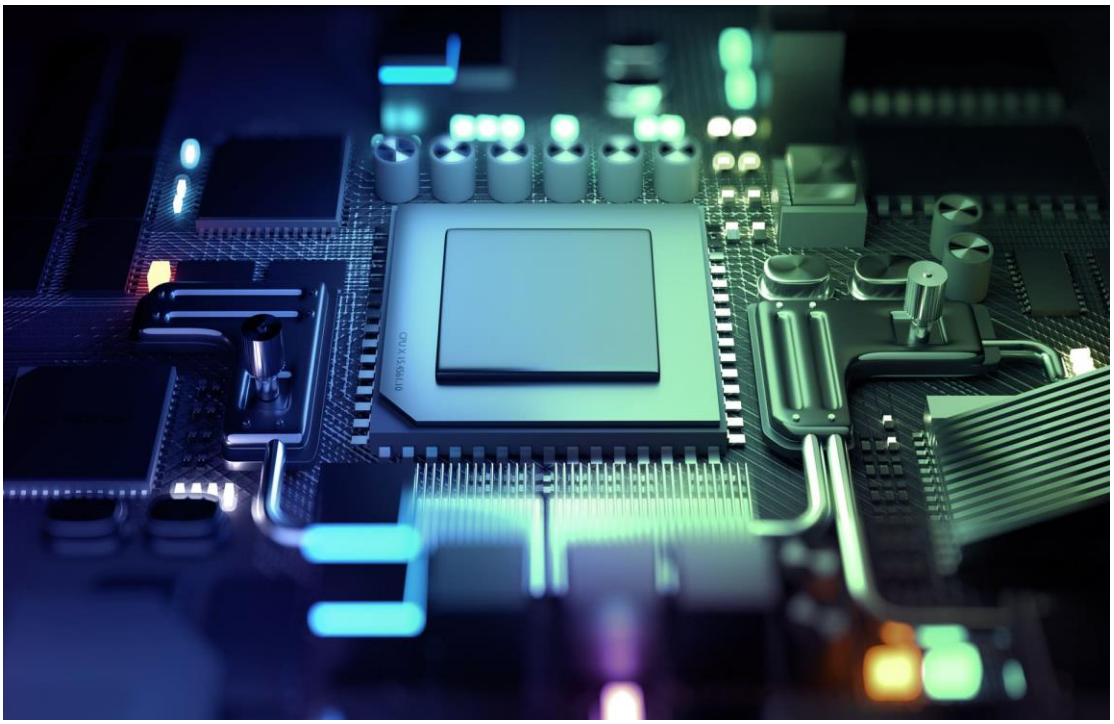
ΠΑΝΕΠΙΣΤΗΜΙΟ ΔΥΤΙΚΗΣ ΑΤΤΙΚΗΣ
ΤΜΗΜΑ : ΜΗΧΑΝΙΚΩΝ
ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ
ΥΠΟΛΟΓΙΣΤΩΝ

Διπλωματική Εργασία

Σχεδίαση και προσομοίωση βασικών και αριθμητικών κυκλωμάτων με χρήση των γλωσσών περιγραφής υλικού VHDL και Verilog



Μαστρογιάννης Δημήτριος
Α.Μ. : 71343858

Δημόπουλος Δημοσθένης
Α.Μ. : 71344464

Επιβλέπων Καθηγητής : Ευστάθιου Κωνσταντίνος

Αθήνα, Νοέμβριος 2021

UNIVERSITY OF WEST ATTICA
DEPARTMENT OF
INFORMATICS AND
COMPUTER ENGINEERING

Diploma Thesis

Design and simulation of basic and numerical circuits using the hardware description languages VHDL and Verilog

Mastrogiannis Dimitrios
Registration Number: 71343858

Dimopoulos Dimosthenis
Registration Number: 71344464

Supervisor: Konstantinos Efstathiou

Athens, November 2021

ΠΑΝΕΠΙΣΤΗΜΙΟ ΔΥΤΙΚΗΣ ΑΤΤΙΚΗΣ

ΣΧΟΛΗ

ΤΜΗΜΑ

Σχεδίαση και προσομοίωση βασικών και αριθμητικών κυκλωμάτων με χρήση των γλωσσών περιγραφής υλικού VHDL και Verilog

Μέλη Εξεταστικής Επιτροπής συμπεριλαμβανομένου και του Εισηγητή

Ιωάννης Βογιατζής

Ιωάννης Αμοργίνος

Κωνσταντίνος Ευσταθίου

Η πτυχιακή/διπλωματική εργασία εξετάστηκε επιτυχώς από την κάτωθι Εξεταστική Επιτροπή:

Α/α	ΟΝΟΜΑ ΕΠΩΝΥΜΟ	ΒΑΘΜΙΔΑ/ΙΔΙΟΤΗΤΑ	ΨΗΦΙΑΚΗ ΥΠΟΓΡΑΦΗ
1	Ιωάννης Βογιατζής	Καθηγητής Πανεπιστημίου Δυτικής Αττικής	Ioannis Voyiatzis Digitally signed by Ioannis Voyiatzis Date: 2021.11.24 16:02:20 +02'00'
2	Ιωάννης Αμοργίνος	Λέκτορας Εφαρμογών Πανεπιστημίου Δυτικής Αττικής	Ioannis Amorginos Digitally signed by Ioannis Amorginos Date: 2021.11.20 21:55:34 +02'00'
3	Κωνσταντίνος Ευσταθίου	Καθηγητής Πανεπιστημίου Δυτικής Αττικής	Konstantinos Efstathiou Digitally signed by Konstantinos Efstathiou Date: 2021.11.08 11:50:22 +02'00'

Δήλωση Συγγραφέων Διπλωματικής Εργασίας

Ο κάτωθι υπογεγραμμένος Μαστρογιάννης Δημήτριος του Μιχαήλ, με αριθμό μητρώου 71343858, φοιτητής του Πανεπιστημίου Δυτικής Αττικής της Σχολής Μηχανικών του Τμήματος Μηχανικών Πληροφορικής και Υπολογιστών,

ο κάτωθι υπογεγραμμένος Δημόπουλος Δημοσθένης του Κωνσταντίνου, με αριθμό μητρώου 71344464, φοιτητής του Πανεπιστημίου Δυτικής Αττικής της Σχολής Μηχανικών του Τμήματος Μηχανικών Πληροφορικής και Υπολογιστών,

δηλώνουμε υπεύθυνα ότι:

«Είμαστε συγγραφείς αυτής της πτυχιακής/διπλωματικής εργασίας και ότι κάθε βοήθεια την οποία είχαμε για την προετοιμασία της είναι πλήρως αναγνωρισμένη και αναφέρεται στην εργασία. Επίσης οι όποιες πηγές από τις οποίες κάναμε χρήση δεδομένων, ιδεών ή λέξεων, είτε ακριβώς παραφρασμένες, αναφέρονται στο σύνολο του, με πλήρη αναφορά στους συγγραφείς, τον εκδοτικό οίκο ή το περιοδικό, συμπεριλαμβανομένων και των πηγών που ενδεχομένων χρησιμοποιήθηκαν από το διαδίκτυο. Επίσης, βεβαιώνουμε ότι αυτή η εργασία έχει συγγραφεί από εμάς αποκλειστικά και αποτελεί προϊόν πνευματικής ιδιοκτησίας τόσο δικής μας, όσο και του Ιδρύματος.

Παράβαση της ανωτέρω ακαδημαϊκής μας ευθύνης αποτελεί ουσιώδη λόγο για την ανάκληση των πτυχίων μας.»

Μαστρογιάννης Δημήτριος **Δημόπουλος Δημοσθένης**

Περιεχόμενα

Λέξεις κλειδιά

Παρακάτω παρατίθενται οι λέξεις κλειδιά της διπλωματικής εργασίας με στόχο την ηλεκτρονική βιβλιογραφική αρχειοθέτηση της.

