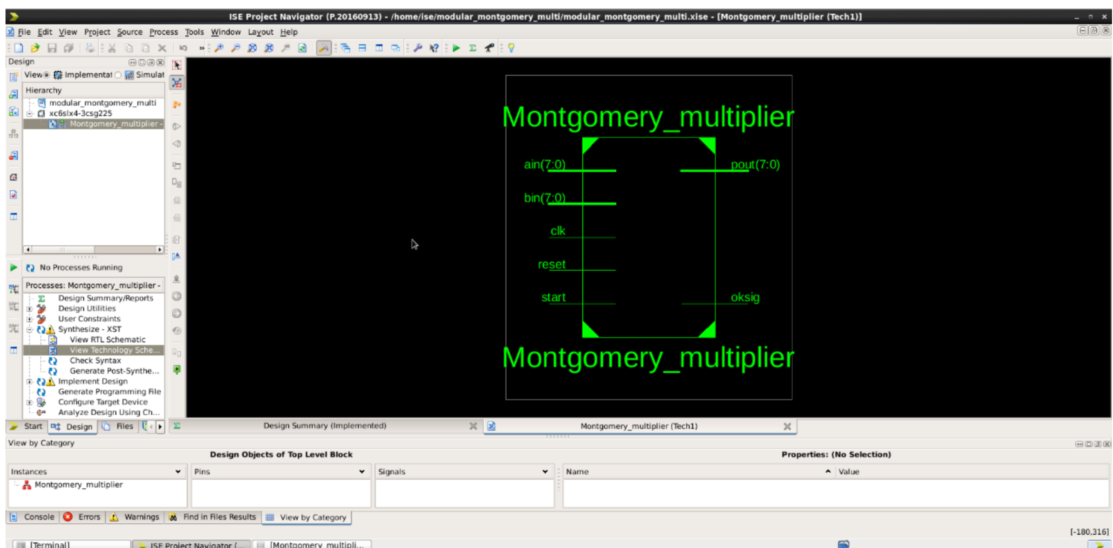
Εικόνα 79: RTL σχηματική - Montgomery Multiplier

Montgomery_multiplier Project Status (11/08/2018 - 19:15:10)			
Project File:	modular_montgomery_multi.xise	**Parser Errors:**	No Errors
Module Name:	Montgomery_multiplier	**Implementation State:**	Placed and Routed
Target Device:	xc6slx4-3csg225	• **Errors:**	No Errors
Product Version:	ISE 14.7	• **Warnings:**	1 Warning (0 new)
Design Goal:	Balanced	• **Routing Results:**	All Signals Completely Routed
Design Strategy:	Xilinx Default (unlocked)	• **Timing Constraints:**	All Constraints Met
Environment:	System Settings	• **Final Timing Score:**	0 (Timing Report)

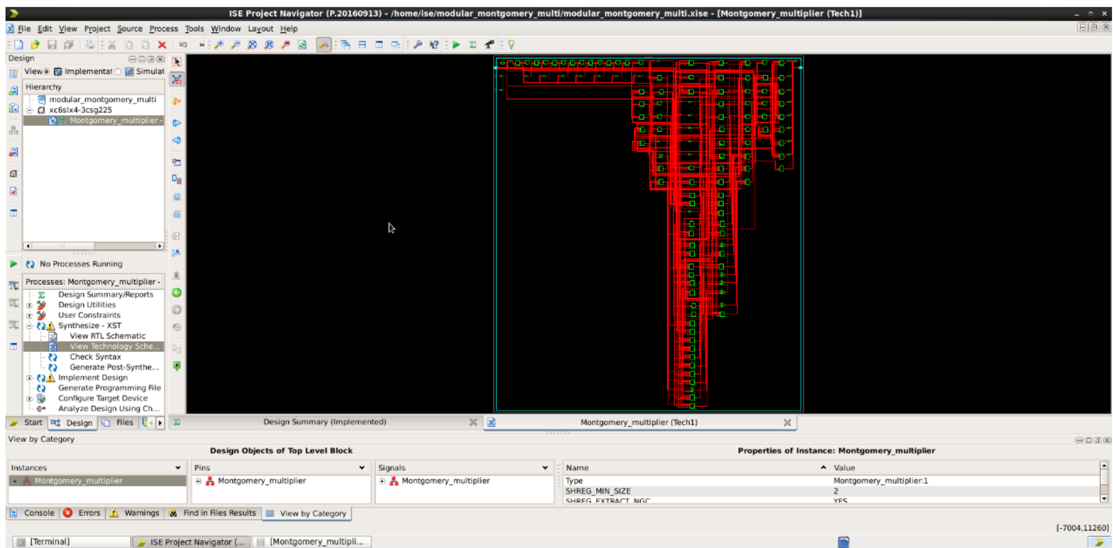
Device Utilization Summary				
Slice Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Registers	24	4,800	1%	
Number used as Flip Flops	24			
Number used as Latches	0			
Number used as Latch-thrus	0			
Number used as AND/OR logics	0			
Number of Slice LUTs	59	2,400	2%	
Number used as logic	59	2,400	2%	
Number using O6 output only	46			
Number using O5 output only	0			
Number using O5 and O6	13			
Number used as ROM	0			
Number used as Memory	0	1,200	0%	

```
Number of warnings: 0
Total time: 3 secs

Overall effort level (-ol):   Process "Generate Post-Place & Route Static Timing" completed successfully
```

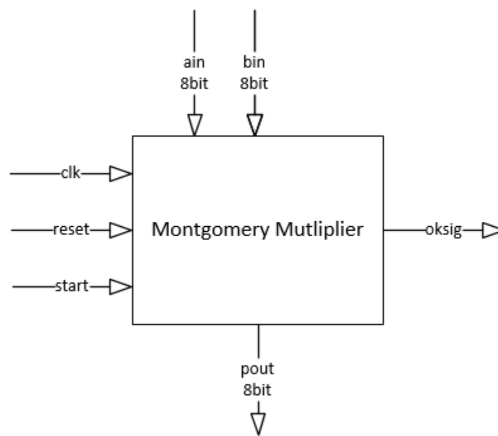
Εικόνα 80: Περίληψη σχεδίασης - Montgomery Multiplier



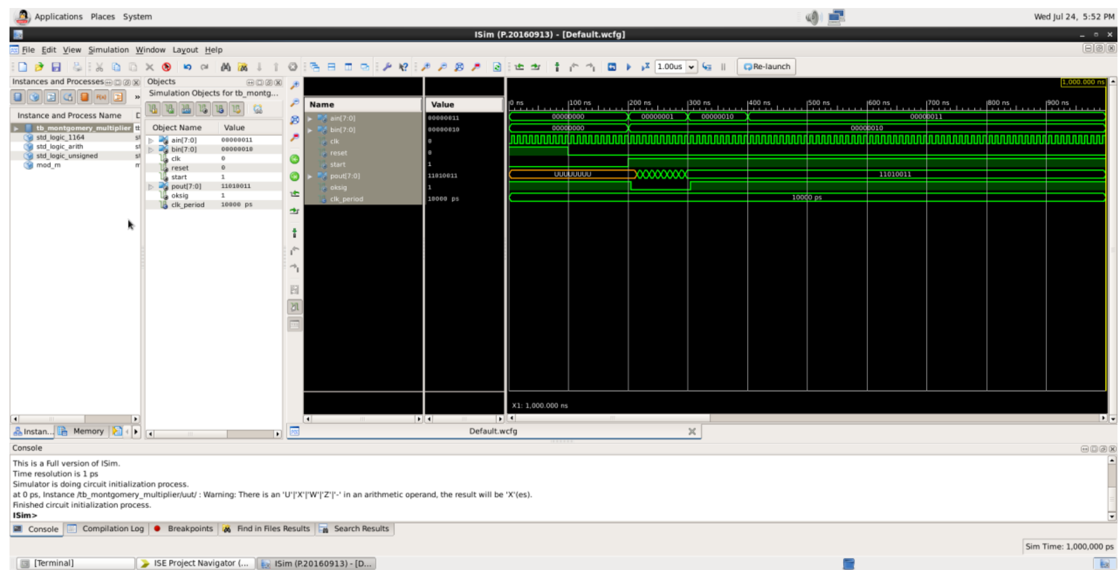
Εικόνα 81: Προβολή τεχνολογικών σχημάτων (top-level block) - Montgomery Multiplier



Εικόνα 82: Προβολή τεχνολογικών σχημάτων - Montgomery Multiplier



Εικόνα 83: Montgomery Multiplier



Εικόνα 84: TestBench Montgomery Multiplier

Pipeline

Το Pipelining είναι μια τεχνική εφαρμογής όπου πολλαπλές οδηγίες (instructions/ εντολές) αλληλεπικαλύπτονται κατά την εκτέλεση. Το pipeline στους υπολογιστές χωρίζεται σε στάδια. Κάθε στάδιο συμπληρώνει παράλληλα ένα μέρος μιας εντολής. Τα στάδια συνδέονται το ένα προς το άλλο για να σχηματίσουν ένα σωλήνα (pipeline) - οι οδηγίες εισάγονται στο ένα άκρο, προχωρούν στα στάδια και εξέρχονται στο άλλο άκρο. Αυτό επιτρέπει στα κυκλώματα ελέγχου ηλεκτρονικών υπολογιστών να εκδίδουν οδηγίες με το ρυθμό επεξεργασίας του βραδύτερου βήματος, το οποίο είναι πολύ γρηγορότερο από τον χρόνο που απαιτείται για την εκτέλεση όλων των βημάτων με τη μία. Ο όρος pipeline αναφέρεται στο γεγονός ότι κάθε βήμα μεταφέρει δεδομένα ταυτόχρονα και κάθε βήμα συνδέεται με το επόμενο.

Ο προγραμματισμός της μεταφοράς δεδομένων από το ένα στάδιο στο επόμενο στάδιο μπορεί να γίνει με τη βοήθεια ενός ρολογιού. Οι περισσότερες σύγχρονες μονάδες CPU κινούνται από ένα ρολόι. Η CPU αποτελείται εσωτερικά από λογική και flip flops. Όταν φτάσει το σήμα ρολογιού, τα flip flops παίρνουν τη νέα τους τιμή και η λογική απαιτεί τότε μια χρονική περίοδο για να αποκωδικοποιήσει τις νέες τιμές. Στη συνέχεια φτάνει ο επόμενος παλμός ρολογιού και τα flip flops παίρνουν και πάλι τις νέες τους τιμές, και ούτω καθεξής.

Το pipeline δεν μειώνει τον χρόνο για την εκτέλεση ξεχωριστών οδηγιών. Αντίθετα, αυξάνει τη διακίνηση εντολών. Η παροχή εντολών του pipeline καθορίζεται από το πόσο συχνά μια εντολή εξέρχεται από τον pipeline. Επειδή τα στάδια του pipeline συνδέονται μεταξύ τους, όλα τα στάδια πρέπει να είναι έτοιμα να προχωρήσουν ταυτόχρονα. Η απόδοση ενός επεξεργαστή ή ενός κυκλώματος που έχει υποστεί επεξεργασία με pipeline μπορεί να ποικίλει ευρέως μεταξύ διαφορετικών προγραμμάτων.

Εφαρμογή και Αποτελέσματα

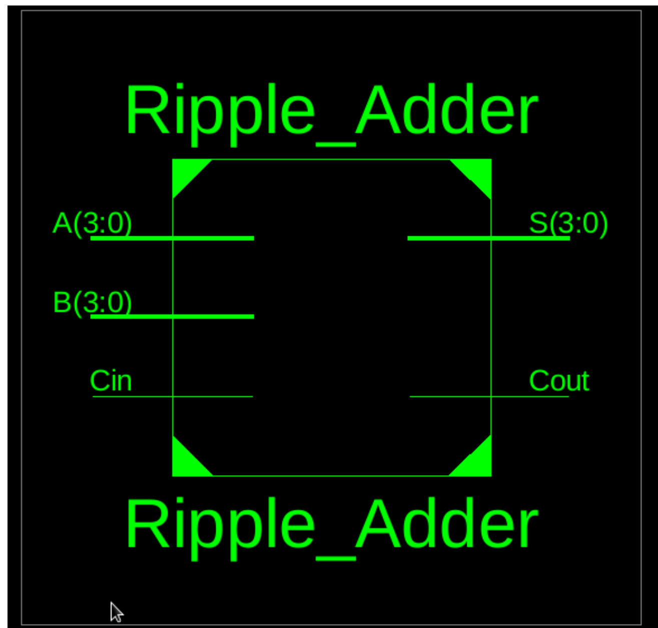
Ripple Adder με τεχνική Pipeline

Ο αθροιστής ριπής Κατασκευάζεται στην περίπτωση μας με Ν πλήρες αθροιστές αθροιστές 1- bit, δομημένοι παράλληλα. Ο ένας πλήρης αθροιστής αθροιστής 1-bit διαδέχεται τον άλλο, έτσι ώστε το κρατούμενο κρατούμενο εξόδου (carry out) από τον ένα γίνετε το κρατούμενο εισόδου (carry in carry in) του επόμενου.

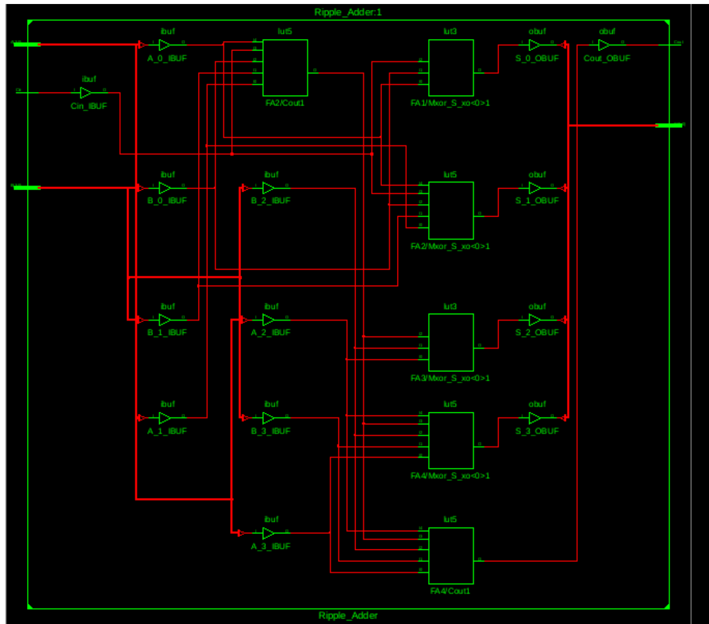
Στο σημείο αυτό είναι ακριβώς που ενδείκνυται η λογική του Pipeline

Η κατασκευή του αθροιστή είναι η παρακάτω:

Οι εισόδου και εξόδου του αθροιστή φαίνονται παρακάτω:



Η γενική του σχεδίαση φαίνεται παρακάτω:



Όπως βλεπουμε έχουμε σειρά από αθροιστές που δίνουν το αποτέλεσμα τους στον επόμενη αθροιστή . Αυτή είναι η λογική PipeLine που έχουμε από το ένα αποτέλεσμα στο άλλο.