- VHDL
- Verilog
- FPGA
- PLD
- CPLD
- ROM
- ModelSim
- Μοντελοποίηση δομής ή πυλών
- Μοντελοποίηση ροής δεδομένων
- Μοντελοποίηση συμπεριφοράς
- Προσομοίωση
- Testbench
- Συνδυαστικά κυκλώματα
- Ακολουθιακά κυκλώματα
- Αριθμητικά κυκλώματα
- Συνεχόμενη αντιστοίχηση
- Διαδικαστική αντιστοίχιση
- Αθροιστής
- Πολυπλέκτης
- Αποκωδικοποιητής
- Μανδαλωτής
- Φλιπ Φλοπ (Flip Flop)
- Κωδικοποιητής προτεραιότητας
- Καταχωρητής
- Καταχωρητής Ολίσθησης
- Απαριθμητής
- Πολλαπλασιαστής
- Αρχιτεκτονική Array
- Αρχιτεκτονική Carry Save
- Αρχιτεκτονική Wallace Tree
- Αρχιτεκτονική Booth για προσημασμένους αριθμούς
- Αρχιτεκτονική Baugh Wooley για προσημασμένους αριθμούς

Ευχαριστίες

Με την περάτωση της παρούσης διπλωματικής εργασίας θα θέλαμε να ευχαριστήσουμε θερμά τον κ. Ευστάθιου Κωνσταντίνο, Καθηγητή του Πανεπιστημίου Δυτικής Αττικής, για την αμέριστη βοήθεια και καθοδήγηση του στην εκπόνηση αυτής της διπλωματικής εργασίας καθώς και όλα τα μέλη του Πανεπιστημίου Δυτικής Αττικής για τις γνώσεις και την βοήθεια που μας προσφέρανε κατά την διάρκεια των προπτυχιακών μας σπουδών.

Επίσης, θα θέλαμε να ευχαριστήσουμε την Αγγελική Θεοδώρα Τούση, η οποία πέρα από την στήριξη και την συμπαράσταση της, βοήθησε στην βελτίωση της συντακτικής και λεκτικής εικόνας του κειμένου.

Τέλος, θα θέλαμε να ευχαριστήσουμε τις οικογένειες και τους φίλους μας για την στήριξη και την συμπαράσταση τους.

Η παρούσα διπλωματική αφιερώνεται σε όλους εκείνους.

Εισαγωγή

Η βιομηχανία κατασκευής ολοκληρωμένων κυκλωμάτων παρουσιάζει εξαιρετικά μεγάλη άνθιση τα τελευταία χρόνια. Κάθε μέρα όλο και μεγαλύτεροι αριθμοί αντικειμένων, είτε αυτοκίνητων, είτε σπιτιών, είτε απλών οικιακών συσκευών, παράγονται με όλα και περισσότερα ηλεκτρονικά στο εσωτερικό τους. Αποτέλεσμα αυτού είναι οι καθημερινά αυξανόμενες απαιτήσεις για νέους τρόπους παραγωγής ολοκληρωμένων κυκλωμάτων και αποτελεσματικότερη χρήση χρόνου όπως και υλικού, ειδικά μετά τις τεράστιες ελλείψεις ημιαγωγών κατά την διάρκεια της πανδημίας του Covid19.

Τα τελευταία τριάντα χρόνια οι γλώσσες περιγραφής υλικού, ξεκινώντας με την VHDL, αποτελούν τον πιο αποτελεσματικό τρόπο για την περιγραφή και υλοποίηση κυκλωμάτων. Από τον πιο απλό πολυπλέκτη ως τον πιο περίπλοκο επεξεργαστή, οι γλώσσες περιγραφής υλικού βρίσκονται σε σχεδόν κάθε κύκλωμα που παράγεται. Όταν κάποιος παρατηρεί τις γλώσσες περιγραφής υλικού θα δει απλές γλώσσες προγραμματισμού καθώς έχουν πολλές ομοιότητες με τις τυπικές γλώσσες προγραμματισμού. Ωστόσο ενώ υπάρχουν οι ίδιες βασικές αρχές, οι γλώσσες περιγραφής υλικού αποτελούν ένα εργαλείο πολύ διαφορετικό από την μέση γλώσσα προγραμματισμού και αυτός είναι ο λόγος για τον οποίο έχουν την θέση τους στην αγορά σήμερα.

Η συγκεκριμένη διπλωματική εργασία έχει σκοπό, αρχικά να πραγματοποιήσει μια σύντομη ανάλυση της εξέλιξης των γλωσσών περιγραφής υλικού, του υλικού στο οποίο υλοποιούν κυκλώματα και θα παραθέσει μερικές πληροφορίες σχετικά με αυτές. Κατόπιν αυτού θα περιλαμβάνει υλοποιήσεις, προσομοιώσεις και έλεγχο ορθής λειτουργίας κυκλωμάτων συνδυαστικής και ακολουθιακής λογικής όπως πολυπλέκτες και φλιπ-φλοπ με τις γλώσσες VHDL και Verilog. Συνεχίζοντας, θα γίνει περιγραφή, προσομοίωση και έλεγχος ορθής λειτουργίας κυκλωμάτων που εκτελούν αριθμητικές πράξεις όπως αθροιστές και πολλαπλασιαστές με τις γλώσσες VHDL και Verilog. Τέλος θα γίνει ανάλυση και εξαγωγή συγκριτικών συμπερασμάτων όσον αφορά την ευχρηστία των δύο γλωσσών για τα εν λόγω κυκλώματα.

Περίληψη

Στον τομέα του σχεδιασμού και υλοποίησης ολοκληρωμένων κυκλωμάτων οι γλώσσες περιγραφής υλικού έχουν σημαντικό ρόλο ξεκινώντας από την δεκαετία του 80’ με την έλευση της VHDL. Τα επόμενα χρόνια καθώς εμφανίζονταν στην αγορά όλο και πιο προχωρημένες προγραμματιζόμενες λογικές συσκευές οι γλώσσες περιγραφής υλικού άρχισαν να ξεπερνούν σε διασημότητα τα σχηματικά διαγράμματα καθώς ήταν η πιο γρήγορη, φθηνή, φιλική προς τον χρήστη και αποτελεσματική λύση. Έπειτα, ξεκίνησε ανταγωνισμός ανάμεσα στις γλώσσες περιγραφής υλικού ως προς το ποια είναι η πιο εύχρηστη.

Λαμβάνοντας υπόψιν το παραπάνω, η παρούσα διπλωματική αποτελεί μια ερεύνα σχετικά με τους τρόπους περιγραφής κυκλωμάτων με τις γλώσσες VHDL και Verilog παράλληλα με μια σύγκριση των δύο γλωσσών ως προς την ευχρηστία τους. Ειδικότερα στα πλαίσια της παρούσας διπλωματικής μελετώνται και αναλύονται οι τεχνολογίες που οδήγησαν στην σημερινή κυριαρχία των γλωσσών περιγραφής υλικού στην αγορά κατασκευής ολοκληρωμένων κυκλωμάτων, μαζί με μια ανάλυση των δημοφιλέστερων γλωσσών. Κατόπιν γίνεται ανάλυση της διαδικασίας της συγγραφής κώδικα στις γλώσσες VHDL και Verilog, συμπεριλαμβάνοντας την δομή τους, τον τρόπο που διαχειρίζονται μεταβλητές, τον γενικό συντακτικό τους και μεθοδολογίες γραφής του κώδικα. Στην συνέχεια επεξηγείται η διαδικασία της προσομοίωσης με το λογισμικό ModelSim της Mentor Graphics και η συγγραφή κώδικα οδήγησης για την εν λόγω προσομοίωση. Έπειτα γίνεται περιγραφή και προσομοίωση ακολουθιακών και συνδυαστικών κυκλωμάτων με τις εν λόγω γλώσσες. Τα κυκλώματα αυτά θα περιγραφούν με ποικίλους τρόπους με σκοπό την καλύτερη ανάλυση των δύο γλωσσών μαζί με την κατανόηση τους. Συνεχίζοντας, θα περιγραφούν κυκλώματα που πραγματοποιούν αριθμητικές πράξεις όπως αθροιστές και πολλαπλασιαστές. Μετά από κάθε περιγραφή θα γίνεται και η προσομοίωση του κάθε κυκλώματος. Τέλος θα γίνουν συγκρίσεις μεταξύ των παραπάνω κυκλωμάτων ως προς την ευχρηστία των γλωσσών VHDL και Verilog.

Η μεθοδολογία που χρησιμοποιήθηκε για την συγγραφή της παρούσας διπλωματικής εργασίας περιελάμβανε έρευνα σε ελληνικά και ξενόγλωσσα βιβλία, άρθρα και σε πηγές διαδικτύου ειδικότερα ως προς την καλύτερη συγγραφή του κώδικα των δύο γλωσσών.

Summary

In the field of integrated circuit design and implementation, hardware description languages have played a significant role since the 80's with the advent of VHDL. In the following years, as more and more advanced programmable logic devices appeared on the market, hardware description languages began to surpass schematic diagrams in popularity as they were the fastest, cheapest, most user-friendly and most effective solution. Then came the competition between the hardware description languages as to which one was the most user-friendly.