Ο κώδικας του παραπάνω σχεδίου είναι ο παρακάτω:

```
library IEEE;

use IEEE.STD_LOGIC_1164.ALL;
```

Στο σημείο αυτό κατασκευάζεται ο πλήρης αθροιστής μας

```
entity full_adder_vhdl_code is

    Port ( A : in STD_LOGIC;

     B : in STD_LOGIC;

     Cin : in STD_LOGIC;

     S : out STD_LOGIC;

     Cout : out STD_LOGIC);

end full_adder_vhdl_code;
```

Η αρχιτεκτονική του πλήρη αθροιστή είναι η παρακάτω

```
architecture Behavioral of full_adder_vhdl_code is

begin

 S <= A XOR B XOR Cin ;
 Cout <= (A AND B) OR (Cin AND A) OR (Cin AND B) ;

end Behavioral;

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

O Ripple Adder ορίζεται παρακάτω

```
entity Ripple_Adder is
        Port ( A : in STD_LOGIC_VECTOR (3 downto 0);
        B : in STD_LOGIC_VECTOR (3 downto 0);
        Cin : in STD_LOGIC;
        S : out STD_LOGIC_VECTOR (3 downto 0);
```

```
        Cout : out STD_LOGIC);

end Ripple_Adder;
```

Καθώς και η αρχιτεκτονική του Ripple Addrer φαίνεται παρακάτω:

```
architecture Behavioral of Ripple_Adder is



component full_adder_vhdl_code

Port ( A : in STD_LOGIC;

B : in STD_LOGIC;

Cin : in STD_LOGIC;

S : out STD_LOGIC;

Cout : out STD_LOGIC);

end component;


signal c1,c2,c3: STD_LOGIC;
```

```
begin
```

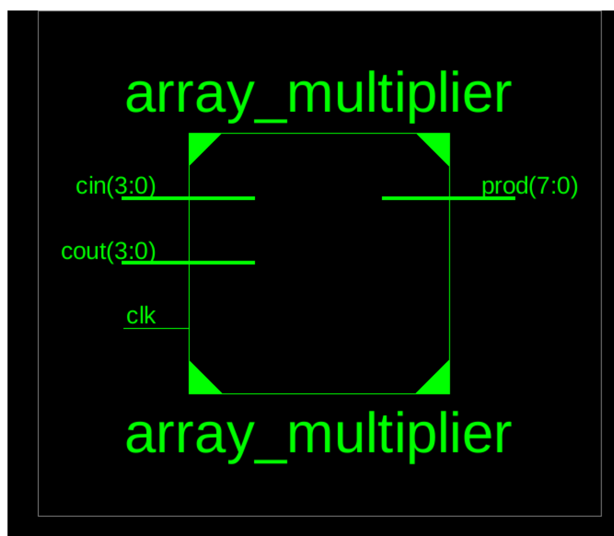
Στο σημείο αυτό έχουμε την σύνδεση μεταξύ των πλήρων αθροιστών όπου δίνουν το αποτέλεσμα του ενός στο αποτέλεσμα του άλλου

```
FA1: full_adder_vhdl_code port map( A(0), B(0), Cin, S(0), c1);

FA2: full_adder_vhdl_code port map( A(1), B(1), c1, S(1), c2);

FA3: full_adder_vhdl_code port map( A(2), B(2), c2, S(2), c3);

FA4: full_adder_vhdl_code port map( A(3), B(3), c3, S(3), Cout);

end Behavioral;
```

Array Multiplier με τεχνική Pipeline

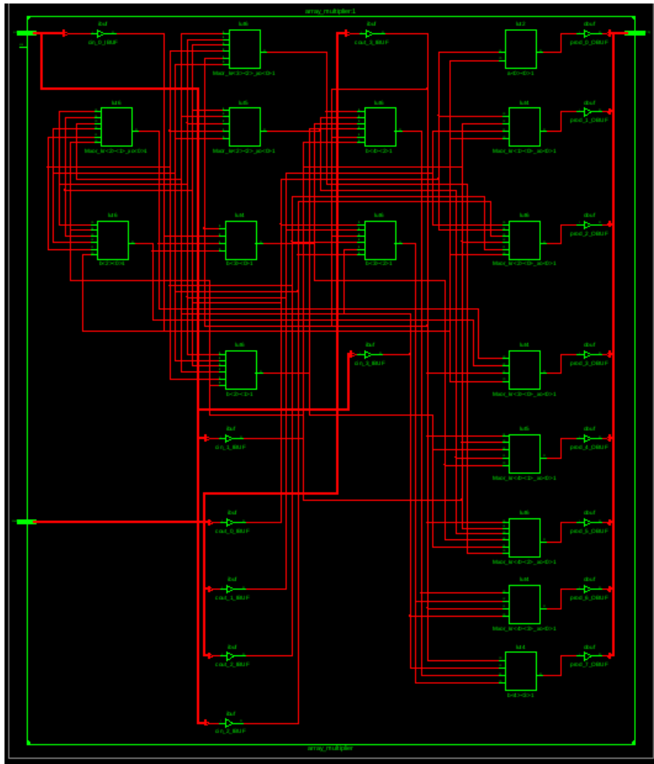
Στόχος της παραπάνω εφαρμογής είναι ο πολλαπλασιασμός πινάκων με την VHDL .

Όπως βλέπουμε στην γενική του σχεδίαση έχουμε σαν είσοδο δύο στοιχεία το Cin , Cout και το clock όπου στο cin έχουμε τα στοιχεία του πίνακα να έρχονται σταδιακά . Το prod είναι τελικά το γινόμενο του πίνακα.



Η γενική αρχιτεκτονική είναι μια σειρά από πύλες XOR , AND και OR Που στέλνουν με μια σειρά τα αποτελέσματα τους η μία στην άλλη ακολουθώντας μια λογική Pipeline.

Η σχεδίαση φαίνεται παρακάτω:



Ο κώδικας για την υλοποίηση είναι ο παρακάτω:

```
library IEEE;

use IEEE.STD_LOGIC_1164.ALL;

use IEEE.STD_LOGIC_ARITH.ALL;

use IEEE.STD_LOGIC_UNSIGNED.ALL;



Εδώ φαίνεται η γενική σχεδίαση του πολλαπλασιαστή

entity array_multiplier is

        Port ( clk:in std_logic;

                cin : in std_logic_vector(3 downto 0);

                cout : in std_logic_vector(3 downto 0);

                prod : out std_logic_vector(7 downto 0));

end array_multiplier;



architecture Behavioral of array_multiplier is



        constant n:integer :=4;
```

```
        subtype stype is std_logic_vector(n-1 downto 0);

        type krat is array(0 to n) of stype;

        signal a,b,kr:krat;


begin
```

Για κάθε σημείο του έχουμε σειρά αποτελεσμάτων από μια σειρά πυλών AND δίνοντας σαν αποτέλεσμα το a(x)(y) που αποτελεί pipeline είσοδος για τις επόμενες πύλες and και or

```
        pg:for x in 0 to n-1 generate

                pg1:for y in 0 to n-1 generate

                        a(x)(y)<=cin(y) and cout(x);

                end generate;

                b(0)(x)<='0';

        end generate;


        kr(0)<=a(0);

        prod(0)<=a(0)(0);
```

```
addr:for x in 1 to n-1 generate

    addc:for y in 0 to n-2 generate
```

Στην συνέχεια έχουμε την χρήση των αποτελεσμάτων a(x)(y) όπου γίνονται χρήση σε σειρά από πύλες xor και and δίνοντας αποτελέσματα kr(x)(y) και γίνονται σειριακά pipeline με πύλες or και and όπου προκύπτει το b(x)(y).

```
        kr(x)(y)<=a(x)(y) xor b(x-1)(y) xor kr(x-1)(y+1);

        b(x)(y)<=(a(x)(y) and b(x-1)(y)) or

                      (a(x)(y) and kr(x-1)(y+1)) or

            (b(x-1)(y)and kr(x-1)(y+1));

    end generate;

    prod(x)<=kr(x)(0);

    kr(x)(n-1)<=a(x)(n-1);

end generate;



b(n)(0)<='0';
```

Αντίστοιχα γίνεται και στο σημείο αυτό κάνοντας χρήση των προηγούμενων αποτελεσμάτων και παράγοντας τα νέα kr και b

```
addlast:for y in 1 to n-1 generate

    kr(n)(y)<=b(n)(y-1) xor b(n-1)(y-1) xor kr(n-1)(y);

    b(n)(y)<=(b(n)(y-1) and b(n-1)(y-1)) or

                        (b(n)(y-1) and kr(n-1)(y)) or

             (b(n-1)(y-1)and kr(n-1)(y));

end generate;
```

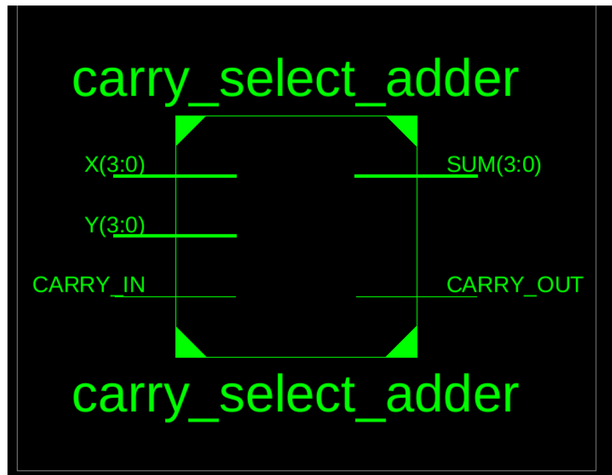
τα αποτελέσματα των νέων b και kr δίνουν το αποτέλεσμα prod

```
prod(2*n-1)<=b(n)(n-1);

prod(2*n-2 downto n)<=kr(n)(n-1 downto 1);



end Behavioral;
```

Carry Select Adder με τεχνική Pipeline

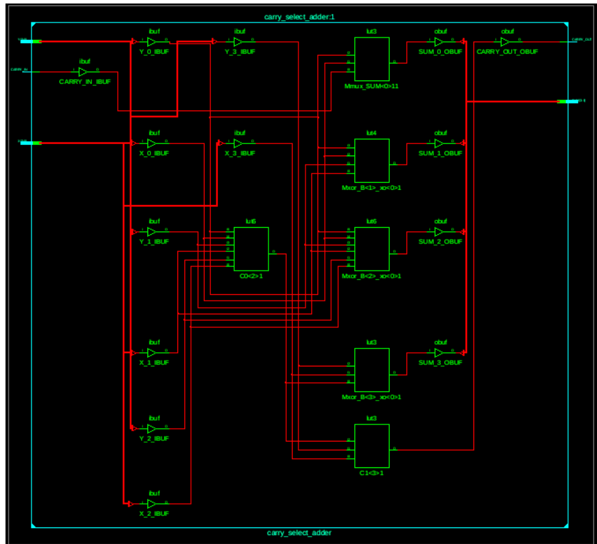
Η αρχή που στηρίζεται ο Carry-Select Adder είναι υπολογίζουμε τα αθροίσματα για δύο περιπτώσεις, μία για κρατούμενο εισόδου ίσο με 0 και μία για κρατούμενο εισόδου ίσο με 1. Ύστερα θα διαλέγουμε ένα από τα δύο, όταν θα έχουμε στη διάθεση μας το κρατούμενο. Αυτό που κάνουμε είναι να χωρίσουμε τον αθροιστή σε ομάδες των m bits, και να υπολογίσουμε για κάθε ομάδα δύο υπό συνθήκη (conditional) αθροίσματα και κρατούμενα εξόδου.

Η γενική αρχιτεκτονική του αθροιστή είναι η παρακάτω:



Όπως βλέπουμε έχουμε το X , Υ που θέλουμε να κάνουμε άθροιση και το κρατούμενο και σαν αποτέλεσμα έχουμε το άθροισμα και το κρατούμενο εξόδου.

Η αρχιτεκτονική του αθροιστή φαίνεται παρακάτω:



Και σε αυτή την περίπτωση έχουμε σειρά από πύλες XOR AND και OR που παίρνουμε τα αντίστοιχα αποτελέσματα και τα μέταφέρουμε σε άλλες πύλες. Στην περίπτωση μας έχουμε και μια σειρά από procedures που αποτελούν την pipeline λογική.

Ο κώδικας VHDL φαίνεται παρακάτω:

```
library IEEE;

use IEEE.STD_LOGIC_1164.ALL;
```

Η βασική σχεδίαση του αθροιστή φαίνεται παρακάτω:

```
entity carry_select_adder is

        Port ( X : in STD_LOGIC_VECTOR (3 downto 0);

        Y : in STD_LOGIC_VECTOR (3 downto 0);

        CARRY_IN : in STD_LOGIC;

        SUM : out STD_LOGIC_VECTOR (3 downto 0);

        CARRY_OUT : out STD_LOGIC);

end carry_select_adder;



architecture Behavioral of carry_select_adder is



signal A,B,C0,C1: STD_LOGIC_VECTOR( 3 DOWNTO 0);

begin
```

Η βασική λογική είναι να παράγουμε τα bit των A , B, C και τα αντίστοιχα κρατούμενα που δίνονται από τις αντίστοιχες procedures για το αποτέλεσμα .

Κάθε bit που παράγεται δίνεται σαν είσοδο για την παραγωγή των άλλων ακολουθώντας έτσι την Pipeline λογική.

A(0)<= X(0) XOR Y(0) XOR '0' ;

C0(0) <= (X(0) AND Y(0)) OR ('0' AND X(0)) OR ('0' AND Y(0)) ;

A(1)<= X(1) XOR Y(1) XOR C0(0) ;

C0(1) <= (X(1) AND Y(1)) OR (C0(0) AND X(1)) OR (C0(0) AND Y(1)) ;

A(2)<= X(2) XOR Y(2) XOR C0(2) ;

C0(2) <= (X(2) AND Y(2)) OR (C0(1) AND X(2)) OR (C0(1) AND Y(2)) ;

A(3)<= X(3) XOR Y(3) XOR C0(2) ;

C0(3) <= (X(3) AND Y(3)) OR (C0(2) AND X(3)) OR (C0(2) AND Y(3)) ;

B(0)<= X(0) XOR Y(0) XOR '1' ;

C1(0) <= (X(0) AND Y(0)) OR ('0' AND X(0)) OR ('0' AND Y(0)) ;

B(1)<= X(1) XOR Y(1) XOR C1(0) ;

```
        C1(1) <= (X(1) AND Y(1)) OR (C1(0) AND  X(1)) OR (C1(0) AND
Y(1)) ;

        B(2)<= X(2) XOR Y(2) XOR C0(2) ;

        C1(2) <= (X(2) AND Y(2)) OR (C1(1) AND  X(2)) OR (C1(1) AND
Y(2)) ;

        B(3)<= X(3) XOR Y(3) XOR C0(2) ;

        C1(3) <= (X(3) AND Y(3)) OR (C1(2) AND  X(3)) OR (C1(2) AND
Y(3)) ;
```

Για κάθε bit ορίζονται και οι παρακάτω συναρτήσεις όπου δίνουν το κρατούμενο

```
process(C0(3),C1(3),CARRY_IN)

begin

        if CARRY_IN = '0' then

                CARRY_OUT <= C0(3);

        else

                CARRY_OUT <= C1(3);

        end if;

end process;
```

```
process(A(0),B(0),CARRY_IN)
begin
        if CARRY_IN = '0' then
                SUM(0) <= A(0);
        else
                SUM(0) <= B(0);
        end if;
end process;

process(A(1),B(1),CARRY_IN)
begin
        if CARRY_IN = '0' then
                SUM(1) <= A(1);
        else
                SUM(1) <= B(1);
        end if;
end process;
```

```
process(A(2),B(2),CARRY_IN)
begin
        if CARRY_IN = '0' then
                SUM(2) <= A(2);
        else
                SUM(2) <= B(2);
        end if;
end process;


process(A(3),B(3),CARRY_IN)
begin
        if CARRY_IN = '0' then
                SUM(3) <= A(3);
        else
                SUM(3) <= B(3);
        end if;
end process;
```

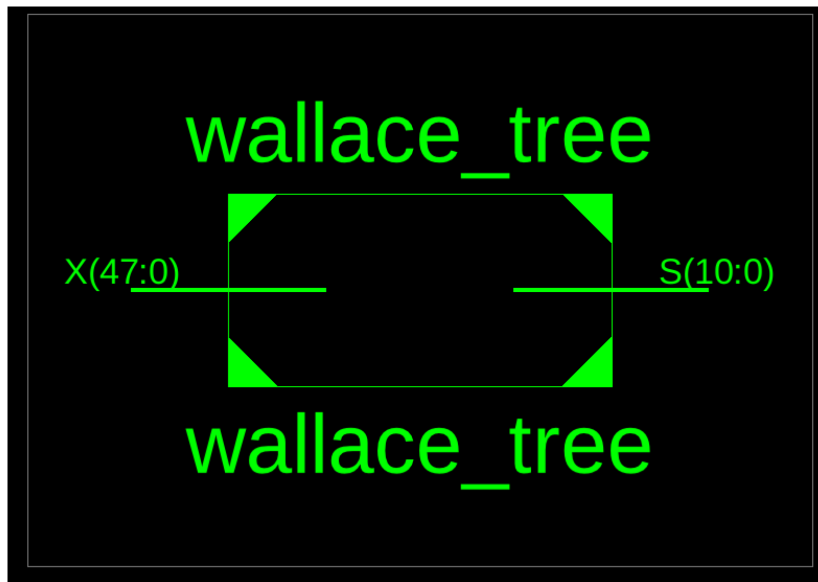
```
end Behavioral;
```

Wallace Tree με τεχνική Pipeline

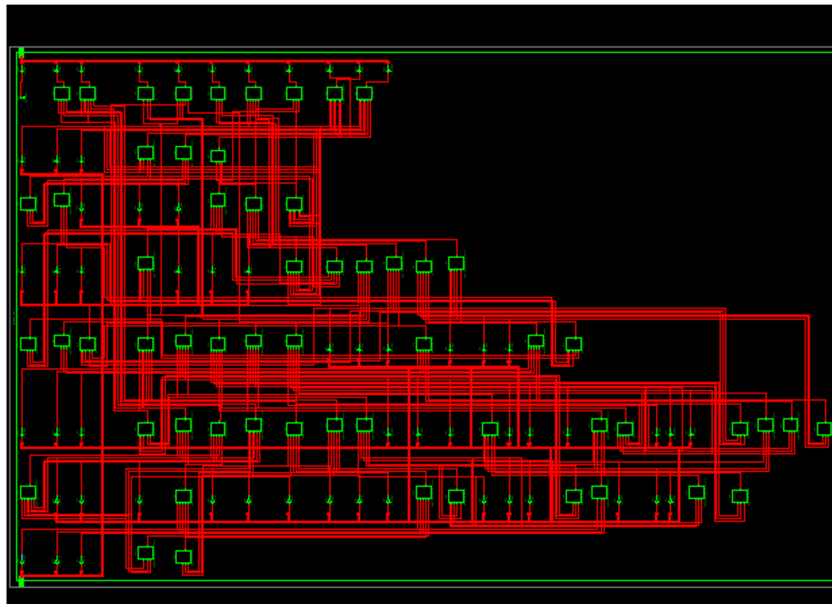
Οι δενδρικοί πολλαπλασιαστές χρησιμοποιούν στόχο έχουν την παραγωγή κυκλωμάτων πολλαπλασιασμού με την ελάχιστη καθυστέρηση. Η δημιουργία των μερικών γινομένων στους δενδρικούς πολλαπλασιαστές γίνεται ανεξάρτητα από το στάδιο της πρόσθεσης. Παράγονται αρχικά τα μερικά γινόμενα και οδηγούν σε ένα δίκτυο πλήρων αρθροιστών και ημιαθριστών από το οποίο παράγεται ένας αριθμός σε μορφή άθροισμα-κρατουμένου. Στο τελικό στάδιο, μπορεί να χρησιμοποιηθεί ένας αθροιστής διάδοσης κρατουμένου ή πρόβλεψης κρατουμένου για να παραχθεί το τελικό αποτέλεσμα σε δυαδική μορφή. Επειδή η δημιουργία των μερικών γινομένων γίνεται ανεξάρτητα από το στάδιο της δενδρικής συμπίεσης, είναι δυνατή η κωδικοποίηση του ενός από τους δύο αριθμούς ή και των δυο με βάση κάποιο άλλο σύστημα. Με αυτόν τον τρόπο μπορεί να μειωθεί ο αριθμός των μερικών γινομένων έτσι ώστε να αυξηθεί η ταχύτητα του δενδρικού πολλαπλασιαστή.

Στόχος του πολλαπλασιαστή είναι η συμπίεση των μερικών γινομένων μετά από όσο το δυνατόν λιγότερα επίπεδα FA και HA με συνέπεια το μικρότερο δυνατό χρόνο. Για το σκοπό αυτό, σε κάθε επίπεδο, τα ψηφία ίδιου βάρους ομαδοποιούνται ανά τρία και εισέρχονται σαν είσοδοι σε έναν FA, εάν περισσέψουν δύο, τότε ομαδοποιούνται ανά δύο και εισέρχονται ως είσοδοι σε έναν HA, ενώ αν περισσέψει μόνο ένα, μεταφέρεται στο επόμενο επίπεδο.

Η γενική αρχιτεκτονική φαίνεται παρακάτω:



και η λεπτομερή αρχιτεκτονική παρακάτω:



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

Εδώ έχουμε τον πλήρη αθροιστή

```
entity full_adder_H is
        Port ( A : in STD_LOGIC;
         B : in STD_LOGIC;
         Cin : in STD_LOGIC;
         S : out STD_LOGIC;
         Cout : out STD_LOGIC);
end full_adder_H;

architecture Behavioral of full_adder_H is

begin

 S <= A XOR B XOR Cin ;
 Cout <= (A AND B) OR (Cin AND A) OR (Cin AND B) ;

end Behavioral;
-------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
```

Εδώ έχουμε τον ημιαθροιστή

```
entity half_adder_H is
   Port ( A : in  STD_LOGIC;
          B : in  STD_LOGIC;
          C : out  STD_LOGIC;
          S : out  STD_LOGIC);
end half_adder_H;

architecture Behavioral of half_adder_H is

begin

S <= A xor B;
C <= A and B;

end Behavioral;
--------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
```

Η αρχιτεκτονική του πολλαπλασιαστή φαίνεται παρακάτω:

```
entity part_of_wallace_tree is
           Port (
           Ci : in STD_LOGIC_VECTOR(2 downto 0);
           X : in STD_LOGIC_VECTOR(5 downto 0);
           Co : out STD_LOGIC_VECTOR(2 downto 0);
           S0 : out STD_LOGIC;
```

```
	S1 : out STD_LOGIC);
end part_of_wallace_tree;

architecture Behavioral of part_of_wallace_tree is

component full_adder_H
	Port ( A : in STD_LOGIC;
	B : in STD_LOGIC;
	Cin : in STD_LOGIC;
	S : out STD_LOGIC;
	Cout : out STD_LOGIC);
end component;

signal W : STD_LOGIC_VECTOR(2 downto 0);

begin
```

Εδώ έχουμε την συνδεσμολογία του αθροιστών

```
FA1: full_adder_H port map(  X(0), X(1) , X(2), Co(0), W(0));
FA2: full_adder_H port map( X(3), X(4) , X(5),  Co(1), W(1));
FA3: full_adder_H port map( W(0), W(1) , Ci(0), Co(2), W(2) );
FA4: full_adder_H port map( W(2),  Ci(1), Ci(2) , S1   , S0  );

end Behavioral;
--------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
```

Αντίστοιχα έχουμε το Prop_adder που έχουμε μια σειρά από ημιαθροιστές και αθροιστές

```
entity Pprop_adder_H is
   Port ( X : in  STD_LOGIC_VECTOR (7 downto 0);
          Y : in  STD_LOGIC_VECTOR (7 downto 0);
          S : out  STD_LOGIC_VECTOR (8 downto 0));
end Pprop_adder_H;

architecture Behavioral of Pprop_adder_H is

component half_adder_H
   Port ( A : in  STD_LOGIC;
          B : in  STD_LOGIC;
          C : out  STD_LOGIC;
          S : out  STD_LOGIC);
end component;

component full_adder_H
            Port ( A : in STD_LOGIC;
             B : in STD_LOGIC;
             Cin : in STD_LOGIC;
             S : out STD_LOGIC;
             Cout : out STD_LOGIC);
end component;
signal W : STD_LOGIC_VECTOR(6 downto 0);
begin
```

```
HA : half_adder_H port map(X(0),Y(0),W(0),S(0));
FA0: full_adder_H port map(X(1) ,Y(1) ,W(0),S(1),W(1));
FA1: full_adder_H port map(X(2) ,Y(2) ,W(1),S(2),W(2));
FA2: full_adder_H port map(X(3) ,Y(3) ,W(2),S(3),W(3));
FA3: full_adder_H port map(X(4) ,Y(4) ,W(3),S(4),W(4));
FA4: full_adder_H port map(X(5) ,Y(5) ,W(4),S(5),W(5));
FA5: full_adder_H port map(X(6) ,Y(6) ,W(5),S(6),W(6));
FA6: full_adder_H port map(X(7) ,Y(7) ,W(6),S(7),S(8));

end Behavioral;
----------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
```

Τελικά ο πολλαπλασιαστής wallece tree είναι ο παρακάτω:

```
entity wallace_tree is
   Port ( S : out  STD_LOGIC_VECTOR (10 downto 0);
          X : in  STD_LOGIC_VECTOR (47 downto 0));
end wallace_tree;

architecture Behavioral of wallace_tree is

component Pprop_adder_H
   Port ( X : in  STD_LOGIC_VECTOR (7 downto 0);
          Y : in  STD_LOGIC_VECTOR (7 downto 0);
          S : out  STD_LOGIC_VECTOR (8 downto 0));
```

```
end component;

component part_of_wallace_tree
        Port (
                Ci : in STD_LOGIC_VECTOR(2 downto 0);
                X : in STD_LOGIC_VECTOR(5 downto 0);
                Co : out STD_LOGIC_VECTOR(2 downto 0);
                S0 : out STD_LOGIC;
                S1 : out STD_LOGIC);
end component;

component half_adder_H
   Port ( A : in  STD_LOGIC;
        B : in  STD_LOGIC;
        C : out  STD_LOGIC;
        S : out  STD_LOGIC);
end component;

component full_adder_H
        Port ( A : in STD_LOGIC;
                B : in STD_LOGIC;
                Cin : in STD_LOGIC;
                S : out STD_LOGIC;
                Cout : out STD_LOGIC);
end component;

signal W3 : STD_LOGIC_VECTOR(8 downto 0);
signal W2 : STD_LOGIC_VECTOR(8 downto 0);
```

```
signal WC0 : STD_LOGIC_VECTOR(2 downto 0);
signal WC1 : STD_LOGIC_VECTOR(2 downto 0);
signal WC2 : STD_LOGIC_VECTOR(2 downto 0);
signal WC3 : STD_LOGIC_VECTOR(2 downto 0);
signal WC4 : STD_LOGIC_VECTOR(2 downto 0);
signal WC5 : STD_LOGIC_VECTOR(2 downto 0);
signal WC6 : STD_LOGIC_VECTOR(2 downto 0);
signal WC7 : STD_LOGIC_VECTOR(2 downto 0);
signal X1,Y1 : STD_LOGIC_VECTOR(7 downto 0);
signal R1 : STD_LOGIC_VECTOR(8 downto 0);
begin
```

Εδώ παρατηρούμε την λογική της συνδεσμόλογίας του κάθε part όπου τελικά αποτελεί κι την βασική λογική του pipe line αφού τελικά παράγει το καθένα ένα ξεχωριστό αποτέλεσμα

```
--FA1: full_adder_H port map( Co(0), W(0), X(0), X(1) , X(2));
PART0 : part_of_wallace_tree port map("000"   , X(5 downto 0),  WC0, W3(0), W2(0));
PART1 : part_of_wallace_tree port map(WC0 , X(11 downto 6), WC1, W3(1), W2(1));
PART2 : part_of_wallace_tree port map(WC1 , X(17 downto 12), WC2, W3(2), W2(2));
PART3 : part_of_wallace_tree port map(WC2 , X(23 downto 18), WC3, W3(3), W2(3));
PART4 : part_of_wallace_tree port map(WC3 , X(29 downto 24), WC4, W3(4), W2(4));
```

```
PART5 : part_of_wallace_tree port map(WC4 , X(35 downto 30), WC5, W3(5), W2(5));
PART6 : part_of_wallace_tree port map(WC5 , X(41 downto 36), WC6, W3(6), W2(6));
PART7 : part_of_wallace_tree port map(WC6 , X(47 downto 42), WC7, W3(7), W2(7));



FA   : full_adder_H port map(WC7(2), WC7(1), WC7(0), W3(8), W2(8));
X1 <= W2(7 downto 0);
Y1 <= W3(8 downto 1);
PROP : Pprop_adder_H port map(X1,Y1,R1);
HA : half_adder_H port map (W2(8),R1(8),S(10),S(9));
        S(8) <= R1(8);
        S(7) <= R1(7);
        S(6) <= R1(6);
        S(5) <= R1(5);
        S(4) <= R1(4);
        S(3) <= R1(3);
        S(2) <= R1(2);
        S(1) <= R1(1);
        S(0) <= W3(0);

end Behavioral;
```

Βιβλιογραφία

Aditya Kumar Singh, B. P. (2012). *Design and Comparison of Multipliers Using Different Logic Styles.*

Chinmay Kumar Behera, S. K. (2014). *DESIGN OF BOOTH MULTIPLIER USING RIPPLE CARRY ADDER.*

GHOSH, M. (2007). *DESIGN AND IMPLEMENTATION OF DIFFERENT MULTIPLIERS USING VHDL.*

M. Bečvář, P. Š. (2005). *Fixed-Point Arithmetic in FPGA.*

MOHANTY, P. S. (2009). *DESIGN AND IMPLEMENTATION OF FASTER AND LOW POWER MULTIPLIERS.*

Pokhrel, K. C. (1997). *A Cornparison of Bit-Serial Muhipliers Using VHDL Based Logic Synthesizers.*

Ελπίδα, Ν. (2014). *Υλοποίηση ενός μικροεπεξεργαστή με VHDL κώδικα.*

Ιωάννης, Κ. (2012). *Κυκλώματα Αριθμητικής Υπολοίπων με Χαμηλή Κατανάλωση και Ανοχή σε Διακυμάνσεις Παραμέτρων.*

Κομνηνός, Δ. (2014). *Αρχιτεκτονικές για το ψηφιακό αριθμητικό υλικό.*

Χαραλαμπίδου Κούρφαλη Αλεξάνδρα, Χ. Ν. (2012). *Σχεδίαση και υλοποίηση παράλληλου, ενσωματωμένου συστήματος σε πλατφόρμα FPGA για την εκτέλεση του αλγορίθμού γενικευμένου προβλεπτικού ελέγχου.*

Χατζηεμμανουήλ, Σ. (2016). *Βασικά αριθμητικά ψηφιακά κυκλώματα.*

Παράρτημα A – Κώδικες Κυκλωμάτων

Σειριακός αθροιστής (BitSerialAdder)

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

entity serial_adder is
      Port (
            I_m : in std_logic_vector(7 downto 0);
            I_l : in std_logic_vector(7 downto 0);
            O : out std_logic;
            c_i : in std_logic;
            c_o : out std_logic;
            load : in std_logic;
            CLK : in std_logic
            );
end serial_adder;

architecture Behavioral of serial_adder is
      signal r_reg : std_logic;
      signal reg_i_mv : std_logic_vector(6 downto 0);
      signal reg_i_lv : std_logic_vector(6 downto 0);
      signal reg_i_m,reg_i_l,ci_reg,co:std_logic;
begin

process (CLK)
begin
   if CLK'event and CLK='1' then
      if (load='1') then
         reg_i_mv <= I_m(7 downto 1);
         reg_i_lv <= I_l(7 downto 1);
                 O <= I_m(0) XOR I_l(0) XOR c_i ;
                 ci_reg <= (I_m(0) AND I_l(0)) OR (c_i AND
I_m(0)) OR (c_i AND I_l(0)) ;
      else
         reg_i_mv <= '0' & reg_i_mv(6 downto 1);
         reg_i_lv <= '0' & reg_i_lv(6 downto 1);
                 O <= reg_i_mv(0) XOR reg_i_lv(0) XOR ci_reg ;
                 ci_reg <= (reg_i_mv(0) AND reg_i_lv(0)) OR
(ci_reg AND  reg_i_mv(0)) OR (ci_reg AND reg_i_lv(0)) ;
      end if;
   end if;
end process;
c_o <= ci_reg;
end Behavioral;
```

Αθροιστής μετάδοσης κρατουμένου (RCA – RippleCarryAdder)

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

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
```

```
--library UNISIM;
--use UNISIM.VComponents.all;

entity full_adder_vhdl_code is
      Port ( A : in STD_LOGIC;
       B : in STD_LOGIC;
       Cin : in STD_LOGIC;
       S : out STD_LOGIC;
       Cout : out STD_LOGIC);
end full_adder_vhdl_code;

architecture Behavioral of full_adder_vhdl_code is

begin

 S <= A XOR B XOR Cin ;
 Cout <= (A AND B) OR (Cin AND A) OR (Cin AND B) ;

end Behavioral;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity Ripple_Adder is
      Port ( A : in STD_LOGIC_VECTOR (3 downto 0);
      B : in STD_LOGIC_VECTOR (3 downto 0);
      Cin : in STD_LOGIC;
      S : out STD_LOGIC_VECTOR (3 downto 0);
      Cout : out STD_LOGIC);
end Ripple_Adder;

architecture Behavioral of Ripple_Adder is


-- Full Adder VHDL Code Component Decalaration
component full_adder_vhdl_code
Port ( A : in STD_LOGIC;
B : in STD_LOGIC;
Cin : in STD_LOGIC;
S : out STD_LOGIC;
Cout : out STD_LOGIC);
end component;

-- Intermediate Carry declaration
signal c1,c2,c3: STD_LOGIC;

begin

-- Port Mapping Full Adder 4 times
```

```
FA1: full_adder_vhdl_code port map( A(0), B(0), Cin, S(0), c1);
FA2: full_adder_vhdl_code port map( A(1), B(1), c1, S(1), c2);
FA3: full_adder_vhdl_code port map( A(2), B(2), c2, S(2), c3);
FA4: full_adder_vhdl_code port map( A(3), B(3), c3, S(3), Cout);


end Behavioral;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity Ripple_Adder_R is
    Port (  clk : in STD_LOGIC;
                       reset : in STD_LOGIC;
                       ABc : in  STD_LOGIC_VECTOR (8 downto 0);
                       C : out  STD_LOGIC_VECTOR (4 downto 0));
end Ripple_Adder_R;

architecture Behavioral of Ripple_Adder_R is

component Ripple_Adder is
      Port ( A : in STD_LOGIC_VECTOR (3 downto 0);
      B : in STD_LOGIC_VECTOR (3 downto 0);
      Cin : in STD_LOGIC;
      S : out STD_LOGIC_VECTOR (3 downto 0);
      Cout : out STD_LOGIC);
end component Ripple_Adder;

signal ABc_in : STD_LOGIC_VECTOR (8 downto 0);
begin

reg : process(clk) is
begin
if(rising_edge(clk)) then
      if(reset = '1') then
            ABc_in <= (others => '0');
      else
            ABc_in <= ABc;
      end if;
 end if;
end process reg;

RA0 : Ripple_Adder
port map(
A           => ABc_in(7 downto 4),
B           => ABc_in(3 downto 0),
Cin   => ABc_in(8),
S           => C(3 downto 0),
Cout  => C(4)
```