In view of the above, the present dissertation is a research on how to describe circuits in VHDL and Verilog languages along with a comparison of the two languages in terms of their usability. In particular, in the context of this dissertation, the technologies that led to the current dominance of hardware description languages in the integrated circuit manufacturing market are studied and analyzed, along with an analysis of the most popular languages. The process of writing code in VHDL and Verilog languages is then analyzed, including their structure, how variables are managed, their general syntax, and code writing methodologies. The following explains the simulation process with Mentor Graphics ModelSim software and the writing of the driver code for this simulation. Then sequential and combinational circuits will be described and simulated with these languages. These circuits will be described in various ways in order to better analyze the two languages along with their understanding. Continuing, circuits that perform arithmetic operations such as adders and multipliers will be described. After each description, the simulation of each circuit will be analyzed. Finally, comparisons will be made between the above circuits in terms of usability of VHDL and Verilog languages.

The methodology used to write this dissertation included research in Greek and foreign language books, articles and internet resources in particular regarding the best writing of the code of the two languages.

1. Εισαγωγή στα προγράμματα σχεδίασης CAD

1.1 Συστήματα CAD

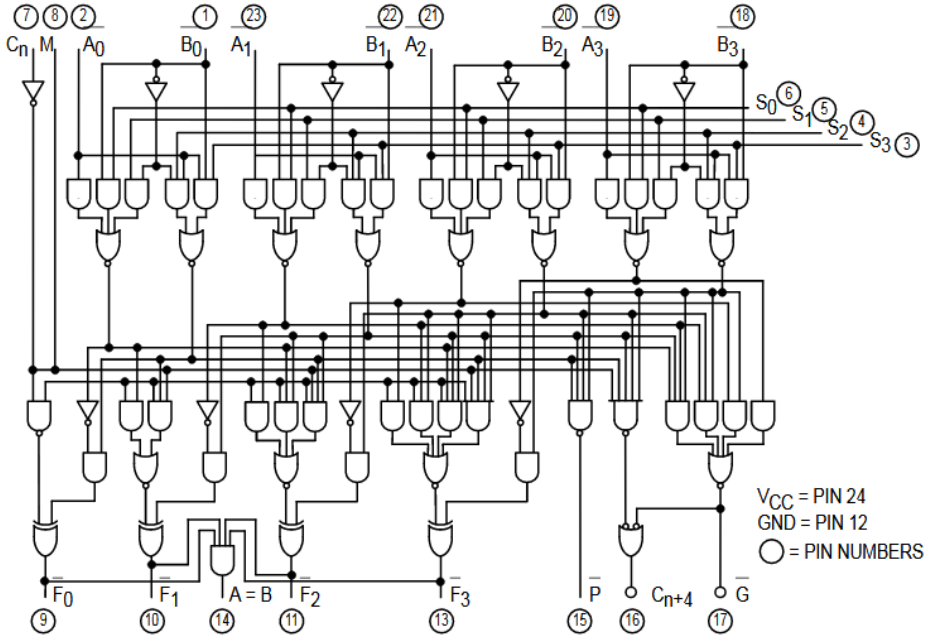
Ένα από τα πλέον βασικά προβλήματα των σύγχρονων υπολογιστικών συστημάτων, αποτελεί το γεγονός ότι είναι αδύνατο να σχεδιαστούν χειροκίνητα. Αντίθετα τα κυκλώματα του παρελθόντος μπορούσαν να υλοποιηθούν χωρίς προβλήματα, αλλά από την δεκαετία του ‘70 και έπειτα υπήρξε μια έκρηξη πολυπλοκότητας στα κυκλώματα, όπως υποδεικνύει ο νόμος του Moore. Για να γίνει αποτελεσματικότερη η διαδικασία υλοποίησης των κυκλωμάτων, δημιουργήθηκαν τα λεγόμενα συστήματα CAD (Computer Aided Design). Τα πακέτα τους τυπικά εμπεριέχουν εργαλεία, για να μπορούν να πραγματοποιήσουν εισαγωγή σχεδίασης, σύνθεση της λογικής και βελτιστοποίηση, προσομοίωση και φυσικό σχεδιασμό.

1.1.1 Εισαγωγή Σχεδίασης

Αρχικά, ο σχεδιαστής δεδομένου ότι έχει κατανοήσει την λειτουργία του κυκλώματος, δημιουργεί μια γενική δομή για το κύκλωμα. Αυτό πραγματώνεται χειροκίνητα από τον σχεδιαστή, καθώς απαιτείται εμπειρία στην σχεδίαση. Η υπόλοιπη διαδικασία σχεδίασης ολοκληρώνεται με χρήση συστημάτων CAD. Το πρώτο βήμα του σχεδιασμού ενός κυκλώματος είναι η εισαγωγή σχεδίασης, η οποία αποτελείται από δύο μεθόδους. Αυτές είναι η χρήση σχηματικού διαγράμματος και η χρήση γλωσσών περιγραφής υλικού οι οποίες θα αναλυθούν παρακάτω.

1.1.2 Μέθοδοι σχεδιασμού με CAD - Σχηματικό

Στην μέθοδο δημιουργίας σχηματικού διαγράμματος, δημιουργείται ένα σχηματικό σχεδιάζοντας τις λογικές πύλες και τις μεταξύ τους συνδέσεις. Πιο συγκεκριμένα τα στοιχεία του κυκλώματος, όπως πύλες, αναπαριστώνται ως γραφικά σύμβολα και οι μεταξύ τους συνδέσεις ως γραμμές. Η δυνατότητα οπτικής απεικόνισης αποτελεί ένα από τα μεγαλύτερα θετικά της χρήσης σχηματικού διαγράμματος. Με αυτή τη μέθοδο γίνεται χρήση της ικανότητας του υπολογιστή για αναπαράσταση γραφικών και του ποντικιού. Ωστόσο, η εισαγωγή των λογικών συμβόλων απαιτεί την χρήση βιβλιοθηκών, που εμπεριέχουν λογικές πύλες. Επιπλέον σε βιβλιοθήκες, μπορούν να εισαχθούν και άλλα υποκυκλώματα που έχουν υλοποιηθεί στο παρελθόν. Μια τεχνική που συνηθίζεται είναι η ιεραρχική σχεδίαση (hierarchical design), όπου μικρότερα κυκλώματα εμπεριέχονται στην σχεδίαση μεγαλύτερων κυκλωμάτων. Η χρήση σχηματικού είναι πολύ απλή στη χρήση, αλλά γίνεται δύσχρηστη σε μεγάλα κυκλώματα. Μολαταύτα, η καλύτερη μέθοδος εισαγωγής σχεδίασης, για μεγάλα κυκλώματα είναι με χρήση γλωσσών περιγραφής υλικού.



Σχήμα 1.1 Σχηματικό διάγραμμα μιας ALU 74181 μήκους τεσσάρων bit

1.1.3 Μέθοδοι σχεδιασμού με CAD – Γλώσσες περιγραφής υλικού

Οι γλώσσες περιγραφής υλικού (Hardware Description Language, HDL), είναι παρόμοιες με τη μέση γλώσσα προγραμματισμού. Ωστόσο μοναδική διαφορά έγκειται στις διεργασίες που πραγματοποιούν, δηλαδή η υλοποίηση ενός προγράμματος ή η υλοποίηση ενός κυκλώματος. Εντούτοις, υπάρχουν πολλές και ποικίλες γλώσσες περιγραφής υλικού, όμως στο πλαίσιο αυτής της διπλωματικής θα πραγματοποιηθεί εμβάθυνση στις VhsicHDL (Very High Speed Intgrated Circuit) και Verilog. Οι τελευταίες είναι οι πιο διαδεδομένες γλώσσες περιγραφής υλικού, και επίσημα πρότυπα (Standard) του Ινστιτούτου Ηλεκτρολόγων και Ηλεκτρονικών Μηχανικών (Institute of Electrical and Electronic Engineers, IEEE). Το ινστιτούτο IEEE αποτελεί έναν διεθνή οργανισμό και προωθεί πρότυπα με σκοπό το όφελος της κοινωνίας. Οι γλώσσες περιγραφής υλικού έχουν διάφορα θετικά, υπέρ των σχηματικών. Αναλυτικότερα συμβάλλουν στην φορητότητα του σχεδιασμού, καθώς μπορούν να υλοποιηθούν σε διαφορετικούς τύπους υλικού, χωρίς να χρειαστούν αλλαγές σε προδιαγραφές. Επίσης τα πρότυπα του IEEE είναι ανεξάρτητα από την τεχνολογία και υποστηρίζονται από τις περισσότερες εταιρίες παραγωγής ψηφιακών συστημάτων. Η εισαγωγή σχεδίασης ενός κυκλώματος με χρήση γλώσσας περιγραφής υλικού γίνεται, γράφοντας τον πηγαίο κώδικα της αντίστοιχης γλώσσας. Τα σήματα του κυκλώματος αναπαριστώνται ως μεταβλητές και οι λογικές εκφράσεις ως εκχώρηση τιμών σε αυτές. Επιπροσθέτως, ο κώδικας είναι σε απλό κείμενο με αποτέλεσμα να είναι εύκολος στην ανάγνωση. Αυτό προάγει την ευκολότερη διαδικασία σχολιασμού, σχετικά με την λειτουργία του κυκλώματος και την δημιουργία βιβλιογραφίας. Τα παραπάνω σε συνδυασμό με την ευρεία χρήση των γλωσσών περιγραφής υλικού, ειδικά των δυο προτύπων του IEEE, επιτρέπουν την γρηγορότερη ανάπτυξη νέων προϊόντων. Με παρόμοιο τρόπο, όπως στην χρήση

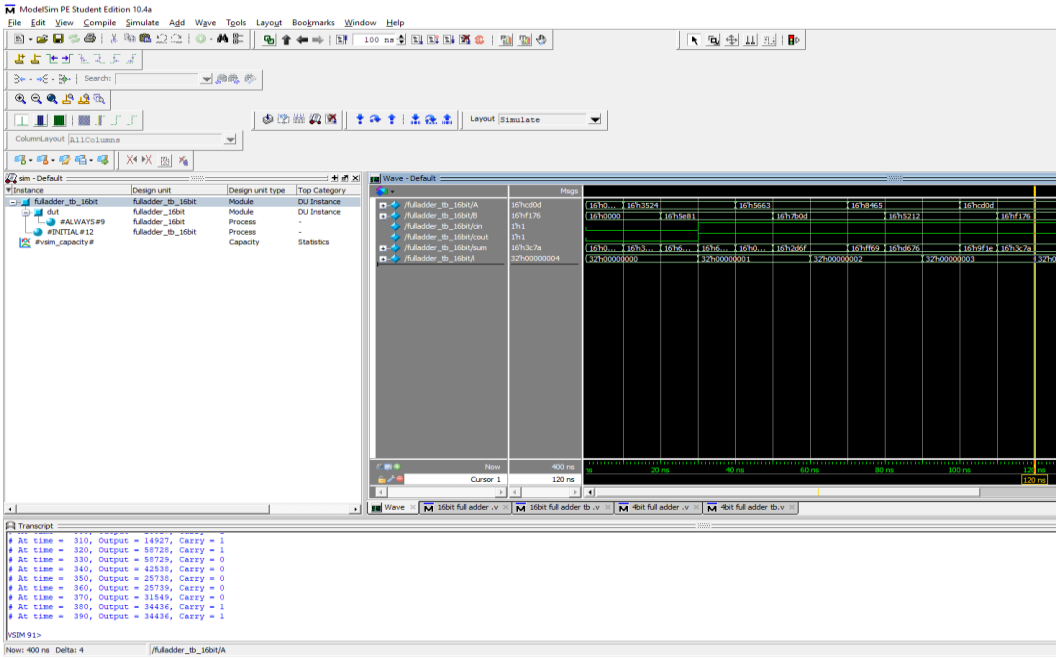
σχηματικού και οι γλώσσες περιγραφής υλικού μπορούν να γραφούν για να υλοποιούν την τεχνική της ιεραρχικής σχεδίασης. Έτσι, μεγάλα και μικρά κυκλώματα μπορούν εύκολα να υλοποιηθούν αποτελεσματικά, με την χρήση γλωσσών περιγραφής υλικού.

1.1.4 Σύνθεση και βελτιστοποίηση

Η σύνθεση είναι η διαδικασία υλοποίησης του λογικού κυκλώματος από μια αρχική περιγραφή, που μπορεί να έχει δοθεί είτε από σχηματικό, είτε από γλώσσα περιγραφής υλικού. Στην περίπτωση των γλωσσών περιγραφής υλικού η διαδικασία της μεταγλώττισης (compile) του κώδικα, είναι τμήμα της σύνθεσης που έχει ως αποτέλεσμα την περιγραφή του κυκλώματος με την χρήση λογικών εκφράσεων. Ωστόσο, είναι πολύ πιθανό ο σχεδιασμός του κυκλώματος, όπως έχει γίνει από τον σχεδιαστή να μην είναι ο βέλτιστος, το οποίο γίνεται πιο ευδιάκριτο σε μεγαλύτερα κυκλώματα. Επιπλέον ένας ρόλος της σύνθεσης, είναι να τροποποιεί το κύκλωμα και να δημιουργεί ένα ίδιο σε λειτουργικότητα, αλλά ανώτερο κύκλωμα. Το παραπάνω γίνεται τροποποιώντας την λίστα δικτύου, η όποια δημιουργείται κατά την φάση της λίστας δικτύου. Βέβαια, η λέξη «ανώτερο» είναι πολύ σχετική, αφού μπορεί να σημαίνει ελαχιστοποίηση του κόστους ή υλοποίηση σε πολύ συγκεκριμένο υλικό ή συνθήκες· επειδή εξαρτάται από τις απαιτήσεις του έργου, τις ανάγκες της υλοποίησης, το περιβάλλον και άλλα. Τέλος, η διαδικασία της σύνθεσης περιέχει την χαρτογράφηση τεχνολογίας, η οποία καθορίζει τον τρόπο με τον οποίο κάθε στοιχείο στη λίστα δικτύου, μπορεί να υλοποιηθεί με δεδομένους τους πόρους που απαιτούνται στο ολοκληρωμένο κύκλωμα.

1.1.5 Προσομοίωση

Η προσομοίωση είναι ένα πολύ σημαντικό εργαλείο των CAD, το οποίο επιτρέπει τον έλεγχο της λειτουργικότητας των κυκλωμάτων. Αυτό μπορεί να συμβεί χωρίς να δαπανάται χρόνος και χρήμα στον έλεγχο. Σε περίπτωση που υπάρχουν σφάλματα στον κώδικα, όπως μη ορισμένα σήματα, παρενθέσεις που λείπουν, και λάθος λέξεις-κλειδιά, αντιλαμβάνονται στη φάση της λίστας δικτύου. Ωστόσο αν δεν υπάρχουν σφάλματα, τότε παράγεται η λίστα δικτύου που χρησιμοποιεί λογικές εκφράσεις, για τη λειτουργία του κυκλώματος. Έτσι η προσομοίωση λειτουργεί με αυτές τις λογικές εκφράσεις, όπου μπορεί να περιέχονται αθροιστές, flip-flops, ακόμα και μηχανές πεπερασμένης κατάστασης. Ειδικότερα θεωρείται ότι υπάρχουν τέλειες πύλες, από τις οποίες τα σήματα περνούν ακαριαία από το εσωτερικό του κυκλώματος. Κατά τη διάρκεια της προσομοίωσης, ο χρήστης παρέχει ό,τι τιμές χρειάζεται, για να πραγματοποιηθεί ο έλεγχος και ο προσομοιωτής παράγει αποτελέσματα βάσει των λογικών εκφράσεων που του έχουν δοθεί νωρίτερα. Τα αποτελέσματα δίνονται συνήθως, σε μορφή ενός διαγράμματος χρόνου, με τον χρήστη να εξετάζει αν το κύκλωμα λειτούργησε σωστά.