```
);

end Behavioral;
```

Αθροιστής πρόβλεψης κρατουμένου (CLA – CarryLookaheadAdder)

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity full_adder is
  port (
    i_bit1  : in std_logic;
    i_bit2  : in std_logic;
    i_carry : in std_logic;
    --
    o_sum   : out std_logic;
    o_carry : out std_logic
    );
end full_adder;


architecture rtl of full_adder is

  signal w_WIRE_1 : std_logic;
  signal w_WIRE_2 : std_logic;
  signal w_WIRE_3 : std_logic;

begin

  w_WIRE_1 <= i_bit1 xor i_bit2;
  w_WIRE_2 <= w_WIRE_1 and i_carry;
  w_WIRE_3 <= i_bit1 and i_bit2;

  o_sum   <= w_WIRE_1 xor i_carry;
  o_carry <= w_WIRE_2 or w_WIRE_3;

end rtl;
-----------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity psg is
  port (
    A   : in std_logic;
    B   : in std_logic;
    C : in std_logic;

    O    : out std_logic
    );
end psg;

architecture Behavioral of psg is
  signal G : std_logic;
  signal P : std_logic;
begin
```

```
G <= A and B;
P <= A or B;
O <= G or (P and C);

end Behavioral;
--------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity carryLookAheadAdder is
port (
    A  : in std_logic_vector(7 downto 0);
    B  : in std_logic_vector(7 downto 0);
    --
    O   : out std_logic_vector(8 downto 0)
    );
end carryLookAheadAdder;

architecture Behavioral of carryLookAheadAdder is
 component full_adder is
    port (
      i_bit1  : in  std_logic;
      i_bit2  : in  std_logic;
      i_carry : in  std_logic;
      o_sum   : out std_logic;
      o_carry : out std_logic);
  end component full_adder;

  component psg is
  port (
    A  : in std_logic;
    B  : in std_logic;
    C : in std_logic;

    O   : out std_logic);
      end component;

       signal w_C : std_logic_vector(8 downto 0); -- Carry

  signal S   : std_logic_vector(7 downto 0);
begin
w_C(0) <= '0';
FULL_ADDER_BIT_0 : full_adder
    port map (
      i_bit1  => A(0),
      i_bit2  => B(0),
      i_carry => w_C(0),
      o_sum   => S(0),
      o_carry => open
      );
   PSG_BIT_1 : psg
  port map(
    A  => A(0),
    B  => B(0),
    C  => w_C(0),

    O  => w_C(1));
```

```
FULL_ADDER_BIT_1 : full_adder
  port map (
    i_bit1  => A(1),
    i_bit2  => B(1),
    i_carry => w_C(1),
    o_sum   => S(1),
    o_carry => open
    );

 PSG_BIT_2 : psg
port map(
  A  => A(1),
  B  => B(1),
  C  => w_C(1),

  O  => w_C(2));
FULL_ADDER_BIT_2 : full_adder
  port map (
    i_bit1  => A(2),
    i_bit2  => B(2),
    i_carry => w_C(2),
    o_sum   => S(2),
    o_carry => open
    );

 PSG_BIT_3 : psg
port map(
  A  => A(2),
  B  => B(2),
  C  => w_C(2),

  O  => w_C(3));
FULL_ADDER_BIT_3 : full_adder
  port map (
    i_bit1  => A(3),
    i_bit2  => B(3),
    i_carry => w_C(3),
    o_sum   => S(3),
    o_carry => open
    );

 PSG_BIT_4 : psg
port map(
  A  => A(3),
  B  => B(3),
  C  => w_C(3),

  O  => w_C(4));

FULL_ADDER_BIT_4 : full_adder
  port map (
    i_bit1  => A(4),
    i_bit2  => B(4),
    i_carry => w_C(4),
    o_sum   => S(4),
    o_carry => open
    );
```

```
 PSG_BIT_5 : psg
port map(
  A  => A(4),
  B  => B(4),
  C  => w_C(4),

  O  => w_C(5));

FULL_ADDER_BIT_5 : full_adder
  port map (
    i_bit1  => A(5),
    i_bit2  => B(5),
    i_carry => w_C(5),
    o_sum   => S(5),
    o_carry => open
    );

 PSG_BIT_6 : psg
port map(
  A  => A(5),
  B  => B(5),
  C  => w_C(5),

  O  => w_C(6));

FULL_ADDER_BIT_6 : full_adder
  port map (
    i_bit1  => A(6),
    i_bit2  => B(6),
    i_carry => w_C(6),
    o_sum   => S(6),
    o_carry => open
    );

 PSG_BIT_7 : psg
port map(
  A  => A(6),
  B  => B(6),
  C  => w_C(6),

  O  => w_C(7));
FULL_ADDER_BIT_7 : full_adder
  port map (
    i_bit1  => A(7),
    i_bit2  => B(7),
    i_carry => w_C(7),
    o_sum   => S(7),
    o_carry => open
    );
 PSG_BIT_8 : psg
port map(
  A  => A(7),
  B  => B(7),
  C  => w_C(7),

  O  => w_C(8));

     O <= w_C(4) & S;
```

```
end Behavioral;
```

Αθροιστής παράκαμψης κρατουμένου (CSK – CarrySkipAdder)

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

entity SkipLogic is
port (
    A       : in std_logic_vector(7 downto 0);
    B       : in std_logic_vector(7 downto 0);
       Ci   : in  std_logic;
       Co   : in std_logic;
       Cn   : out std_logic
    );
end SkipLogic;

architecture Behavioral of SkipLogic is
signal p0, p1, p2, p3,  p4,  p5,  p6,  p7, P, e : std_logic;
begin

  p0 <= A(0) or B(0);
  p1 <= A(1) or B(1);
  p2 <= A(2) or B(2);
  p3 <= A(3) or B(3);
  p4 <= A(4) or B(4);
  p5 <= A(5) or B(5);
  p6 <= A(6) or B(6);
  p7 <= A(7) or B(7);

  P   <= p0 and p1 and p2 and p3 and p4 and p5 and p6 and p7 ;
  e   <= P and Ci;
  Cn  <= e or Co;

end Behavioral;
---------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity ripplecadder is
port (
    A       : in std_logic_vector(7 downto 0);
    B       : in std_logic_vector(7 downto 0);
       Ci   : in  std_logic;
    --
    O     : out std_logic_vector(7 downto 0);
       Co   : out std_logic
    );
end ripplecadder;

architecture Behavioral of ripplecadder is
 component full_adder is
    port (
      i_bit1  : in  std_logic;
      i_bit2  : in  std_logic;
      i_carry : in  std_logic;
      o_sum   : out std_logic;
      o_carry : out std_logic);
```

```
  end component full_adder;
  signal carry_tmp   : std_logic_vector(6 downto 0);
begin
FULL_ADDER_BIT_0 : full_adder
    port map (
      i_bit1  => A(0),
      i_bit2  => B(0),
      i_carry => Ci,
      o_sum   => O(0),
      o_carry => carry_tmp(0)
      );
FULL_ADDER_BIT_1 : full_adder
    port map (
      i_bit1  => A(1),
      i_bit2  => B(1),
      i_carry => carry_tmp(0),
      o_sum   => O(1),
      o_carry => carry_tmp(1)
      );
FULL_ADDER_BIT_2 : full_adder
    port map (
      i_bit1  => A(2),
      i_bit2  => B(2),
      i_carry => carry_tmp(1),
      o_sum   => O(2),
      o_carry => carry_tmp(2)
      );
FULL_ADDER_BIT_3 : full_adder
    port map (
      i_bit1  => A(3),
      i_bit2  => B(3),
      i_carry => carry_tmp(2),
      o_sum   => O(3),
      o_carry => carry_tmp(3)
      );
FULL_ADDER_BIT_4 : full_adder
    port map (
      i_bit1  => A(4),
      i_bit2  => B(4),
      i_carry => carry_tmp(3),
      o_sum   => O(4),
      o_carry => carry_tmp(4)
      );
FULL_ADDER_BIT_5 : full_adder
    port map (
      i_bit1  => A(5),
      i_bit2  => B(5),
      i_carry => carry_tmp(4),
      o_sum   => O(5),
      o_carry => carry_tmp(5)
      );
FULL_ADDER_BIT_6 : full_adder
    port map (
      i_bit1  => A(6),
      i_bit2  => B(6),
      i_carry => carry_tmp(5),
      o_sum   => O(6),
      o_carry => carry_tmp(6)
```

```
      );
FULL_ADDER_BIT_7 : full_adder
    port map (
      i_bit1  => A(7),
      i_bit2  => B(7),
      i_carry => carry_tmp(6),
      o_sum   => O(7),
      o_carry => Co
      );
end Behavioral;
-----------------------------------------------------------------
---
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity full_adder is
  port (
    i_bit1  : in std_logic;
    i_bit2  : in std_logic;
    i_carry : in std_logic;
    --
    o_sum   : out std_logic;
    o_carry : out std_logic
    );
end full_adder;


architecture rtl of full_adder is

  signal w_WIRE_1 : std_logic;
  signal w_WIRE_2 : std_logic;
  signal w_WIRE_3 : std_logic;

begin

  w_WIRE_1 <= i_bit1 xor i_bit2;
  w_WIRE_2 <= w_WIRE_1 and i_carry;
  w_WIRE_3 <= i_bit1 and i_bit2;

  o_sum   <= w_WIRE_1 xor i_carry;
  o_carry <= w_WIRE_2 or w_WIRE_3;

end rtl;
-----------------------------------------------------------------
---
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity CarrySkipAdder16 is
    Port ( A : in  STD_LOGIC_VECTOR (15 downto 0);
           B : in  STD_LOGIC_VECTOR (15 downto 0);
           O : out  STD_LOGIC_VECTOR (15 downto 0);
           co : out  STD_LOGIC);
end CarrySkipAdder16;

  architecture Behavioral of CarrySkipAdder16 is
component ripplecadder
```

```
port (
    A       : in std_logic_vector(7 downto 0);
    B       : in std_logic_vector(7 downto 0);
       Ci   : in  std_logic;
    O     : out std_logic_vector(7 downto 0);
       Co   : out std_logic
    );
end component ripplecadder;

component SkipLogic
port (
    A       : in std_logic_vector(7 downto 0);
    B       : in std_logic_vector(7 downto 0);
       Ci   : in  std_logic;
       Co   : in std_logic;
    --
       Cn   : out std_logic
    );
end component SkipLogic;

signal couts     : std_logic_vector(1 downto 0);
signal e         : std_logic;

begin

RCA0 : ripplecadder
    port map (
      A  => A(7 downto 0),
      B  => B(7 downto 0),
      Ci => '0',
      O   => O(7 downto 0),
      Co => couts(0)
      );

RCA1 : ripplecadder
    port map (
      A  => A(15 downto 8),
      B  => B(15 downto 8),
      Ci => e,
      O   => O(15 downto 8),
      Co => couts(1)
      );

SKL0 : SkipLogic
    port map (
      A  => A(7 downto 0),
      B  => B(7 downto 0),
      Ci => '0',
      Co => couts(0),
      Cn => e
      );
SKL1 : SkipLogic
    port map (
      A  => A(15 downto 8),
      B  => B(15 downto 8),
      Ci => e,
      Co => couts(1),
      Cn => co
```

```
	);
end Behavioral;
```

Αθροιστής επιλογής κρατουμένου (CSL – CarrySelectAdder)

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

entity carry_select_adder is
	Port ( X : in STD_LOGIC_VECTOR (3 downto 0);
	Y : in STD_LOGIC_VECTOR (3 downto 0);
	CARRY_IN : in STD_LOGIC;
	SUM : out STD_LOGIC_VECTOR (3 downto 0);
	CARRY_OUT : out STD_LOGIC);
end carry_select_adder;

architecture Behavioral of carry_select_adder is

signal A,B,C0,C1: STD_LOGIC_VECTOR( 3 DOWNTO 0);
begin

	A(0)<= X(0) XOR Y(0) XOR '0' ;
	C0(0) <= (X(0) AND Y(0)) OR ('0' AND  X(0)) OR ('0' AND
Y(0)) ;
	A(1)<= X(1) XOR Y(1) XOR C0(0) ;
	C0(1) <= (X(1) AND Y(1)) OR (C0(0) AND  X(1)) OR (C0(0) AND
Y(1)) ;
	A(2)<= X(2) XOR Y(2) XOR C0(2) ;
	C0(2) <= (X(2) AND Y(2)) OR (C0(1) AND  X(2)) OR (C0(1) AND
Y(2)) ;
	A(3)<= X(3) XOR Y(3) XOR C0(2) ;
	C0(3) <= (X(3) AND Y(3)) OR (C0(2) AND  X(3)) OR (C0(2) AND
Y(3)) ;

	B(0)<= X(0) XOR Y(0) XOR '1' ;
	C1(0) <= (X(0) AND Y(0)) OR ('0' AND  X(0)) OR ('0' AND
Y(0)) ;
	B(1)<= X(1) XOR Y(1) XOR C1(0) ;
	C1(1) <= (X(1) AND Y(1)) OR (C1(0) AND  X(1)) OR (C1(0) AND
Y(1)) ;
	B(2)<= X(2) XOR Y(2) XOR C0(2) ;
	C1(2) <= (X(2) AND Y(2)) OR (C1(1) AND  X(2)) OR (C1(1) AND
Y(2)) ;
	B(3)<= X(3) XOR Y(3) XOR C0(2) ;
	C1(3) <= (X(3) AND Y(3)) OR (C1(2) AND  X(3)) OR (C1(2) AND
Y(3)) ;

process(C0(3),C1(3),CARRY_IN)
begin
	if CARRY_IN = '0' then
		CARRY_OUT <= C0(3);
	else
		CARRY_OUT <= C1(3);
	end if;
end process;

process(A(0),B(0),CARRY_IN)
```

```
begin
      if CARRY_IN = '0' then
            SUM(0) <= A(0);
      else
            SUM(0) <= B(0);
      end if;
end process;

process(A(1),B(1),CARRY_IN)
begin
      if CARRY_IN = '0' then
            SUM(1) <= A(1);
      else
            SUM(1) <= B(1);
      end if;
end process;

process(A(2),B(2),CARRY_IN)
begin
      if CARRY_IN = '0' then
            SUM(2) <= A(2);
      else
            SUM(2) <= B(2);
      end if;
end process;

process(A(3),B(3),CARRY_IN)
begin
      if CARRY_IN = '0' then
            SUM(3) <= A(3);
      else
            SUM(3) <= B(3);
      end if;
end process;

end Behavioral;
```

Αθροιστής με δέντρο Wallace

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

entity full_adder_H is
      Port ( A : in STD_LOGIC;
       B : in STD_LOGIC;
       Cin : in STD_LOGIC;
       S : out STD_LOGIC;
       Cout : out STD_LOGIC);
end full_adder_H;

architecture Behavioral of full_adder_H is

begin

 S <= A XOR B XOR Cin ;
 Cout <= (A AND B) OR (Cin AND A) OR (Cin AND B) ;
```

```
end Behavioral;
---------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity half_adder_H is
    Port ( A : in  STD_LOGIC;
           B : in  STD_LOGIC;
           C : out  STD_LOGIC;
           S : out  STD_LOGIC);
end half_adder_H;

architecture Behavioral of half_adder_H is

begin

S <= A xor B;
C <= A and B;

end Behavioral;
---------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity part_of_wallace_tree is
      Port (
      Ci : in STD_LOGIC_VECTOR(2 downto 0);
      X : in STD_LOGIC_VECTOR(5 downto 0);
      Co : out STD_LOGIC_VECTOR(2 downto 0);
      S0 : out STD_LOGIC;
      S1 : out STD_LOGIC);
end part_of_wallace_tree;

architecture Behavioral of part_of_wallace_tree is

component full_adder_H
      Port ( A : in STD_LOGIC;
      B : in STD_LOGIC;
      Cin : in STD_LOGIC;
      S : out STD_LOGIC;
      Cout : out STD_LOGIC);
end component;

signal W : STD_LOGIC_VECTOR(2 downto 0);

begin

FA1: full_adder_H port map(  X(0), X(1) , X(2), Co(0), W(0));
FA2: full_adder_H port map( X(3), X(4) , X(5),  Co(1), W(1));
FA3: full_adder_H port map( W(0), W(1) , Ci(0), Co(2), W(2) );
FA4: full_adder_H port map( W(2),  Ci(1), Ci(2) , S1   , S0  );

end Behavioral;
--------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Pprop_adder_H is
```

```
    Port ( X : in  STD_LOGIC_VECTOR (7 downto 0);
           Y : in  STD_LOGIC_VECTOR (7 downto 0);
           S : out  STD_LOGIC_VECTOR (8 downto 0));
end Pprop_adder_H;

architecture Behavioral of Pprop_adder_H is

component half_adder_H
    Port ( A : in  STD_LOGIC;
           B : in  STD_LOGIC;
           C : out  STD_LOGIC;
           S : out  STD_LOGIC);
end component;

component full_adder_H
      Port ( A : in STD_LOGIC;
       B : in STD_LOGIC;
       Cin : in STD_LOGIC;
       S : out STD_LOGIC;
       Cout : out STD_LOGIC);
end component;
signal W : STD_LOGIC_VECTOR(6 downto 0);
begin

HA : half_adder_H port map(X(0),Y(0),W(0),S(0));
FA0: full_adder_H port map(X(1) ,Y(1) ,W(0),S(1),W(1));
FA1: full_adder_H port map(X(2) ,Y(2) ,W(1),S(2),W(2));
FA2: full_adder_H port map(X(3) ,Y(3) ,W(2),S(3),W(3));
FA3: full_adder_H port map(X(4) ,Y(4) ,W(3),S(4),W(4));
FA4: full_adder_H port map(X(5) ,Y(5) ,W(4),S(5),W(5));
FA5: full_adder_H port map(X(6) ,Y(6) ,W(5),S(6),W(6));
FA6: full_adder_H port map(X(7) ,Y(7) ,W(6),S(7),S(8));

end Behavioral;
------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity wallace_tree is
    Port ( S : out  STD_LOGIC_VECTOR (10 downto 0);
           X : in  STD_LOGIC_VECTOR (47 downto 0));
end wallace_tree;

architecture Behavioral of wallace_tree is

component Pprop_adder_H
    Port ( X : in  STD_LOGIC_VECTOR (7 downto 0);
           Y : in  STD_LOGIC_VECTOR (7 downto 0);
           S : out  STD_LOGIC_VECTOR (8 downto 0));
end component;

component part_of_wallace_tree
      Port (
            Ci : in STD_LOGIC_VECTOR(2 downto 0);
            X : in STD_LOGIC_VECTOR(5 downto 0);
            Co : out STD_LOGIC_VECTOR(2 downto 0);
            S0 : out STD_LOGIC;
            S1 : out STD_LOGIC);
```

```
end component;

component half_adder_H
    Port ( A : in  STD_LOGIC;
           B : in  STD_LOGIC;
           C : out  STD_LOGIC;
           S : out  STD_LOGIC);
end component;

component full_adder_H
      Port ( A : in STD_LOGIC;
            B : in STD_LOGIC;
            Cin : in STD_LOGIC;
            S : out STD_LOGIC;
            Cout : out STD_LOGIC);
end component;

signal W3 : STD_LOGIC_VECTOR(8 downto 0);
signal W2 : STD_LOGIC_VECTOR(8 downto 0);

signal WC0 : STD_LOGIC_VECTOR(2 downto 0);
signal WC1 : STD_LOGIC_VECTOR(2 downto 0);
signal WC2 : STD_LOGIC_VECTOR(2 downto 0);
signal WC3 : STD_LOGIC_VECTOR(2 downto 0);
signal WC4 : STD_LOGIC_VECTOR(2 downto 0);
signal WC5 : STD_LOGIC_VECTOR(2 downto 0);
signal WC6 : STD_LOGIC_VECTOR(2 downto 0);
signal WC7 : STD_LOGIC_VECTOR(2 downto 0);
signal X1,Y1 : STD_LOGIC_VECTOR(7 downto 0);
signal R1 : STD_LOGIC_VECTOR(8 downto 0);
begin

--FA1: full_adder_H port map( Co(0), W(0), X(0), X(1) , X(2));
PART0 : part_of_wallace_tree port map("000"   , X(5 downto 0),
WC0, W3(0), W2(0));
PART1 : part_of_wallace_tree port map(WC0 , X(11 downto 6), WC1,
W3(1), W2(1));
PART2 : part_of_wallace_tree port map(WC1 , X(17 downto 12), WC2,
W3(2), W2(2));
PART3 : part_of_wallace_tree port map(WC2 , X(23 downto 18), WC3,
W3(3), W2(3));
PART4 : part_of_wallace_tree port map(WC3 , X(29 downto 24), WC4,
W3(4), W2(4));
PART5 : part_of_wallace_tree port map(WC4 , X(35 downto 30), WC5,
W3(5), W2(5));
PART6 : part_of_wallace_tree port map(WC5 , X(41 downto 36), WC6,
W3(6), W2(6));
PART7 : part_of_wallace_tree port map(WC6 , X(47 downto 42), WC7,
W3(7), W2(7));


FA   : full_adder_H port map(WC7(2), WC7(1), WC7(0), W3(8),
W2(8));
X1 <= W2(7 downto 0);
Y1 <= W3(8 downto 1);
PROP : Pprop_adder_H port map(X1,Y1,R1);
HA : half_adder_H port map (W2(8),R1(8),S(10),S(9));
      S(8) <= R1(8);
```

```
      S(7) <= R1(7);
      S(6) <= R1(6);
      S(5) <= R1(5);
      S(4) <= R1(4);
      S(3) <= R1(3);
      S(2) <= R1(2);
      S(1) <= R1(1);
      S(0) <= W3(0);

end Behavioral;
```

Πολλαπλασιαστής Booth των n·mbits βάσης-2

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY booth_multi_radix IS
  GENERIC(a: NATURAL:= 8; b: NATURAL:= 4);
PORT(
  c, d: IN STD_LOGIC_VECTOR(a DOWNTO 0);
  cin: IN STD_LOGIC_VECTOR(b DOWNTO 0);
  cout: OUT STD_LOGIC_VECTOR(a+b+1 DOWNTO 0)
);
END booth_multi_radix;

ARCHITECTURE circuit OF booth_multi_radix IS
  TYPE m IS ARRAY (0 TO b) OF STD_LOGIC_VECTOR(a+1 DOWNTO 0);
  SIGNAL f: m;
BEGIN
  f(0) <= ((d(a)&d) - (c(a)&c)) when cin(0) = '1' ELSE d(a)&d;
  cout(0) <= f(0)(0);
  mnit: FOR i IN 1 TO b GENERATE
    f(i) <= ((f(i-1)(a+1)&f(i-1)(a+1 DOWNTO 1)) - (c(a)&c)) WHEN
(cin(i-1) = '0' AND cin(i) = '1')
    ELSE ((f(i-1)(a+1)&f(i-1)(a+1 DOWNTO 1)) + (c(a)&c)) WHEN
(cin(i-1) = '1' AND cin(i) = '0')
    ELSE f(i-1)(a+1)&f(i-1)(a+1 DOWNTO 1);
    cout(i) <= f(i)(0);
  END GENERATE;
  cout(a+b+1 DOWNTO b+1) <= f(b)(a+1 DOWNTO 1);

END circuit;
```

Πολλαπλασιαστής Booth των n·mbits βάσης-4

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;
use ieee.numeric_std.all;

entity booth_multi_radix4 is
    Port ( sin : in std_logic_vector(7 downto 0);
           par : in std_logic_vector(2 downto 0);
```

```
            sout : out std_logic_vector(15 downto 0));
end booth_multi_radix4;

architecture Behavioral of booth_multi_radix4 is
      function booth_multi(par1: std_logic_vector(2 downto
0);val:std_logic_vector(7 downto 0))
            return std_logic_vector is
                  variable y,y1,y2: std_logic_vector(8 downto 0);
                  variable sign: std_logic;
      begin
            case par1 is
             when "001"|"010"  =>
                  if val <0 then
                  y:='1'& val;
             else
                  y:='0'&val;
             end if;
            when "011" =>
              if val<0 then
                  y1:='1'&val;
                  y:=y1(7 downto 0)&'0';
              else
              y:='0'&val(6 downto 0)&'0';
              end if  ;
            when "100"  =>
             if val<0 then
              y1:='1'&val;
              y2:=(not y1)+"000000001";
              y:=(y2(7 downto 0)&'0');
             else
              y1:='0'&val;
              y2:=(not y1)+"000000001";
              y:=(y2(7 downto 0)&'0');
             end if;
            when "101"|"110" =>
                  if val < 0 then
                   y1:='1'&val;
              y:=not(y1)+"000000001";
                  else
                   y1:='0'&val;
              y:=(not y1)+"000000001";
                  end if;
            when others =>
              y:="000000000";
        end case;
        return y;
      end booth_multi;
        signal s1: std_logic_vector(8 downto 0);
        signal s2: std_logic;
begin
      s1<=booth_multi(par,sin);
   sout<=sxt(s1,16);
end Behavioral;
```

Πολλαπλασιασμός αριθμών με δέντρο Wallace και Dadda

```
library IEEE;
```

```
use IEEE.STD_LOGIC_1164.all;

package dadda_utils is
      function andbit(bit2: std_logic; t: std_logic_vector) return
std_logic_vector;
      function s2bit(bit1: std_logic; bit2: std_logic) return
std_logic;
      function s3bit(bit1: std_logic; bit2: std_logic; bit3:
std_logic) return std_logic;
      function c2bit(bit1: std_logic; bit2: std_logic) return
std_logic;
      function c3bit(bit1: std_logic; bit2: std_logic; bit3:
std_logic) return std_logic;
end;

package body dadda_utils is

function andbit(bit2: std_logic; t: std_logic_vector) return
std_logic_vector is
      variable rout: std_logic_vector(7 downto 0);
      begin
            for i in 0 to 7 loop
                  rout(i) := t(i) and bit2;
            end loop;
            return rout;
end function;
function s2bit(bit1: std_logic; bit2: std_logic) return std_logic
is
      begin
            return bit1 xor bit2;
end;
function s3bit(bit1: std_logic; bit2: std_logic; bit3: std_logic)
return std_logic is
      begin
            return bit1 xor bit2 xor bit3;
end;
function c2bit(bit1: std_logic; bit2: std_logic) return std_logic
is
      begin
            return bit1 and bit2;
end;
function c3bit(bit1: std_logic; bit2: std_logic; bit3: std_logic)
return std_logic is
      begin
            return (bit1 and bit2) or (bit1 and bit3) or (bit2 and
bit3);
end;
end package body;
-----------------------------------------------------------------
---------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.dadda_utils.all;
use ieee.numeric_std.all;

entity dadda_multi is
  port(
    xin: in std_logic_vector(7 downto 0);
```

```
    yin: in std_logic_vector(7 downto 0);
    zout: out std_logic_vector(15 downto 0)
  );
end dadda_multi;

architecture dadda_multi_arch of dadda_multi is

signal l1: std_logic_vector(7 downto 0);
signal l2: std_logic_vector(7 downto 0);
signal l3: std_logic_vector(7 downto 0);
signal l4: std_logic_vector(7 downto 0);
signal l5: std_logic_vector(7 downto 0);
signal l6: std_logic_vector(7 downto 0);
signal l7: std_logic_vector(7 downto 0);
signal l8: std_logic_vector(7 downto 0);

signal c: std_logic_vector(30 downto 1);

signal s1_1: std_logic_vector(14 downto 0);
signal s1_2: std_logic_vector(13 downto 1);
signal s1_3: std_logic_vector(12 downto 2);
signal s1_4: std_logic_vector(11 downto 3);
signal s1_5: std_logic_vector(10 downto 4);
signal s1_6: std_logic_vector(10 downto 5);

signal s2_1: std_logic_vector(14 downto 0);
signal s2_2: std_logic_vector(13 downto 1);
signal s2_3: std_logic_vector(12 downto 2);
signal s2_4: std_logic_vector(12 downto 3);

signal s3_1: std_logic_vector(14 downto 0);
signal s3_2: std_logic_vector(13 downto 1);
signal s3_3: std_logic_vector(13 downto 2);

signal s4_1: std_logic_vector(15 downto 0);
signal s4_2: std_logic_vector(15 downto 0);
begin

      l1 <= andbit(yin(0), xin); l2 <= andbit(yin(1), xin);
      l3 <= andbit(yin(2), xin); l4 <= andbit(yin(3), xin);
      l5 <= andbit(yin(4), xin); l6 <= andbit(yin(5), xin);
      l7 <= andbit(yin(6), xin); l8 <= andbit(yin(7), xin);

      s1_1(0)  <= l1(0);
      s1_1(1)  <= l1(1);
      s1_1(2)  <= l1(2);
      s1_1(3)  <= l1(3);
      s1_1(4)  <= l1(4);
      s1_1(5)  <= l1(5);
      s1_1(6)  <= s2bit(l1(6), l2(5));
      s1_1(7)  <= s3bit(l1(7), l2(6), l3(5));
      s1_1(8)  <= s3bit(l2(7), l3(6), l4(5));
      s1_1(9)  <= s3bit(l3(7), l4(6), l5(5));
      s1_1(10) <= l4(7);
      s1_1(11) <= l5(7);
      s1_1(12) <= l6(7);
      s1_1(13) <= l7(7);
      s1_1(14) <= l8(7);
```

```
s1_2(1)  <= l2(0);
s1_2(2)  <= l2(1);
s1_2(3)  <= l2(2);
s1_2(4)  <= l2(3);
s1_2(5)  <= l2(4);
s1_2(6)  <= l3(4);
s1_2(7)  <= s2bit(l4(4), l5(3));
s1_2(8)  <= s2bit(l5(4), l6(3));
s1_2(9)  <= l6(4);
s1_2(10) <= l5(6);
s1_2(11) <= l6(6);
s1_2(12) <= l7(6);
s1_2(13) <= l8(6);

s1_3(2)  <= l3(0);
s1_3(3)  <= l3(1);
s1_3(4)  <= l3(2);
s1_3(5)  <= l3(3);
s1_3(6)  <= l4(3);
s1_3(7)  <= l6(2);
s1_3(8)  <= l7(2);
s1_3(9)  <= l7(3);
s1_3(10) <= l6(5);
s1_3(11) <= l7(5);
s1_3(12) <= l8(5);

s1_4(3)  <= l4(0);
s1_4(4)  <= l4(1);
s1_4(5)  <= l4(2);
s1_4(6)  <= l5(2);
s1_4(7)  <= l7(1);
s1_4(8)  <= l8(1);
s1_4(9)  <= l8(2);
s1_4(10) <= l7(4);
s1_4(11) <= l8(4);

s1_5(4)  <= l5(0);
s1_5(5)  <= l5(1);
s1_5(6)  <= l6(1);
s1_5(7)  <= l8(0);
s1_5(8)  <= s3bit(l1(7), l2(6), l3(5));
s1_5(9)  <= s3bit(l2(7), l3(6), l4(5));
s1_5(10) <= l8(3);

s1_6(5)  <= l6(0);
s1_6(6)  <= l7(0);
s1_6(7)  <= c2bit(l1(6), l2(5));
s1_6(8)  <= c2bit(l4(4), l5(3));
s1_6(9)  <= c2bit(l5(4), l6(3));
s1_6(10) <= s3bit(l3(7), l4(6), l5(5));

s2_1(0)  <= s1_1(0);
s2_1(1)  <= s1_1(1);
s2_1(2)  <= s1_1(2);
s2_1(3)  <= s1_1(3);
s2_1(4)  <= s2bit(s1_1(4), s1_2(4));
s2_1(5)  <= s3bit(s1_1(5), s1_2(5), s1_3(5));
```

```
s2_1(6)  <= s3bit(s1_1(6), s1_2(6), s1_3(6));
s2_1(7)  <= s3bit(s1_1(7), s1_2(7), s1_3(7));
s2_1(8)  <= s3bit(s1_1(8), s1_2(8), s1_3(8));
s2_1(9)  <= s3bit(s1_1(9), s1_2(9), s1_3(9));
s2_1(10) <= s3bit(s1_1(10), s1_2(10), s1_3(10));
s2_1(11) <= s3bit(s1_1(11), s1_2(11), s1_3(11));
s2_1(12) <= s1_1(12);
s2_1(13) <= s1_1(13);
s2_1(14) <= s1_1(14);

s2_2(1)  <= s1_2(1);
s2_2(2)  <= s1_2(2);
s2_2(3)  <= s1_2(3);
s2_2(4)  <= s1_3(4);
s2_2(5)  <= s2bit(s1_4(5), s1_5(5));
s2_2(6)  <= s3bit(s1_4(6), s1_5(6), s1_6(6));
s2_2(7)  <= s3bit(s1_4(7), s1_5(7), s1_6(7));
s2_2(8)  <= s3bit(s1_4(8), s1_5(8), s1_6(8));
s2_2(9)  <= s3bit(s1_4(9), s1_5(9), s1_6(9));
s2_2(10) <= s3bit(s1_4(10), s1_5(10), s1_6(10));
s2_2(11) <= s1_4(11);
s2_2(12) <= s1_2(12);
s2_2(13) <= s1_2(13);

s2_3(2)  <= s1_3(2);
s2_3(3)  <= s1_3(3);
s2_3(4)  <= s1_4(4);
s2_3(5)  <= s1_6(5);
s2_3(6)  <= s3bit(s1_1(5), s1_2(5), s1_3(5));
s2_3(7)  <= s3bit(s1_1(6), s1_2(6), s1_3(6));
s2_3(8)  <= s3bit(s1_1(7), s1_2(7), s1_3(7));
s2_3(9)  <= s3bit(s1_1(8), s1_2(8), s1_3(8));
s2_3(10) <= s3bit(s1_1(9), s1_2(9), s1_3(9));
s2_3(11) <= s3bit(s1_1(10), s1_2(10), s1_3(10));
s2_3(12) <= s1_3(12);

s2_4(3)  <= s1_4(3);
s2_4(4)  <= s1_5(4);
s2_4(5)  <= c2bit(s1_1(4), s1_2(4));
s2_4(6)  <= c2bit(s1_4(5), s1_5(5));
s2_4(7)  <= s3bit(s1_4(6), s1_5(6), s1_6(6));
s2_4(8)  <= s3bit(s1_4(7), s1_5(7), s1_6(7));
s2_4(9)  <= s3bit(s1_4(8), s1_5(8), s1_6(8));
s2_4(10) <= s3bit(s1_4(9), s1_5(9), s1_6(9));
s2_4(11) <= s3bit(s1_4(10), s1_5(10), s1_6(10));
s2_4(12) <= s3bit(s1_1(11), s1_2(11), s1_3(11));

s3_1(0)  <= s2_1(0);
s3_1(1)  <= s2_1(1);
s3_1(2)  <= s2_1(2);
s3_1(3)  <= s2bit(s2_1(3), s2_2(3));
s3_1(4)  <= s3bit(s2_1(4), s2_2(4), s2_3(4));
s3_1(5)  <= s3bit(s2_1(5), s2_2(5), s2_3(5));
s3_1(6)  <= s3bit(s2_1(6), s2_2(6), s2_3(6));
s3_1(7)  <= s3bit(s2_1(7), s2_2(7), s2_3(7));
s3_1(8)  <= s3bit(s2_1(8), s2_2(8), s2_3(8));
s3_1(9)  <= s3bit(s2_1(9), s2_2(9), s2_3(9));
s3_1(10) <= s3bit(s2_1(10), s2_2(10), s2_3(10));
```