Σχήμα 1.2 Προσομοίωση αθροιστή 16bit στο λογισμικό Modelsim

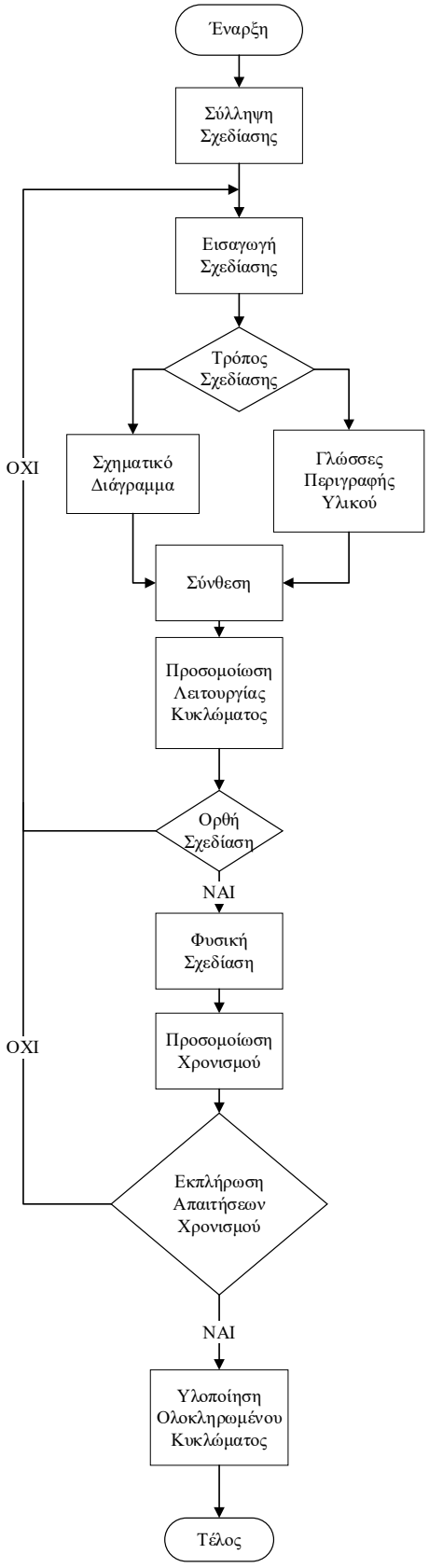
1.1.6 Φυσικός σχεδιασμός

Μετά τη σύνθεση ακολουθεί το βήμα του φυσικού σχεδιασμού. Συγκεκριμένα εμπεριέχονται αρκετές διαφορετικές τεχνολογίες, όπως λογικές πύλες NMOS και CMOS, συστήματα αρνητικής λογικής, προγραμματιζόμενες λογικές συσκευές (PLD) και άλλα. Τα εργαλεία του φυσικού σχεδιασμού, ενός κυκλώματος αποτυπώνουν ένα κύκλωμα με μορφή λογικών εκφράσεων, σε μια υλοποίηση με δεδομένους τους διαθέσιμους πόρους για το συγκεκριμένο ολοκληρωμένο κύκλωμα. Ταυτόχρονα καθορίζεται η τοποθέτηση συγκεκριμένων λογικών στοιχείων. Τα στοιχεία αυτά δεν αποτελούνται απαραίτητα από κάποια προ-υπάρχουσα πύλη. Παράλληλα με το παραπάνω καθορίζονται και οι μεταξύ τους συνδέσεις καλωδίων, για την υλοποίηση του κυκλώματος.

1.1.7 Καθυστέρηση διάδοσης – Προσομοίωση Χρονισμού

Όπως προαναφέρθηκε, κατά τη διαδικασία της προσομοίωσης δε λαμβάνονται υπόψιν τυχόν καθυστερήσεις διάδοσης, μεταξύ των μερών του κυκλώματος, και κατ' επέκταση καθυστέρηση της εξόδου, πράγμα αδύνατο υπό φυσιολογικές συνθήκες. Αυτό ονομάζεται καθυστέρηση διάδοσης και αποτελείται από δύο είδη διαφορετικής καθυστέρησης, τα οποία μπορούν να επηρεάσουν δραματικά την ταχύτητα μιας πράξης. Το πρώτο είναι η καθυστέρηση, που προκαλείται στην επεξεργασία του σήματος από κάθε λογικό στοιχείο του κυκλώματος, όταν αλλάζουν οι είσοδοι. Το δεύτερο είναι η μεταφορά του σήματος μέσω των καλωδίων που συνδέουν τα λογικά στοιχεία μεταξύ τους. Συνεπώς ένας προσομοιωτής χρονισμού, αξιολογεί τις καθυστερήσεις αυτές στο κύκλωμα και αποφασίζει αν το κύκλωμα πληροί τις προδιαγραφές χρονισμού για τον σχεδιασμό. Έτσι αν ο χρόνος που απαιτείται δεν

πληροί τις προδιαγραφές, ο σχεδιαστής θα πρέπει, είτε να βελτιστοποιήσει το σχέδιο στο στάδιο της σύνθεσης είτε να βελτιώσει, είτε αλλάξει το αρχικό.



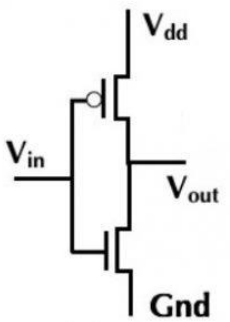
Σχήμα 1.3 Διάγραμμα ροής διαδικασίας σχεδίασης ολοκληρωμένων κυκλωμάτων

1.2 Μέθοδοι υλοποίησης κυκλωμάτων

Οι γλώσσες περιγραφής υλικού παρά την λειτουργικότητα τους, αποτελούνται απλώς από κώδικα. Ο εν λόγω κώδικας απαιτεί κάποιο μέσο για να είναι δυνατό να υλοποιήσει το κύκλωμα το οποίο περιγράφει. Υπάρχουν διάφοροι μέθοδοι υλοποίησης κυκλωμάτων. Οι τρεις αυτοί μέθοδοι είναι, τα πλήρως προσαρμοσμένα ολοκληρωμένα κυκλώματα (Full Custom Integrated Circuit), γνωστά και ως VLSI (Very Large Scale Integration), τα Semi-custom ASIC (Application-Specific Integrated Circuit) και τα PLD (Programmable Logic Device). Οι παραπάνω μέθοδοι υλοποιούν ολοκληρωμένα κυκλώματα, με βάση πληθώρας τεχνολογιών. Η πιο διαδεδομένη εκ των οποίων είναι η CMOS (Complementary Metal Oxide Semiconductor).

1.2.1 CMOS (Complementary Metal Oxide Semiconductor)

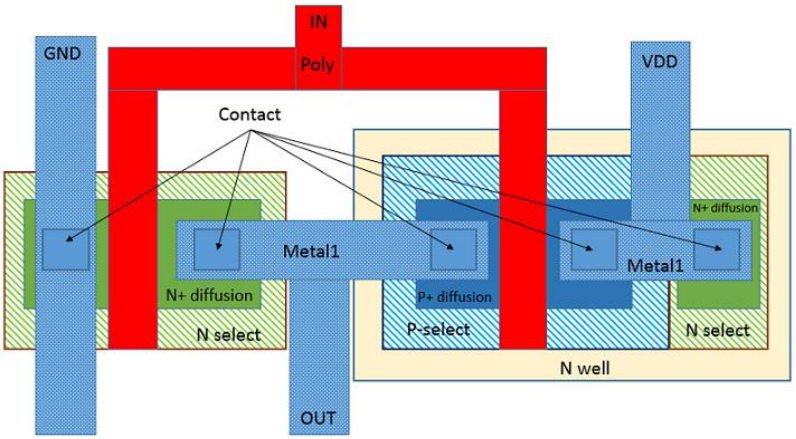
Η τεχνολογία CMOS, αποτελεί τον πιο διαδεδομένο τύπο διαδικασίας κατασκευής ενσωματωμένων κυκλωμάτων. Τα πρώτα τρανζίστορ με βάση την τεχνολογία αυτή ξεκίνησαν παραγωγή κατά την διάρκεια της δεκαετίας του ΄60. Συνδυάζει ζευγάρια ημιαγωγών τύπου PMOS και NMOS, με σκοπό την υλοποίηση λογικών συναρτήσεων. Βάσει της τεχνολογίας CMOS, τα «άνω» τμήματα των λογικών συναρτήσεων που υλοποιούνται είναι τύπου P και τα «κάτω» είναι τύπου N. Οι ημιαγωγοί αυτοί αποτελούνται από διάφορα επίπεδα. Τα κατώτερα επίπεδα συνθέτουν τα τρανζίστορ, τα ενδιάμεσα λογικές πύλες και τα ανώτερα συνδέουν τις πύλες αυτές με καλώδια.