```
s3_1(11) <= s3bit(s2_1(11), s2_2(11), s2_3(11));
s3_1(12) <= s3bit(s2_1(12), s2_2(12), s2_3(12));
s3_1(13) <= s2_1(13);
s3_1(14) <= s2_1(14);

s3_2(1)  <= s2_2(1);
s3_2(2)  <= s2_2(2);
s3_2(3)  <= s2_3(3);
s3_2(4)  <= s2_4(4);
s3_2(5)  <= s2_4(5);
s3_2(6)  <= s2_4(6);
s3_2(7)  <= s2_4(7);
s3_2(8)  <= s2_4(8);
s3_2(9)  <= s2_4(9);
s3_2(10) <= s2_4(10);
s3_2(11) <= s2_4(11);
s3_2(12) <= s2_4(12);
s3_2(13) <= s2_2(13);

s3_3(2)  <= s2_3(2);
s3_3(3)  <= s2_4(3);
s3_3(4)  <= c2bit(s2_1(3), s2_2(3));
s3_3(5)  <= s3bit(s2_1(4), s2_2(4), s2_3(4));
s3_3(6)  <= s3bit(s2_1(5), s2_2(5), s2_3(5));
s3_3(7)  <= s3bit(s2_1(6), s2_2(6), s2_3(6));
s3_3(8)  <= s3bit(s2_1(7), s2_2(7), s2_3(7));
s3_3(9)  <= s3bit(s2_1(8), s2_2(8), s2_3(8));
s3_3(10) <= s3bit(s2_1(9), s2_2(9), s2_3(9));
s3_3(11) <= s3bit(s2_1(10), s2_2(10), s2_3(10));
s3_3(12) <= s3bit(s2_1(11), s2_2(11), s2_3(11));
s3_3(13) <= s3bit(s2_1(12), s2_2(12), s2_3(12));

s4_1(0)  <= s3_1(0);
s4_1(1)  <= s3_1(1);
s4_1(2)  <= s2bit(s3_1(2), s3_2(2));
s4_1(3)  <= s3bit(s3_1(3), s3_2(3), s3_3(3));
s4_1(4)  <= s3bit(s3_1(4), s3_2(4), s3_3(4));
s4_1(5)  <= s3bit(s3_1(5), s3_2(5), s3_3(5));
s4_1(6)  <= s3bit(s3_1(6), s3_2(6), s3_3(6));
s4_1(7)  <= s3bit(s3_1(7), s3_2(7), s3_3(7));
s4_1(8)  <= s3bit(s3_1(8), s3_2(8), s3_3(8));
s4_1(9)  <= s3bit(s3_1(9), s3_2(9), s3_3(9));
s4_1(10) <= s3bit(s3_1(10), s3_2(10), s3_3(10));
s4_1(11) <= s3bit(s3_1(11), s3_2(11), s3_3(11));
s4_1(12) <= s3bit(s3_1(12), s3_2(12), s3_3(12));
s4_1(13) <= s3bit(s3_1(13), s3_2(13), s3_3(13));
s4_1(14) <= s3_1(14);
s4_1(15) <= '0';

s4_2(0)  <= '0';
s4_2(1)  <= s3_2(1);
s4_2(2)  <= s3_3(2);
s4_2(3)  <= c2bit(s3_1(2), s3_2(2));
s4_2(4)  <= s3bit(s3_1(3), s3_2(3), s3_3(3));
s4_2(5)  <= s3bit(s3_1(4), s3_2(4), s3_3(4));
s4_2(6)  <= s3bit(s3_1(5), s3_2(5), s3_3(5));
s4_2(7)  <= s3bit(s3_1(6), s3_2(6), s3_3(6));
s4_2(8)  <= s3bit(s3_1(7), s3_2(7), s3_3(7));
```

```
      s4_2(9)  <= s3bit(s3_1(8), s3_2(8), s3_3(8));
      s4_2(10) <= s3bit(s3_1(9), s3_2(9), s3_3(9));
      s4_2(11) <= s3bit(s3_1(10), s3_2(10), s3_3(10));
      s4_2(12) <= s3bit(s3_1(11), s3_2(11), s3_3(11));
      s4_2(13) <= s3bit(s3_1(12), s3_2(12), s3_3(12));
      s4_2(14) <= s3bit(s3_1(13), s3_2(13), s3_3(13));
      s4_2(15) <= '0';

      zout <= std_logic_vector(unsigned(s4_1) + unsigned(s4_2));
end architecture;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity full_adder is
Port ( bit1 : in  STD_LOGIC;
       bit2 : in  STD_LOGIC;
       cin  : in  STD_LOGIC;
       sout : out  STD_LOGIC;
       cout : out  STD_LOGIC);
end full_adder;

architecture Behavioral of full_adder is
begin
      sout <= (bit1 xor bit2 xor cin);
      cout <= (bit1 and bit2) xor (cin and (bit1 xor bit2));
end Behavioral;
----------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity half_adder is
Port ( bit1 : in  STD_LOGIC;
       bit2 : in  STD_LOGIC;
       sout : out  STD_LOGIC;
       cout : out  STD_LOGIC);
end half_adder;

architecture Behavioral of half_adder is
begin
      sout <= bit1 xor bit2;
      cout <= bit1 and bit2;
end Behavioral;
-------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity wallace4 is
Port ( x : in  STD_LOGIC_VECTOR (3 downto 0);
           y : in  STD_LOGIC_VECTOR (3 downto 0);
           Product : out  STD_LOGIC_VECTOR (7 downto 0));
end wallace4;

architecture Behavioral of wallace4 is
component full_adder is
    Port ( bit1 : in  STD_LOGIC;
       bit2 : in  STD_LOGIC;
       cin  : in  STD_LOGIC;
```

```
       sout : out  STD_LOGIC;
       cout : out  STD_LOGIC);
end component;

component half_adder is
    Port ( bit1 : in  STD_LOGIC;
       bit2 : in  STD_LOGIC;
       sout : out  STD_LOGIC;
       cout : out  STD_LOGIC);
end component;

signal
s11,s12,s13,s14,s15,s22,s23,s24,s25,s26,s32,s34,s35,s36,s37 :
std_logic;
signal
c11,c12,c13,c14,c15,c22,c23,c24,c25,c26,c32,c34,c35,c36,c37 :
std_logic;
signal pr0,pr1,pr2,pr3 : std_logic_vector(3 downto 0);

begin

process(x,y)
begin
    for i in 0 to 3 loop
        pr0(i) <= x(i) and y(0);
        pr1(i) <= x(i) and y(1);
        pr2(i) <= x(i) and y(2);
        pr3(i) <= x(i) and y(3);
    end loop;
end process;

Product(0) <= pr0(0);
Product(1) <= s11;
Product(2) <= s22;
Product(3) <= s32;
Product(4) <= s34;
Product(5) <= s35;
Product(6) <= s36;
Product(7) <= s37;

--first stage
ha11 : half_adder port map(pr0(1),pr1(0),s11,c11);
fa12 : full_adder port map(pr0(2),pr1(1),pr2(0),s12,c12);
fa13 : full_adder port map(pr0(3),pr1(2),pr2(1),s13,c13);
fa14 : full_adder port map(pr1(3),pr2(2),pr3(1),s14,c14);
ha15 : half_adder port map(pr2(3),pr3(2),s15,c15);

--second stage
ha22 : half_adder port map(c11,s12,s22,c22);
fa23 : full_adder port map(pr3(0),c12,s13,s23,c23);
fa24 : full_adder port map(c13,c23,s14,s24,c24);
fa25 : full_adder port map(c14,c24,s15,s25,c25);
fa26 : full_adder port map(c15,c25,pr3(3),s26,c26);

--third stage
ha32 : half_adder port map(c22,s23,s32,c32);
ha34 : half_adder port map(c32,s24,s34,c34);
ha35 : half_adder port map(c34,s25,s35,c35);
```

```
ha36 : half_adder port map(c35,s26,s36,c36);
ha37 : half_adder port map(c36,c26,s37,c37);

end Behavioral;
```

Πολλαπλασιαστής με διάταξη πίνακα (ArrayMultiplier)

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

entity array_multiplier is
      Port ( cin : in std_logic_vector(3 downto 0);
            cout : in std_logic_vector(3 downto 0);
            prod : out std_logic_vector(7 downto 0));
end array_multiplier;

architecture Behavioral of array_multiplier is

      constant n:integer :=4;
      subtype stype is std_logic_vector(n-1 downto 0);
      type krat is array(0 to n) of stype;
      signal a,b,kr:krat;

begin

      pg:for x in 0 to n-1 generate
            pg1:for y in 0 to n-1 generate
                  a(x)(y)<=cin(y) and cout(x);
            end generate;
            b(0)(x)<='0';
      end generate;

      kr(0)<=a(0);
      prod(0)<=a(0)(0);
      addr:for x in 1 to n-1 generate
            addc:for y in 0 to n-2 generate
                  kr(x)(y)<=a(x)(y) xor b(x-1)(y) xor kr(x-
1)(y+1);
                  b(x)(y)<=(a(x)(y) and b(x-1)(y)) or
                                   (a(x)(y) and kr(x-1)(y+1)) or
                        (b(x-1)(y)and kr(x-1)(y+1));
            end generate;
            prod(x)<=kr(x)(0);
            kr(x)(n-1)<=a(x)(n-1);
      end generate;

      b(n)(0)<='0';

      addlast:for y in 1 to n-1 generate
            kr(n)(y)<=b(n)(y-1) xor b(n-1)(y-1) xor kr(n-1)(y);
            b(n)(y)<=(b(n)(y-1) and b(n-1)(y-1)) or
                              (b(n)(y-1) and kr(n-1)(y)) or
                  (b(n-1)(y-1)and kr(n-1)(y));
      end generate;

      prod(2*n-1)<=b(n)(n-1);
```

```
      prod(2*n-2 downto n)<=kr(n)(n-1 downto 1);

end Behavioral;
```

Modular πολλαπλασιαστής με βάση τον αλγόριθμο Montgomery

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
PACKAGE mod_m IS

  CONSTANT nbits: NATURAL := 8;
  CONSTANT vec: STD_LOGIC_VECTOR(7 DOWNTO 0) := "11101111";
  CONSTANT minus_vec: STD_LOGIC_VECTOR(8 DOWNTO 0):= "100010001";

END mod_m;
----------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
USE work.mod_m.ALL;

ENTITY Montgomery_multiplier IS
PORT(
  ain, bin: IN STD_LOGIC_VECTOR(nbits-1 DOWNTO 0);
  clk, reset, start: STD_LOGIC;
  pout: OUT STD_LOGIC_VECTOR(nbits-1 DOWNTO 0);
  oksig: OUT STD_LOGIC
);
END Montgomery_multiplier;

ARCHITECTURE circuit OF Montgomery_multiplier IS

  SIGNAL p, next_p, p_minus_m: STD_LOGIC_VECTOR(nbits DOWNTO 0);
  SIGNAL two_p: STD_LOGIC_VECTOR(nbits+1 DOWNTO 0);
  SIGNAL vector_xi, vector_q, int_x: STD_LOGIC_VECTOR(nbits-1
DOWNTO 0);
  SIGNAL xi, q: STD_LOGIC;

  SIGNAL load, update: STD_LOGIC;
  SUBTYPE index IS NATURAL RANGE 0 TO nbits-1;
  SIGNAL count: index;
  TYPE state IS RANGE 0 TO 4;
  SIGNAL current_state: state;
BEGIN
  q <= p(0) XOR (xi AND bin(0));
  vector_xi <= (OTHERS => xi); vector_q <= (OTHERS => q);
  two_p <= ('0'&p) + (vector_xi AND bin) +  (vector_q AND vec);
  next_p <= two_p(nbits+1 DOWNTO 1);
  p_minus_m <= p + minus_vec;
  WITH p_minus_m(nbits) SELECT pout <= p_minus_m(nbits-1 DOWNTO
0) WHEN '0', p(nbits-1 DOWNTO 0) WHEN OTHERS;

  register_p: PROCESS(clk)
  BEGIN
```

```
    IF clk'EVENT AND CLK = '1' THEN
      IF load = '1' THEN p <= (OTHERS => '0');
      ELSIF update = '1' THEN p <= next_p;
      END IF;
    END IF;
  END PROCESS;

  shift_register: PROCESS(clk)
  BEGIN
    IF clk'EVENT AND CLK = '1' THEN
      IF load = '1' THEN int_x <= ain;
      ELSIF update = '1' THEN int_x <= '0'&int_x(nbits-1 DOWNTO
1);
      END IF;
    END IF;
  END PROCESS;
  xi <= int_x(0);

  counter: PROCESS(clk)
  BEGIN
    IF clk'EVENT and clk = '1' THEN
      IF load = '1' THEN count <= 0;
      ELSIF update = '1' THEN count <= (count+1) MOD nbits;
      END IF;
    END IF;
  END PROCESS;

  next_state: PROCESS(clk)
  BEGIN
    IF reset = '1' THEN current_state <= 0;
    ELSIF clk'EVENT AND clk = '1' THEN
      CASE current_state IS
        WHEN 0 => IF start = '0' THEN current_state <= 1; END IF;
        WHEN 1 => IF start = '1' THEN current_state <= 2; END IF;
        WHEN 2 => current_state <= 3;
        WHEN 3 => IF count = nbits-1 THEN current_state <= 4; END
IF;
        WHEN 4 => current_state <= 0;

      END CASE;
    END IF;
  END PROCESS;

  output_function: PROCESS(clk, current_state)
  BEGIN
    CASE current_state IS
      WHEN 0 TO 1 => load <= '0'; update <= '0'; oksig <= '1';
      WHEN 2 => load <= '1'; update <= '0'; oksig <= '0';
      WHEN 3 => load <= '0'; update <= '1'; oksig <= '0';

      WHEN 4 => load <= '0'; update <= '0'; oksig <= '0';

    END CASE;
  END PROCESS;
END circuit;
```

Ripple Adder με τεχνική Pipeline

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

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity RIP_Ripple_Adder is
       Port (  clk :    in    STD_LOGIC;
                        reset :     in    STD_LOGIC;
                        A :         in    STD_LOGIC_VECTOR (3
downto 0);
                        B :         in    STD_LOGIC_VECTOR (3
downto 0);
                        C :         in    STD_LOGIC_VECTOR (3
downto 0);
                        D :         in    STD_LOGIC_VECTOR (3
downto 0);
                        Ci:         in    STD_LOGIC;
                        Co:         out   STD_LOGIC;
                        S :         out   STD_LOGIC_VECTOR (3
downto 0));
end RIP_Ripple_Adder;

architecture Behavioral of RIP_Ripple_Adder is

component Ripple_Adder is
      Port ( A : in STD_LOGIC_VECTOR (3 downto 0);
      B : in STD_LOGIC_VECTOR (3 downto 0);
      Cin : in STD_LOGIC;
      S : out STD_LOGIC_VECTOR (3 downto 0);
      Cout : out STD_LOGIC);
end component Ripple_Adder;


signal A_in0 :  STD_LOGIC_VECTOR (3 downto 0);
signal A_in1 :  STD_LOGIC_VECTOR (3 downto 0);
signal A_in2 :  STD_LOGIC_VECTOR (3 downto 0);
signal C0 : STD_LOGIC;
signal C1 : STD_LOGIC;
signal C2 : STD_LOGIC;

signal A_in0_reg :  STD_LOGIC_VECTOR (3 downto 0);
signal A_in1_reg :  STD_LOGIC_VECTOR (3 downto 0);
signal A_in2_reg :  STD_LOGIC_VECTOR (3 downto 0);
signal C0_reg    :  STD_LOGIC;
signal C1_reg    :  STD_LOGIC;
signal C2_reg    :  STD_LOGIC;

signal C_reg :  STD_LOGIC_VECTOR (3 downto 0);
signal D_reg0 :  STD_LOGIC_VECTOR (3 downto 0);
signal D_reg1 :  STD_LOGIC_VECTOR (3 downto 0);
```

```
begin



RA0 : Ripple_Adder
port map(
A           => A,
B           => B,
Cin   => Ci,
S           => A_in0,
Cout  => C0
);

RA1 : Ripple_Adder
port map(
A           => A_in0_reg,
B           => C_reg,
Cin   => C0_reg,
S           => A_in1,
Cout  => C1
);

RA2 : Ripple_Adder
port map(
A           => A_in1_reg,
B           => D_reg1,
Cin   => C1_reg,
S           => S,
Cout  => Co
);

reg0 : process(clk,reset) is
begin
if(reset = '1') then
            A_in0_reg <= (others => '0');
            A_in1_reg <= (others => '0');
            C0_reg      <= '0';
            C1_reg      <= '0';
            C_reg <= (others => '0');
            D_reg0 <= (others => '0');
            D_reg1 <= (others => '0');
else
      if(rising_edge(clk)) then
            A_in0_reg <= A_in0;
            A_in1_reg <= A_in1;
            A_in2_reg <= A_in2;
            C0_reg      <= C0;
            C1_reg      <= C1;
            C_reg <= C;
            D_reg0 <= D;
            D_reg1 <= D_reg0;
      end if;
end if;
end process reg0;

end Behavioral;
```

Array Multiplier με τεχνική Pipeline

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

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity RIP_array_multiplier is
    Port ( clk : in  STD_LOGIC;
           reset : in  STD_LOGIC;
           cin : in  STD_LOGIC_VECTOR (3 downto 0);
           cout1 : in  STD_LOGIC_VECTOR (3 downto 0);
           cout2 : in  STD_LOGIC_VECTOR (3 downto 0);
           cout3 : in  STD_LOGIC_VECTOR (3 downto 0);
           prod : out  STD_LOGIC_VECTOR (7 downto 0));
end RIP_array_multiplier;

architecture Behavioral of RIP_array_multiplier is

component array_multiplier is
      Port ( cin : in std_logic_vector(3 downto 0);
             cout : in std_logic_vector(3 downto 0);
             prod : out std_logic_vector(7 downto 0));
end component array_multiplier;

signal prod0,prod1,prod2 : std_logic_vector(7 downto 0);
signal r_prod0,r_prod1,r_prod2 : std_logic_vector(3 downto 0);
signal r0_cout2,r0_cout3,r1_cout3 : std_logic_vector(3 downto 0);

begin

reg: process(clk)
begin
if (reset = '1') then
r_prod0 <= (others => '0');
r_prod1 <= (others => '0');
r_prod2 <= (others => '0');
r0_cout2 <= (others => '0');
r0_cout3 <= (others => '0');
r1_cout3 <= (others => '0');
else
r0_cout2 <= cout2;
r0_cout3 <= cout3;
r1_cout3 <= r0_cout3;

r_prod0 <= prod0(7 downto 4);
r_prod1 <= prod1(7 downto 4);
prod    <= prod2;
end if;
```