Σχήμα 1.4 Αντιστροφέας Τύπου CMOS

1.2.2 Full Custom / VLSI

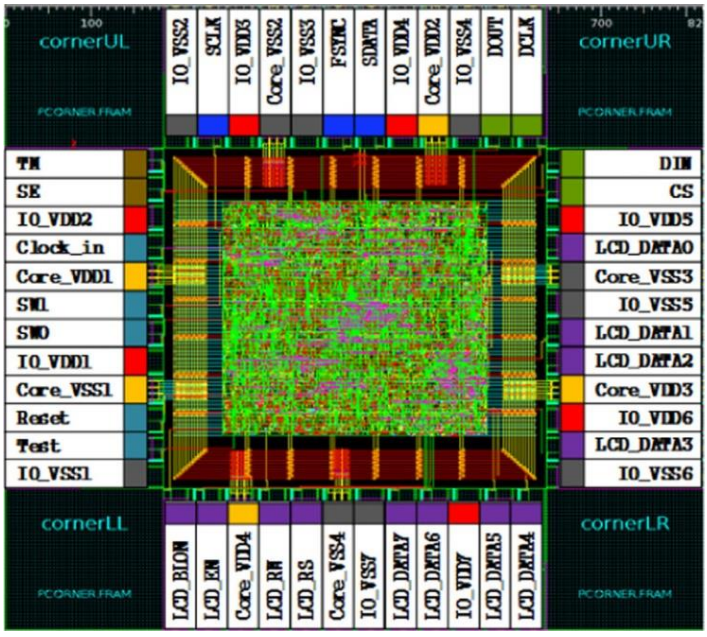
Η πλήρως προσαρμοσμένη τεχνολογία ολοκληρωμένων κυκλωμάτων, επιτρέπει την βελτίωση όλων των επιπέδων του ενσωματωμένου συστήματος. Οι βελτιστοποιήσεις περιέχουν αλλαγές, όπως δρομολόγηση καλωδίων, αλλαγή μεγέθους τρανζίστορ και τοποθέτηση τρανζίστορ με σκοπό, την μείωση μηκών διασύνδεσης. Μετά το πέρας της διαδικασίας, είναι έτοιμες οι μάσκες και αποστέλλονται οι προδιαγραφές σε εργοστάσιο, όπου ξεκινάει η παραγωγή. Ο σχεδιασμός έχει πολύ υψηλό κόστος και είναι χρονοβόρος αλλά το ολοκληρωμένο που παράγεται έχει χαμηλή κατανάλωση και εξαιρετική απόδοση. Η τεχνολογία VLSI χρησιμοποιείται σε εφαρμογές υψηλού φόρτου ή εξαιρετικών απαιτήσεων.



Σχήμα 1.5 Μάσκα VLSI

1.2.3 Semi-Custom ASIC

Στην τεχνολογία ASIC τα κατώτερα επίπεδα, δηλαδή τρανζίστορ και πύλες, είναι ήδη κατασκευασμένα και πρέπει να ολοκληρωθεί η κατασκευή των ανώτερων επιπέδων. Έτσι το υπόλοιπο τμήμα της μάσκας χρησιμοποιείται για την σύνδεση μεταξύ τρανζίστορ και πυλών με σκοπό να ολοκληρωθεί η υλοποίηση. Σε μια δεδομένη τεχνολογία, οι μάσκες των κελίων λογικού επιπέδου είναι ήδη σχεδιασμένες, κυρίως με το χέρι. Με βάση αυτό, το τμήμα της μάσκας που δεν χρησιμοποιείται, θα τεθεί προς την οργάνωση των επιμέρους τμημάτων σε ολοκληρωμένες μάσκες και έπειτα στην ένωση των κελιών. Αυτό έχει σαν αποτέλεσμα η τεχνολογία ASIC, να είναι η επικρατέστερη τεχνολογία παραγωγής ολοκληρωμένων κυκλωμάτων. Πιο συγκεκριμένα παρέχει καλή απόδοση, με πολύ μικρότερο κόστος εξέλιξης σε σχέση με τα VLSI.



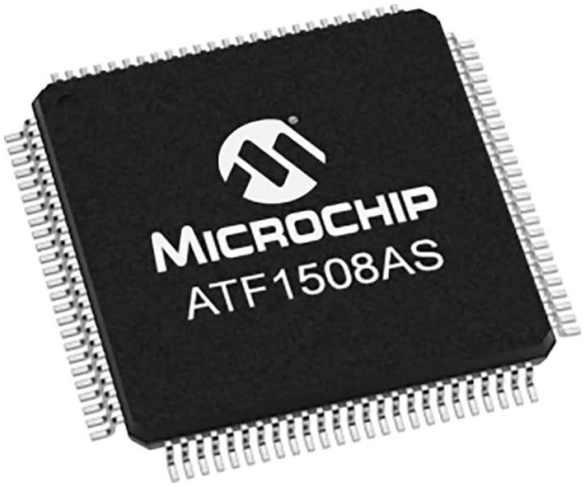
Σχήμα 1.6 Σχέδιο δημιουργίας ενός τσιπ ASIC

1.2.4 PLD (Programmable Logic Device)

Η τεχνολογία PLD, φέρει όλα τα επίπεδα της δημιουργημένα και δίνει την δυνατότητα στην απευθείας αγορά του ολοκληρωμένου. Σε αυτή την περίπτωση ένα προγραμματιζόμενο κύκλωμα υλοποιείται από τα επίπεδα. Επιπροσθέτως τα PLD, έχουν πολύ χαμηλό κόστος ανάπτυξης και σχεδόν άμεση διαθεσιμότητα. Σε σύγκριση με τα ASIC, απαιτούν μεγαλύτερη ισχύ, έχουν υψηλότερο κόστος ανά μονάδα και μικρότερη ταχύτητα. Ωστόσο παρέχουν καλή απόδοση και είναι ιδανικά για μεγάλη παραγωγή. Ο προγραμματισμός γίνεται με τρόπο παρόμοιο με αυτόν της PROM (Programmable Read Only Memory) δηλαδή, μέσω λιωσίματος ασφαλειών ή με την χρήση προγραμματιζόμενων διακοπτών. Έτσι διεξάγεται η δημιουργία ή η καταστροφή συνδέσεων των καλωδίων μεταξύ των πυλών και επιτελείται το τελικό κύκλωμα. Τα παραπάνω απαρτίζονται από τρεις κατηγορίες, τα PLA (Programmable Logic Array), τα PAL (Programmable Array Logic) και τα FPGA (Field Programmable Gate Array).

1.2.5 PLA (Programmable Logic Array) και PAL (Programmable Array Logic)

Τα PLA όντας η παλαιότερη κατηγορία PLD, άρχισαν να αναπτύσσονται στις αρχές της δεκαετίας του 70’. Τα πρώτα είναι βασισμένα στην τεχνολογία των PROM, που προϋπήρχαν στην αγορά εκείνη την εποχή. Τα ολοκληρωμένα κυκλώματα (Programmable Logic Array) συγκροτούνται από έναν προγραμματιζόμενο πίνακα πυλών AND και έναν προγραμματιζόμενο πίνακα πυλών OR. Σύμφωνα με τα βήματα των PLA, κατασκευάστηκαν τα PAL στα τέλη της δεκαετίας του 70’ με παρόμοια τεχνολογία. Ωστόσο, αντιθετικά με τα PLA χρησιμοποιεί μόνο έναν προγραμματιζόμενο πίνακα. Αποτέλεσμα αυτού ήταν μικρότερα, ταχύτερα και φθηνότερα ολοκληρωμένα. Έπειτα μιμώντας την τεχνολογία των EPROM (Erasable Programmable Read-Only Memory), η νέα γενιά PAL μπορούσε να σβήνεται και να ξανάπρογραμματίζεται. Αργότερα αναπτύχθηκαν τεχνολογίες CPLD (Complex Programmable Logic Device), με πολύ περισσότερες λογικές πύλες και EPLD (Erasable Programmable Logic Device), συνεχίζοντας την μίμηση τεχνολογιών των ROM.



Σχήμα 1.7 Το CPLD ATF1508AS-10AU100 της Microchip

1.2.6 Field Programmable Gate Array (FPGA)

Τα FPGA είναι μια άλλη κατηγορία PLD, η οποία εξελίχτηκε παράλληλα με τα PLA και έχει γνωρίσει ιδιαίτερη άνθιση τα τελευταία χρόνια. Συγκεκριμένα εφευρέθηκε το 1985 από την Xilinx βασισμένο, όπως τα PLD σε τεχνολογίες ανάπτυξης ROM. Αποτελεί το επόμενο λογικό βήμα από τα PLD, καθώς παρέχει προγραμματιζόμενες πύλες και προγραμματιζόμενες συνδέσεις μεταξύ των πυλών. Επίσης παρέχει γενικότερη συνδεσιμότητα μεταξύ μπλοκ, αντί των πινάκων λογικής, όπως γίνεται στα PLA και PAL.Επομένως αναλύοντας περαιτέρω ένα FPGA, έχει πολλά CLBs (Configurable Logic Blocks), τα οποία μπορούν να υλοποιήσουν μακράν πιο πολύπλοκες λογικές πράξεις. Το τελευταίο είναι δυνατό, διότι τα CLBs αποτελούνται από Look-Up Tables, πολυπλέκτες και Flip-Flops. Συμπερασματικά υπάρχει δυνατότητα να διενεργήσει ,πολύ πιο σύνθετα σχέδια. Ωστόσο παρότι βασίζεται στην τεχνολογία των CPLD, έχει κάποιες πολύ βασικές διαφορές.