```
end process  reg;

AM0 : array_multiplier
      port map ( cin => cin,
            cout => cout1,
            prod => prod0);

AM1 : array_multiplier
      port map ( cin => r_prod0,
            cout => r0_cout2,
            prod => prod1);


AM2 : array_multiplier
      port map ( cin => r_prod1,
            cout => r1_cout3,
            prod => prod2);
end Behavioral;
```

Carry Select Adder με τεχνική Pipeline

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

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity CarrySkipAdder16Reg is
    Port ( clk : STD_LOGIC;
                  reset : STD_LOGIC;
                  AB : in  STD_LOGIC_VECTOR (31 downto 0);
           C : out  STD_LOGIC_VECTOR (15 downto 0);
           Co : out  STD_LOGIC);
end CarrySkipAdder16Reg;

architecture Behavioral of CarrySkipAdder16Reg is
component CarrySkipAdder16 is
    Port ( A : in  STD_LOGIC_VECTOR (15 downto 0);
           B : in  STD_LOGIC_VECTOR (15 downto 0);
           O : out  STD_LOGIC_VECTOR (15 downto 0);
           co : out  STD_LOGIC);
end component CarrySkipAdder16;
signal A : std_logic_vector(15 downto 0);
signal B : std_logic_vector(15 downto 0);
begin

process(clk) is
begin
if rising_edge(clk) then
      if(reset = '1') then
            A <= X"0000";
```

```
            B <= X"0000";
      else
            A <= AB(31 downto 16);
            B <= AB(15 downto 0);
      end if;
end if;
end process;

CSA0 : CarrySkipAdder16
    port map ( A => A,
                              B => B,
                              O => C,
                              co => Co
                              );

end Behavioral;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity PIPCarrySelectAdder16 is
    Port ( AB : in  STD_LOGIC_VECTOR (31 downto 0);
           C : out  STD_LOGIC_VECTOR (15 downto 0);
           Co : out  STD_LOGIC;
           clk : in  STD_LOGIC;
           reset : in  STD_LOGIC);
end PIPCarrySelectAdder16;

architecture Behavioral of PIPCarrySelectAdder16 is
component CarrySkipAdder16Reg is
    Port ( clk : STD_LOGIC;
                   reset : STD_LOGIC;
                   AB : in  STD_LOGIC_VECTOR (31 downto 0);
           C : out  STD_LOGIC_VECTOR (15 downto 0);
           Co : out  STD_LOGIC);
end component CarrySkipAdder16Reg;

component mux is
    Port ( A : in  STD_LOGIC_VECTOR(16 DOWNTO 0);
           B : in  STD_LOGIC_VECTOR(16 DOWNTO 0);
           SEL : in  STD_LOGIC;
           O : out  STD_LOGIC_VECTOR(16 DOWNTO 0));
end component mux;

signal C_shift : std_logic_vector(33 downto 0);
signal C_pip0     : std_logic_vector(16 downto 0);
signal C_pip1     : std_logic_vector(16 downto 0);
signal sel        : std_logic;
```

```
begin

proc_sell: process(clk) is
begin
  if(falling_edge(clk)) then
    sel   <= not sel;
  end if;
end process proc_sell;

RCSA0 : CarrySkipAdder16Reg
    port map ( clk => clk,
                   reset => reset,
                   AB => AB,
           C => C_pip0(15 downto 0),
           Co => C_pip0(16));

RCSA1 : CarrySkipAdder16Reg
    port map ( clk => clk,
                   reset => reset,
                   AB => AB,
           C => C_pip1(15 downto 0),
           Co => C_pip1(16));

multiplexer : mux port map (A => C_pip0, B => C_pip1, SEL =>
sel);


end Behavioral;
```

Wallace Tree με τεχνική Pipeline

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

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity RIP_wallace_tree is
    Port ( clk : in  STD_LOGIC;
           reset : in  STD_LOGIC;
           X : in  STD_LOGIC_VECTOR (47 downto 0);
           S : out  STD_LOGIC_VECTOR (10 downto 0));
end RIP_wallace_tree;

architecture Behavioral of RIP_wallace_tree is

component wallace_tree is
    Port ( S : out  STD_LOGIC_VECTOR (10 downto 0);
           X : in  STD_LOGIC_VECTOR (47 downto 0));
end component;
```

```
begin

end Behavioral;
```

Παράρτημα Β – Κώδικες TESTBENCH Κυκλωμάτων

Σειριακός αθροιστής (BitSerialAdder)

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--USE ieee.numeric_std.ALL;

ENTITY tb_serial_adder IS
END tb_serial_adder;

ARCHITECTURE behavior OF tb_serial_adder IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT serial_adder
    PORT(
         I_m : IN  std_logic_vector(7 downto 0);
         I_l : IN  std_logic_vector(7 downto 0);
         O : OUT  std_logic;
         c_i : IN  std_logic;
         c_o : OUT  std_logic;
         load : IN  std_logic;
         CLK : IN  std_logic
        );
    END COMPONENT;


   --Inputs
   signal I_m : std_logic_vector(7 downto 0) := (others => '0');
   signal I_l : std_logic_vector(7 downto 0) := (others => '0');
   signal c_i : std_logic := '0';
   signal load : std_logic := '0';
   signal CLK : std_logic := '0';

      --Outputs
   signal O : std_logic;
   signal c_o : std_logic;

   -- Clock period definitions
   constant CLK_period : time := 10 ns;

BEGIN

      -- Instantiate the Unit Under Test (UUT)
   uut: serial_adder PORT MAP (
          I_m => I_m,
          I_l => I_l,
          O => O,
          c_i => c_i,
          c_o => c_o,
          load => load,
          CLK => CLK
        );

   -- Clock process definitions
```

```
   CLK_process :process
   begin
           CLK <= '0';
           wait for CLK_period/2;
           CLK <= '1';
           wait for CLK_period/2;
   end process;


   -- Stimulus process
   stim_proc: process
   begin

           load <= '1';
      wait for 100 ns;
           I_m <= "00000000";
           I_l <= "00000000";
           c_i <= '0';
      wait for CLK_period*10;
           I_m <= "00000001";
           I_l <= "00000001";
           c_i <= '0';
      wait for CLK_period*10;
           I_m <= "00000011";
           I_l <= "00000011";
           c_i <= '0';
      wait for CLK_period*10;

      -- insert stimulus here

      wait;
   end process;

END;
```

Αθροιστής μετάδοσης κρατουμένου (RCA – RippleCarryAdder)

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--USE ieee.numeric_std.ALL;

ENTITY tb_Ripple_Adder IS
END tb_Ripple_Adder;

ARCHITECTURE behavior OF tb_Ripple_Adder IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT Ripple_Adder
    PORT(
         A : IN  std_logic_vector(3 downto 0);
         B : IN  std_logic_vector(3 downto 0);
         Cin : IN  std_logic;
         S : OUT  std_logic_vector(3 downto 0);
         Cout : OUT  std_logic
```

```
        );
    END COMPONENT;


   --Inputs
   signal A : std_logic_vector(3 downto 0) := (others => '0');
   signal B : std_logic_vector(3 downto 0) := (others => '0');
   signal Cin : std_logic := '0';

      --Outputs
   signal S : std_logic_vector(3 downto 0);
   signal Cout : std_logic;
   signal clk : std_logic;
   -- No clocks detected in port list. Replace <clock> below with
   -- appropriate port name

   constant clk_period : time := 10 ns;

BEGIN

      -- Instantiate the Unit Under Test (UUT)
   uut: Ripple_Adder PORT MAP (
          A => A,
          B => B,
          Cin => Cin,
          S => S,
          Cout => Cout
        );

   -- Clock process definitions
   clk_process :process
   begin
           clk <= '0';
           wait for clk_period/2;
           clk <= '1';
           wait for clk_period/2;
   end process;


   -- Stimulus process
   stim_proc: process
   begin
      -- hold reset state for 100 ns.
      wait for 100 ns;
           A <= "0001"; -- 0x1
           B <= "0011"; -- 0x3
           Cin <= '0';
      wait for clk_period*10;
           A <= "0011"; -- 0x3
           B <= "0011"; -- 0x3
      wait for clk_period*10;
           A <= "1111"; -- 0xF
           B <= "1111"; -- 0xF
      wait for clk_period*10;

      -- insert stimulus here

      wait;
```

```
   end process;

END;
```

Αθροιστής πρόβλεψης κρατουμένου (CLA – CarryLookaheadAdder)

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--USE ieee.numeric_std.ALL;

ENTITY tb_carrLookAheadAdder IS
END tb_carrLookAheadAdder;

ARCHITECTURE behavior OF tb_carrLookAheadAdder IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT carryLookAheadAdder
    PORT(
         A : IN  std_logic_vector(7 downto 0);
         B : IN  std_logic_vector(7 downto 0);
         O : OUT  std_logic_vector(8 downto 0)
        );
    END COMPONENT;


   --Inputs
   signal A : std_logic_vector(7 downto 0) := (others => '0');
   signal B : std_logic_vector(7 downto 0) := (others => '0');

      --Outputs
   signal O : std_logic_vector(8 downto 0);
   -- No clocks detected in port list. Replace <clock> below with
   -- appropriate port name


BEGIN

      -- Instantiate the Unit Under Test (UUT)
   uut: carryLookAheadAdder PORT MAP (
          A => A,
          B => B,
          O => O
        );




   -- Stimulus process
   stim_proc: process
   begin
      -- hold reset state for 100 ns.
      wait for 100 ns;
           A<="00000001"; -- 0x01
           B<="00000010"; -- 0x02
```

```
      wait for 50 ns;
           A<="00000001"; -- 0x01
           B<="00000010"; -- 0x02
      wait for 50 ns;
           A<="00000011"; -- 0x03
           B<="00000010"; -- 0x02
      wait for 50 ns;
           A<="00000111"; -- 0x07
           B<="00000010"; -- 0x02
      wait for 50 ns;
           A<="00001111"; -- 0x0F
           B<="00000010"; -- 0x02
      wait for 50 ns;
           A<="11111111"; -- 0xFF
           B<="11111111"; -- 0xFF
      wait for 50 ns;

      wait;
   end process;

END;
```

Αθροιστής παράκαμψης κρατουμένου (CSK – CarrySkipAdder)

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--USE ieee.numeric_std.ALL;

ENTITY tb_CarrySkipAdder IS
END tb_CarrySkipAdder;

ARCHITECTURE behavior OF tb_CarrySkipAdder IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT CarrySkipAdder16
    PORT(
         A : IN  std_logic_vector(15 downto 0);
         B : IN  std_logic_vector(15 downto 0);
         O : OUT  std_logic_vector(15 downto 0);
         co : OUT  std_logic
        );
    END COMPONENT;


   --Inputs
   signal A : std_logic_vector(15 downto 0) := (others => '0');
   signal B : std_logic_vector(15 downto 0) := (others => '0');

      --Outputs
   signal O : std_logic_vector(15 downto 0);
   signal co : std_logic;

   signal clk : std_logic;
   -- No clocks detected in port list. Replace <clock> below with
```

```
   -- appropriate port name

   constant clk_period : time := 10 ns;

BEGIN

      -- Instantiate the Unit Under Test (UUT)
   uut: CarrySkipAdder16 PORT MAP (
          A => A,
          B => B,
          O => O,
          co => co
        );

   -- Clock process definitions
   clk_process :process
   begin
           clk <= '0';
           wait for clk_period/2;
           clk <= '1';
           wait for clk_period/2;
   end process;


   -- Stimulus process
   stim_proc: process
   begin
      -- hold reset state for 100 ns.
      wait for 100 ns;
           A <= "0000000000000001"; -- 0x0001
           B <= "0000000000000001"; -- 0x0001
      wait for clk_period*10;
           A <= "0000000000000010"; -- 0x0002
           B <= "0000000000000001"; -- 0x0001
      wait for clk_period*10;
           A <= "0000000000000011"; -- 0x0003
           B <= "0000000000000001"; -- 0x0001
      wait for clk_period*10;
           A <= "0000000000000011"; -- 0x0003
           B <= "0000000000000011"; -- 0x0003
      wait for clk_period*10;
           A <= "1111111111111111"; -- 0xFFFF
           B <= "0000000000000001"; -- 0x0001
      wait for clk_period*10;
           A <= "0000000000001111"; -- 0x000F
           B <= "0000000000001110"; -- 0x000E
      wait for clk_period*10;


      wait;
   end process;

END;
```

Αθροιστής επιλογής κρατουμένου (CSL – CarrySelectAdder)

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
```

```
-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--USE ieee.numeric_std.ALL;

ENTITY tb_carry_select_adder IS
END tb_carry_select_adder;

ARCHITECTURE behavior OF tb_carry_select_adder IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT carry_select_adder
    PORT(
         X : IN  std_logic_vector(3 downto 0);
         Y : IN  std_logic_vector(3 downto 0);
         CARRY_IN : IN  std_logic;
         SUM : OUT  std_logic_vector(3 downto 0);
         CARRY_OUT : OUT  std_logic
        );
    END COMPONENT;


   --Inputs
   signal X : std_logic_vector(3 downto 0) := (others => '0');
   signal Y : std_logic_vector(3 downto 0) := (others => '0');
   signal CARRY_IN : std_logic := '0';

     --Outputs
   signal SUM : std_logic_vector(3 downto 0);
   signal CARRY_OUT : std_logic;
   -- No clocks detected in port list. Replace <clock> below with
   -- appropriate port name


BEGIN

     -- Instantiate the Unit Under Test (UUT)
   uut: carry_select_adder PORT MAP (
          X => X,
          Y => Y,
          CARRY_IN => CARRY_IN,
          SUM => SUM,
          CARRY_OUT => CARRY_OUT
        );



   -- Stimulus process
   stim_proc: process
   begin
      -- hold reset state for 100 ns.
      wait for 100 ns;

           X <= "0000"; -- 0x0
           Y <= "0001"; -- 0x1
           CARRY_IN <= '1';
      wait for 50 ns;
```

```
            X <= "0011"; -- 0x3
            Y <= "0001"; -- 0x1
            CARRY_IN <= '1';
      wait for 50 ns;
            X <= "1111"; -- 0xF
            Y <= "1111"; -- 0xF
            CARRY_IN <= '0';



      -- insert stimulus here

      wait;
   end process;

END;
```

Αθροιστής με δέντρο Wallace

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--USE ieee.numeric_std.ALL;

ENTITY tb_wallace_tree IS
END tb_wallace_tree;

ARCHITECTURE behavior OF tb_wallace_tree IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT wallace_tree
    PORT(
         S : OUT  std_logic_vector(10 downto 0);
         X : IN  std_logic_vector(47 downto 0)
        );
    END COMPONENT;


   --Inputs
   signal X : std_logic_vector(47 downto 0) := (others => '0');

      --Outputs
   signal S : std_logic_vector(10 downto 0);
   -- No clocks detected in port list. Replace <clock> below with
   -- appropriate port name

BEGIN

      -- Instantiate the Unit Under Test (UUT)
   uut: wallace_tree PORT MAP (
          S => S,
          X => X
        );
```

```
   -- Stimulus process
   stim_proc: process
   begin
      -- hold reset state for 100 ns.
      wait for 100 ns;
           X <= X"000000000000";
      -- insert stimulus here

      wait for 100 ns;
   end process;

END;
```

Πολλαπλασιαστής Booth των n·mbits βάσης-2

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--USE ieee.numeric_std.ALL;

ENTITY tb_booth_multi_radix IS
END tb_booth_multi_radix;

ARCHITECTURE behavior OF tb_booth_multi_radix IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT booth_multi_radix
    PORT(
         c : IN  std_logic_vector(8 downto 0);
         d : IN  std_logic_vector(8 downto 0);
         cin : IN  std_logic_vector(4 downto 0);
         cout : OUT  std_logic_vector(13 downto 0)
        );
    END COMPONENT;


   --Inputs
   signal c : std_logic_vector(8 downto 0) := (others => '0');
   signal d : std_logic_vector(8 downto 0) := (others => '0');
   signal cin : std_logic_vector(4 downto 0) := (others => '0');

      --Outputs
   signal cout : std_logic_vector(13 downto 0);
   -- No clocks detected in port list. Replace <clock> below with
   -- appropriate port name


BEGIN

      -- Instantiate the Unit Under Test (UUT)
   uut: booth_multi_radix PORT MAP (
          c => c,
          d => d,
          cin => cin,
```

```
          cout => cout
        );


   -- Stimulus process
   stim_proc: process
   begin
      -- hold reset state for 100 ns.
      wait for 100 ns;
           c <= "000000001"; -- 0x001
           d <= "000000011"; -- 0x003
           cin <= "00000";
      wait for 50 ns;
           c <= "000000011"; -- 0x003
           d <= "000000011"; -- 0x003
           cin <= "00000";
      wait for 50 ns;
           c <= "111111111"; -- 0x1FF
           d <= "111111111"; -- 0x1FF
           cin <= "11111";

      wait for 50 ns;

      -- insert stimulus here

      wait;
   end process;

END;
```