1. Ένα FPGA πρέπει να φορτώσει δεδομένα από εξωτερική ROM, όποτε χρειάζεται κάποιο χρόνο για να τεθεί σε κατάσταση λειτουργίας.
2. Τα FPGA χρησιμοποιούν στατικές μνήμες RAM, με αποτέλεσμα να χάνεται ο προγραμματισμός αν χαθεί το ρεύμα.
3. Σε σχέση με τα CPLD, ένα FPGA είναι τεράστιο και σε φυσικό μέγεθος και ως προς το μέγεθος της πολυπλοκότητας της λογικής του. Επίσης, παρά τα βοηθητικά εργαλεία από παραγωγούς, ο σωστός χρονισμός σε ένα FPGA είναι αρκετά δύσκολος.
4. Έχουν σχετικά υψηλή κατανάλωση και σε περιπτώσεις που δεν απαιτείται πολύ ενέργεια.
5. Ενώ ένα CPLD μπορεί να είναι φθηνότερο για μικρά κυκλώματα, ένα FPGA είναι πιο αποτελεσματικό, αλλά συνήθως πιο ακριβό.
6. Με τα CLB που έχει ένα FPGA υπάρχει δυνατότητα σχεδιασμού πολύ πιο πολύπλοκων κυκλωμάτων. Επομένως τα προηγούμενα ευθύνονται, για ένα μεγάλο τμήμα της διασημότητας τους, με τους σχεδιαστές.
7. Ένα FPGA έχει ποικίλα συστήματα ενσωματωμένα, όπως μνήμη RAM, τα οποία προσφέρουν τρομερή ευλυγισία στο σχεδιασμό.
8. Στα FPGA υπάρχει η δυνατότητα, να αλλάξει το κύκλωμα που υλοποιείται και στη περίπτωση που βρίσκεται σε λειτουργία.

Σχήμα 1.8 Το Xilinx XC2064, το πρώτο FPGA που δημιουργήθηκε

1.3 Γλώσσες περιγραφής υλικού

Ο προγραμματισμός των FPGA και των υπόλοιπων PLDs της αγοράς πραγματώνεται, είτε με γλώσσες περιγραφής υλικού, είτε με χρήση σχηματικού διαγράμματος. Παρακάτω αναλύονται μερικές από τις πιο γνωστές γλώσσες περιγραφής υλικού.

1.3.1 VHDL

Η VHDL είναι αποτέλεσμα ανάγκης της βιομηχανίας, για την σχεδίαση κυκλωμάτων και η πρώτη γλώσσα περιγραφής υλικού. Σκοπός της, η ύπαρξη μιας κοινής γλώσσας για την περιγραφή περίπλοκων ψηφιακών κυκλωμάτων. Όπως αναφέρει η IEEE είναι αναγνώσιμη από μηχανή και από άνθρωπο και υποστηρίζει ανάπτυξη, σύνθεση, επαλήθευση και δοκιμή σχεδίων υλικού. Η παραπάνω θα δουλεύει σε κάθε προσομοιωτή και θα είναι ανεξάρτητη τεχνολογίας ή μεθοδολογίας σχεδίασης. Το αρχικό πρότυπο υιοθετήθηκε το 1987 από την IEEE ως IEEE 1076. Μια αναθεωρημένη εκδοχή υιοθετήθηκε το 1993 ως IEEE 1164. Επιπλέον αναθεωρήθηκε άλλες δυο φόρες, η πρώτη ήταν το 2000 και η δεύτερη το 2002. Επίσης το 2007 πραγματώθηκε τροποποίηση της έκδοσης του 2002 στην οποία προστέθηκε η διεπαφή VHDL και μερικές μικρές αλλαγές. Ως επίσημο πρότυπο της IEEE παρέχει έναν τρόπο καταγραφής κυκλωμάτων, σε μορφή κειμένου και δίνει την δυνατότητα μοντελοποίησης της συμπεριφοράς ψηφιακών κυκλωμάτων. Κατά συνέπεια μπορεί να χρησιμοποιηθεί ως είσοδος σε προγράμματα προσομοίωσης συμπεριφοράς ψηφιακών κυκλωμάτων.

1.3.2 Verilog

Η Verilog αποτελεί απόρροια των ραγδαίων προόδων στον τομέα των ενσωματωμένων κυκλωμάτων κατά μήκος της δεκαετίας του 1980. Η πρώτη σχεδιάστηκε από την Gateway Design Automation για ενδοεταιτερική χρήση ενώ μεγάλη επιρροή στην εξέλιξη της, ήταν η γλώσσα C. Στην συνέχεια η εταιρεία εξαγοράστηκε από την Cadence Design Systems. Τελικά η Verilog τοποθετήθηκε στον δημόσιο τομέα το 1990, αγαπήθηκε από τους σχεδιαστές και πέντε χρόνια αργότερα το 1995 υιοθετήθηκε, ως πρότυπο από την IEEE ως 1364-1995. Μια επιπλέον έκδοση με ορισμένες βελτιώσεις υιοθετήθηκε από την IEEE το 2001 ως 1364-2001. Έτσι η συγκεκριμένη μορφή της γλώσσας υποστηρίζεται, από την πλειοψηφία των λογισμικών ηλεκτρονικής σχεδίασης κυκλωμάτων. Ωστόσο η τελική μορφή της Verilog εγκολπώθηκε από την IEEE το 2005 ως 1364-2005, η οποία περιείχε ορισμένες μικρές σε έκταση διορθώσεις και βελτιώσεις. Ο κώδικας της θυμίζει ιδιαίτερα την δομή και τον τρόπο γραφής της C. Σχεδιάστηκε υπόψιν της απλοποίησης της γραφής κώδικα γλώσσας περιγραφής υλικού.

1.3.3 System Verilog

Η System Verilog είναι η πρώτη γλώσσα περιγραφής και επαλήθευσης υλικού, η οποία σχεδιάστηκε από την Accellera. Η παραπάνω στηρίζεται στην έκδοση της Verilog του 2005 και υιοθετήθηκε από την IEEE ως 1800-2005. Το 2009 έγινε ενσωμάτωση με την έκδοση του 2005 της Verilog, ως το πρότυπο 1800-2009. Η τελευταία έκδοση είναι του 2017, το πρότυπο της IEEE 1800-2017. Η System Verilog βασίζεται στην γλώσσα επαλήθευσης υλικού OpenVera, σε ότι είχε χτιστεί από τις VHDL και Verilog, αλλά περιέχει και στοιχεία από C++. Αποτελεσματικά, η επιρροή της C++ είναι η αντικειμενοστράφεια της γλώσσας, όπου αποδεσμεύεται από την παλιά δομή της Verilog με τον σειριακό κώδικα και χρησιμοποιεί κλάσεις στην δομή του κώδικα της. Η System Verilog προσθέτει νέες εντολές, είδη μεταβλητών και δυνατότητες στην Verilog. Κάποιες δυνατότητες από αυτές είναι τα packages (πακέτα) που έχουν ίδια λειτουργικότητα με αυτά της VHDL, ορισμένες διαφοροποιήσεις του διαδικαστικού μπλοκ «always» για περαιτέρω λειτουργικότητα και υποστήριξη πινάκων δύο διαστάσεων σε περιπτώσεις δηλώσεων τιμών εισόδου και εξόδου.

1.3.4 SystemC

Η SystemC είναι σχεδιασμένη και χτισμένη πάνω στην ίδια την C++, από την Accellera. Πιο συγκεκριμένα, είναι ένα σύνολο βιβλιοθηκών ανοιχτού λογισμικού, που επεκτείνουν την λειτουργικότητα της C++ και στον τομέα της σχεδίασης κυκλωμάτων. Ο σχεδιασμός της ξεκίνησε το 1999 και εντέλει υιοθετήθηκε από την IEEE το 2011 ως 1666-2011. Παρόλο που παραμένει ένα σύνολο βιβλιοθηκών, οι σχεδιαστές την κατατάσσουν ανάμεσα στις γλώσσες περιγραφής και επαλήθευσης υλικού. Η SystemC προσφέρει δυνατότητες υλοποίησης πολύπλοκων συστημάτων, τα οποία αποτελούν ένα υβρίδιο μεταξύ υλικού και λογισμικού. Παράδειγμα είναι τα SOCs (System on Chip/ σύστημα σε τσιπ) τα οποία είναι εξαιρετικά απαιτητικά ως προς την ανάπτυξη των αλγορίθμων και των αρχιτεκτονικών τους, απαιτήσεις στις οποίες η SystemC μπορεί να αντεπεξέλθει.

```
#include "systemc.h"

int sc_main (int argc, char* argv[]) {

cout <<"Hello World "<< endl;

return 0; }
```

Σχήμα 1.9 Κώδικας Hello World με χρήση SystemC

Έκτος από τις παραπάνω γλώσσες περιγραφής και επαλήθευσης υλικού, υπάρχουν πολλές, όπως η βασισμένη στην Python MyHDL, η Chisel η οποία είναι χτισμένη πάνω στην Scala και η OpenVera που χρησιμοποιήθηκε στην δημιουργία της System Verilog.