Πολλαπλασιαστής Booth των n·mbits βάσης-4

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--USE ieee.numeric_std.ALL;

ENTITY tb_both_multi_radix4 IS
END tb_both_multi_radix4;

ARCHITECTURE behavior OF tb_both_multi_radix4 IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT booth_multi_radix4
    PORT(
         sin : IN  std_logic_vector(7 downto 0);
         par : IN  std_logic_vector(2 downto 0);
         sout : OUT  std_logic_vector(15 downto 0)
        );
    END COMPONENT;


   --Inputs
   signal sin : std_logic_vector(7 downto 0) := (others => '0');
```

```
   signal par : std_logic_vector(2 downto 0) := (others => '0');

      --Outputs
   signal sout : std_logic_vector(15 downto 0);
   -- No clocks detected in port list. Replace <clock> below with
   -- appropriate port name


BEGIN

      -- Instantiate the Unit Under Test (UUT)
   uut: booth_multi_radix4 PORT MAP (
          sin => sin,
          par => par,
          sout => sout
        );




   -- Stimulus process
   stim_proc: process
   begin
      -- hold reset state for 100 ns.
      wait for 100 ns;
           sin <= "00000000"; -- 0x00
           par <= "001";      --0x1
      wait for 50 ns;
           sin <= "00000001"; -- 0x01
           par <= "001";      --0x1
      wait for 50 ns;
           sin <= "00000011"; -- 0x03
           par <= "001";      --0x1      -- insert stimulus here

      wait;
   end process;

END;
```

Πολλαπλασιασμός αριθμών με δέντρο Wallace και Dadda

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--USE ieee.numeric_std.ALL;

ENTITY tb_dadda_multi IS
END tb_dadda_multi;

ARCHITECTURE behavior OF tb_dadda_multi IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT dadda_multi
    PORT(
         xin : IN  std_logic_vector(7 downto 0);
```

```
         yin : IN  std_logic_vector(7 downto 0);
         zout : OUT  std_logic_vector(15 downto 0)
        );
    END COMPONENT;


   --Inputs
   signal xin : std_logic_vector(7 downto 0) := (others => '0');
   signal yin : std_logic_vector(7 downto 0) := (others => '0');

      --Outputs
   signal zout : std_logic_vector(15 downto 0);
   -- No clocks detected in port list. Replace <clock> below with
   -- appropriate port name


BEGIN

      -- Instantiate the Unit Under Test (UUT)
   uut: dadda_multi PORT MAP (
          xin => xin,
          yin => yin,
          zout => zout
        );


   -- Stimulus process
   stim_proc: process
   begin
      -- hold reset state for 100 ns.
      wait for 100 ns;
           xin <= "00000000"; --0x00
           yin <= "00000001"; --0x01
      wait for 50 ns;
           xin <= "00000011"; --0x03
           yin <= "00000001"; --0x01
      wait for 50 ns;
           xin <= "00000111"; --0x07
           yin <= "00000111"; --0x07
      wait for 50 ns;

      wait for 50 ns;
      -- insert stimulus here

      wait;
   end process;

END;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity full_adder is
Port ( bit1 : in  STD_LOGIC;
       bit2 : in  STD_LOGIC;
       cin  : in  STD_LOGIC;
       sout : out  STD_LOGIC;
       cout : out  STD_LOGIC);
```

```
end full_adder;

architecture Behavioral of full_adder is
begin
      sout <= (bit1 xor bit2 xor cin);
      cout <= (bit1 and bit2) xor (cin and (bit1 xor bit2));
end Behavioral;
----------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity half_adder is
Port ( bit1 : in  STD_LOGIC;
       bit2 : in  STD_LOGIC;
       sout : out  STD_LOGIC;
       cout : out  STD_LOGIC);
end half_adder;

architecture Behavioral of half_adder is
begin
      sout <= bit1 xor bit2;
      cout <= bit1 and bit2;
end Behavioral;
-------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity wallace4 is
Port ( x : in  STD_LOGIC_VECTOR (3 downto 0);
           y : in  STD_LOGIC_VECTOR (3 downto 0);
           Product : out  STD_LOGIC_VECTOR (7 downto 0));
end wallace4;

architecture Behavioral of wallace4 is
component full_adder is
    Port ( bit1 : in  STD_LOGIC;
       bit2 : in  STD_LOGIC;
       cin  : in  STD_LOGIC;
       sout : out  STD_LOGIC;
       cout : out  STD_LOGIC);
end component;

component half_adder is
    Port ( bit1 : in  STD_LOGIC;
       bit2 : in  STD_LOGIC;
       sout : out  STD_LOGIC;
       cout : out  STD_LOGIC);
end component;

signal
s11,s12,s13,s14,s15,s22,s23,s24,s25,s26,s32,s34,s35,s36,s37 :
std_logic;
signal
c11,c12,c13,c14,c15,c22,c23,c24,c25,c26,c32,c34,c35,c36,c37 :
std_logic;
signal pr0,pr1,pr2,pr3 : std_logic_vector(3 downto 0);

begin
```

```
process(x,y)
begin
    for i in 0 to 3 loop
        pr0(i) <= x(i) and y(0);
        pr1(i) <= x(i) and y(1);
        pr2(i) <= x(i) and y(2);
        pr3(i) <= x(i) and y(3);
    end loop;
end process;

Product(0) <= pr0(0);
Product(1) <= s11;
Product(2) <= s22;
Product(3) <= s32;
Product(4) <= s34;
Product(5) <= s35;
Product(6) <= s36;
Product(7) <= s37;

--first stage
ha11 : half_adder port map(pr0(1),pr1(0),s11,c11);
fa12 : full_adder port map(pr0(2),pr1(1),pr2(0),s12,c12);
fa13 : full_adder port map(pr0(3),pr1(2),pr2(1),s13,c13);
fa14 : full_adder port map(pr1(3),pr2(2),pr3(1),s14,c14);
ha15 : half_adder port map(pr2(3),pr3(2),s15,c15);

--second stage
ha22 : half_adder port map(c11,s12,s22,c22);
fa23 : full_adder port map(pr3(0),c12,s13,s23,c23);
fa24 : full_adder port map(c13,c23,s14,s24,c24);
fa25 : full_adder port map(c14,c24,s15,s25,c25);
fa26 : full_adder port map(c15,c25,pr3(3),s26,c26);

--third stage
ha32 : half_adder port map(c22,s23,s32,c32);
ha34 : half_adder port map(c32,s24,s34,c34);
ha35 : half_adder port map(c34,s25,s35,c35);
ha36 : half_adder port map(c35,s26,s36,c36);
ha37 : half_adder port map(c36,c26,s37,c37);

end Behavioral;
```

Πολλαπλασιαστής με διάταξη πίνακα (ArrayMultiplier)

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--USE ieee.numeric_std.ALL;

ENTITY tb_array_multiplier IS
END tb_array_multiplier;

ARCHITECTURE behavior OF tb_array_multiplier IS
```

```
    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT array_multiplier
    PORT(
         -- clk : IN  std_logic;
         cin : IN  std_logic_vector(3 downto 0);
         cout : IN  std_logic_vector(3 downto 0);
         prod : OUT  std_logic_vector(7 downto 0)
        );
    END COMPONENT;


   --Inputs
   signal clk : std_logic := '0';
   signal cin : std_logic_vector(3 downto 0) := (others => '0');
   signal cout : std_logic_vector(3 downto 0) := (others => '0');

     --Outputs
   signal prod : std_logic_vector(7 downto 0);

   -- Clock period definitions
   constant clk_period : time := 10 ns;

BEGIN

     -- Instantiate the Unit Under Test (UUT)
   uut: array_multiplier PORT MAP (
          -- clk => clk,
          cin => cin,
          cout => cout,
          prod => prod
        );

   -- Clock process definitions
   clk_process :process
   begin
           clk <= '0';
           wait for clk_period/2;
           clk <= '1';
           wait for clk_period/2;
   end process;


   -- Stimulus process
   stim_proc: process
   begin

      wait for 100 ns;
           cin  <= "0001";  -- 0x1
           cout <= "0001";  -- 0x1
      wait for clk_period*10;
           cin  <= "0011";  -- 0x3
           cout <= "0011";  -- 0x3
      wait for clk_period*10;
           cin  <= "0010";  -- 0x2
           cout <= "0011";  -- 0x3
      wait for clk_period*10;
```

```
      -- insert stimulus here

      wait;
   end process;

END;
```

Modular πολλαπλασιαστής με βάση τον αλγόριθμο Montgomery

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--USE ieee.numeric_std.ALL;

ENTITY tb_Montgomery_multiplier IS
END tb_Montgomery_multiplier;

ARCHITECTURE behavior OF tb_Montgomery_multiplier IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT Montgomery_multiplier
    PORT(
         ain : IN  std_logic_vector(7 downto 0);
         bin : IN  std_logic_vector(7 downto 0);
         clk : IN  std_logic;
         reset : IN  std_logic;
         start : IN  std_logic;
         pout : OUT  std_logic_vector(7 downto 0);
         oksig : OUT  std_logic
        );
    END COMPONENT;


   --Inputs
   signal ain : std_logic_vector(7 downto 0) := (others => '0');
   signal bin : std_logic_vector(7 downto 0) := (others => '0');
   signal clk : std_logic := '0';
   signal reset : std_logic := '0';
   signal start : std_logic := '0';

      --Outputs
   signal pout : std_logic_vector(7 downto 0);
   signal oksig : std_logic;

   -- Clock period definitions
   constant clk_period : time := 10 ns;

BEGIN

      -- Instantiate the Unit Under Test (UUT)
   uut: Montgomery_multiplier PORT MAP (
          ain => ain,
          bin => bin,
          clk => clk,
          reset => reset,
```

```
          start => start,
          pout => pout,
          oksig => oksig
        );

   -- Clock process definitions
   clk_process :process
   begin
           clk <= '0';
           wait for clk_period/2;
           clk <= '1';
           wait for clk_period/2;
   end process;


   -- Stimulus process
   stim_proc: process
   begin
      reset <= '1';
      wait for 100 ns;
           reset <= '0';
      wait for clk_period*10;
           start <= '1';
           ain <= "00000001"; -- 0x01
           bin <= "00000010"; -- 0x02
      wait for clk_period*10;
           ain <= "00000010"; -- 0x02
           bin <= "00000010"; -- 0x02
      wait for clk_period*10;
           ain <= "00000011"; -- 0x03
           bin <= "00000010"; -- 0x02
      -- insert stimulus here

      wait;
   end process;

END;
```

Ripple Adder με τεχνική Pipeline

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--USE ieee.numeric_std.ALL;

ENTITY tb_RIP_Ripple_Adder IS
END tb_RIP_Ripple_Adder;

ARCHITECTURE behavior OF tb_RIP_Ripple_Adder IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT RIP_Ripple_Adder
    PORT(
         clk : IN  std_logic;
         reset : IN  std_logic;
```

```
        A : IN  std_logic_vector(3 downto 0);
        B : IN  std_logic_vector(3 downto 0);
        C : IN  std_logic_vector(3 downto 0);
        D : IN  std_logic_vector(3 downto 0);
        Ci : IN  std_logic;
        Co : OUT  std_logic;
        S : OUT  std_logic_vector(3 downto 0)
       );
    END COMPONENT;


   --Inputs
   signal clk : std_logic := '0';
   signal reset : std_logic := '0';
   signal A : std_logic_vector(3 downto 0) := (others => '0');
   signal B : std_logic_vector(3 downto 0) := (others => '0');
   signal C : std_logic_vector(3 downto 0) := (others => '0');
   signal D : std_logic_vector(3 downto 0) := (others => '0');
   signal Ci : std_logic := '0';

      --Outputs
   signal Co : std_logic;
   signal S : std_logic_vector(3 downto 0);

   -- Clock period definitions
   constant clk_period : time := 10 ns;

BEGIN

      -- Instantiate the Unit Under Test (UUT)
   uut: RIP_Ripple_Adder PORT MAP (
          clk => clk,
          reset => reset,
          A => A,
          B => B,
          C => C,
          D => D,
          Ci => Ci,
          Co => Co,
          S => S
        );

   -- Clock process definitions
   clk_process :process
   begin
         clk <= '0';
         wait for clk_period/2;
         clk <= '1';
         wait for clk_period/2;
   end process;


   -- Stimulus process
   stim_proc: process
   begin
      reset <= '1';
      wait for 100 ns;
         reset <= '0';
```

```
            A  <= "0000";
            B  <= "0001";
            C  <= "0001";
            D  <= "0001";
            Ci <= '0';

      wait for clk_period*10;

      -- insert stimulus here

      wait;
   end process;

END;
```

Array Multiplier με τεχνική Pipeline

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--USE ieee.numeric_std.ALL;

ENTITY tb_RIP_array_multiplier IS
END tb_RIP_array_multiplier;

ARCHITECTURE behavior OF tb_RIP_array_multiplier IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT RIP_array_multiplier
    PORT(
         clk : IN  std_logic;
         reset : IN  std_logic;
         cin : IN  std_logic_vector(3 downto 0);
         cout1 : IN  std_logic_vector(3 downto 0);
         cout2 : IN  std_logic_vector(3 downto 0);
         cout3 : IN  std_logic_vector(3 downto 0);
         prod : OUT  std_logic_vector(7 downto 0)
        );
    END COMPONENT;


   --Inputs
   signal clk : std_logic := '0';
   signal reset : std_logic := '0';
   signal cin : std_logic_vector(3 downto 0) := (others => '0');
   signal cout1 : std_logic_vector(3 downto 0) := (others =>
'0');
   signal cout2 : std_logic_vector(3 downto 0) := (others =>
'0');
   signal cout3 : std_logic_vector(3 downto 0) := (others =>
'0');

      --Outputs
   signal prod : std_logic_vector(7 downto 0);
```

```
   -- Clock period definitions
   constant clk_period : time := 10 ns;

BEGIN

      -- Instantiate the Unit Under Test (UUT)
   uut: RIP_array_multiplier PORT MAP (
          clk => clk,
          reset => reset,
          cin => cin,
          cout1 => cout1,
          cout2 => cout2,
          cout3 => cout3,
          prod => prod
        );

   -- Clock process definitions
   clk_process :process
   begin
           clk <= '0';
           wait for clk_period/2;
           clk <= '1';
           wait for clk_period/2;
   end process;


   -- Stimulus process
   stim_proc: process
   begin
      reset <= '1';
      wait for 100 ns;
      reset <= '0';
           cin <= "1001";
           cout1 <= "1001";
           cout2 <= "1001";
           cout3 <= "1001";
      -- insert stimulus here

      wait;
   end process;

END;
```

Carry Select Adder με τεχνική Pipeline

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--USE ieee.numeric_std.ALL;

ENTITY tb_PIPCarrySelectAdder16 IS
END tb_PIPCarrySelectAdder16;

ARCHITECTURE behavior OF tb_PIPCarrySelectAdder16 IS

    -- Component Declaration for the Unit Under Test (UUT)
```

```
    COMPONENT PIPCarrySelectAdder16
    PORT(
         AB : IN  std_logic_vector(31 downto 0);
         C : OUT  std_logic_vector(15 downto 0);
         Co : OUT  std_logic;
         clk : IN  std_logic;
         reset : IN  std_logic
        );
    END COMPONENT;


   --Inputs
   signal AB : std_logic_vector(31 downto 0) := (others => '0');
   signal clk : std_logic := '0';
   signal reset : std_logic := '0';

 	--Outputs
   signal C : std_logic_vector(15 downto 0);
   signal Co : std_logic;

   -- Clock period definitions
   constant clk_period : time := 10 ns;

BEGIN

	-- Instantiate the Unit Under Test (UUT)
   uut: PIPCarrySelectAdder16 PORT MAP (
          AB => AB,
          C => C,
          Co => Co,
          clk => clk,
          reset => reset
        );

   -- Clock process definitions
   clk_process :process
   begin
		clk <= '0';
		wait for clk_period/2;
		clk <= '1';
		wait for clk_period/2;
   end process;


   -- Stimulus process
   stim_proc: process
   begin
		reset <= '1';
      wait for 100 ns;

		reset <= '0';
      AB <= X"00010001";
      wait until rising_edge(clk);
      AB <= X"00010002";

      wait;
   end process;
```

```
END;
```

Wallace Tree με τεχνική Pipeline

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--USE ieee.numeric_std.ALL;

ENTITY tb_wallace_tree IS
END tb_wallace_tree;

ARCHITECTURE behavior OF tb_wallace_tree IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT wallace_tree
    PORT(
         S : OUT  std_logic_vector(10 downto 0);
         X : IN  std_logic_vector(47 downto 0)
        );
    END COMPONENT;


   --Inputs
   signal X : std_logic_vector(47 downto 0) := (others => '0');

      --Outputs
   signal S : std_logic_vector(10 downto 0);
   -- No clocks detected in port list. Replace <clock> below with
   -- appropriate port name

BEGIN

      -- Instantiate the Unit Under Test (UUT)
   uut: wallace_tree PORT MAP (
          S => S,
          X => X
        );



   -- Stimulus process
   stim_proc: process
   begin
      -- hold reset state for 100 ns.
      wait for 100 ns;
           X <= X"000000000000";
      -- insert stimulus here

      wait for 100 ns;
   end process;

END;
```