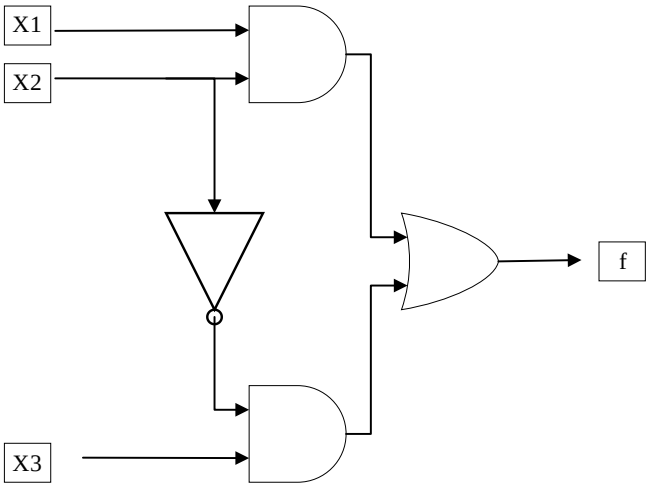
2. Περιγραφή βασικών, συνδυαστικών και ακολουθιακών κυκλωμάτων με VHDL και Verilog

2.1 Εισαγωγή στις VHDL και Verilog

Όπως προαναφέρθηκε, κατά την δεκαετία του ‘80 υπήρξαν μεγάλες πρόοδοι στον τομέα των ενσωματωμένων κυκλωμάτων. Αυτό είχε ως αποτέλεσμα τη δημιουργία και χρήση των πρώτων γλωσσών περιγραφής υλικού, για τη σχεδίαση ψηφιακών συστημάτων. Η σχεδίαση των ψηφιακών κυκλωμάτων πριν την έλευση των γλωσσών περιγραφής υλικού γινόταν με σχηματικά, τα οποία ήταν χρονοβόρα και άβολα στη χρήση τους. Σήμερα, θα ήταν να αδύνατο να γίνει ο σχεδιασμός τους με τέτοιο τρόπο. Οι γλώσσες περιγραφής υλικού δημιουργήθηκαν για να πραγματοποιείται πιο γρήγορα και αποτελεσματικά η διαδικασία. Αρχικά, δημιουργήθηκε το πρότυπο της IEEE (Institute of Electrical and Electronics Engineers), η VHDL. Έπειτα ακολούθησε το δεύτερο πρότυπο, η Verilog. Οι δύο γλώσσες παραμένουν από τις πιο διαδεδομένες γλώσσες περιγραφής υλικού (Hardware Description Languages, HDL). Το κεφαλαίο αυτό, θα αφιερωθεί στη μελέτη συνδυαστικών και ακολουθιακών κυκλωμάτων με τις γλώσσες περιγραφής υλικού VHDL και Verilog. Παράλληλα θα πραγματοποιηθούν και προσομοιώσεις των εν λόγω κυκλωμάτων με το πρόγραμμα προσομοίωσης ModelSim.

2.2 Δομή της VHDL

Η δομή της VHDL αποτελείται από δύο ενότητες, που δημιουργούν το σύνολο της οντότητας. Στην εκάστοτε σχεδίαση κυκλώματος σε VHDL, είναι απαραίτητο να δηλώνονται τα σήματα εισόδου αλλά και τα σήματα εξόδου της οντότητας. Αυτό επιτυγχάνεται με την εντολή «ENTITY», η οποία επιτρέπει στον σχεδιαστή την δήλωση όλων των αναγκαίων σημάτων εισόδου και εξόδου. Ο τρόπος χρήσης της εντολής βρίσκεται στο παράδειγμα του σχήματος 2.2. Ακολουθεί παράδειγμα κώδικα για την λογική παράσταση $f = x_1 x_2 + \bar{x}_2 x_3$ του σχήματος 2.1, που περιγράφεται στα σχήματα 2.3 και 2.4.



Σχήμα 2.1 Λογικό κύκλωμα που υλοποιεί την $f = x_1 x_2 + \bar{x}_2 x_3$

```
ENTITY example1 IS
  PORT (x1, x2, x3 :   IN   STD_LOGIC;
                  f : OUT STD_LOGIC);
END example1;
```

Σχήμα 2.2 Δήλωση οντότητας ENTITY.

Παρατήρηση

Τα σήματα εισόδου δηλώνονται ως IN ενώ τα σήματα εξόδου OUT. Επιπλέον, υπάρχει η κατάσταση INOUT, όπου ένα σήμα μπορεί να αποτελέσει είσοδο και έξοδο στην οντότητα. Ακόμα, υπάρχει και η κατάσταση BUFFER η οποία λειτουργεί όπως η OUT, με την διαφορά ότι έχει γίνει προσθήκη της ικανότητας να μπορεί να διαβαστεί από την οντότητα. Τέλος, είναι θεμιτό τα σήματα με ίδιο τύπο να δηλώνονται στην ίδια γραμμή κώδικα.

Το δεύτερο μέρος μιας οντότητας στην VHDL είναι η αρχιτεκτονική της, στην οποία αποτυπώνονται όλες οι λεπτομέρειες του κυκλώματος. Η εντολή για την δημιουργία της αρχιτεκτονικής είναι η «ARCHITECTURE». Αναλυτικότερα, το τμήμα της αρχιτεκτονικής αποτελείται από δύο μέρη εκ των οποίων το πρώτο είναι η περιοχή δήλωσης. Στην περιοχή δήλωσης δηλώνονται σήματα, σταθερές, συνιστώσες και καθορίζονται χαρακτηριστικά της οντότητας. Το δεύτερο σκέλος της αρχιτεκτονικής είναι το «σώμα» της, στο οποίο καθορίζεται η λειτουργικότητα του κυκλώματος. Η λειτουργικότητα μπορεί να περιγραφεί με λογικές συναρτήσεις ή με απλές αριθμητικές πράξεις καθώς επίσης και με χρήση ακολουθιακών εντολών. Στο σχήμα 2.3 αποτυπώνεται η συνολική ενότητα της αρχιτεκτονικής.

```
ARCHITECTURE LogicFunc OF example1 IS
   BEGIN
      f <=(x1 AND x2) OR ((NOT x2) AND x3);
END LogicFunc;
```

Σχήμα 2.3 Αρχιτεκτονική (ARCHITECTURE) της Οντότητας.

Παρατήρηση

Ο διαχωρισμός της περιοχής δήλωσης με το «σώμα» της αρχιτεκτονικής, γίνεται με την χρήση του BEGIN. Επιπλέον χρήζει δήλωση ονόματος, όπως στην δήλωση οντότητας.

Ένα επιπρόσθετο μέρος της οντότητας είναι οι βιβλιοθήκες, που βοηθούν στην υλοποίηση των κυκλωμάτων. Οι βιβλιοθήκες που συνήθως χρησιμοποιούνται είναι δημιουργία της IEEE, με επεκτάσεις που έχει δημιουργήσει η Synopsys.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
```

Άξιο αναφοράς αποτελεί ότι με τον ίδιο τρόπο κλήσης, ο σχεδιαστής μπορεί να επικαλεστεί κυκλώματα που έχει αποθηκεύσει ως πακέτα, ώστε να χρησιμοποιηθούν στο κύκλωμα του. Τα πακέτα μπορεί να είναι προγράμματα, όπως ένας πλήρης αθροιστής ή ένας αποκωδικοποιητής. Επίσης τα παραπάνω μπορούν να χρησιμοποιηθούν σε άλλες οντότητες, με την απλή συμπερίληψή τους ως βιβλιοθήκες. Τα σχήματα 2.4, 2.5 και 2.6 δείχνουν την συγγραφή, την κλήση του πακέτου και την ενσωμάτωση τους στο «σώμα» της αρχιτεκτονικής αντίστοιχα.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;


PACKAGE full_add_package IS
        COMPONENT full_add
                PORT (x, y, cin : IN  STD_LOGIC;
               cout, s   : OUT STD_LOGIC);
      END COMPONENT;
END full_add_package;
```

Σχήμα 2.4 Συγγραφή πακέτου.

```
LIBRARY work;
USE work.full_add_package.all;
```

Σχήμα 2.5 Δήλωση πακέτου.

```
FA1:full_add PORT MAP(a2b0,a1b1,a0b2,c02,ap02);
```

Σχήμα 2.6 Χρήση του κυκλώματος του πακέτου.

2.3 Δομή της Verilog

Σε αντίθεση με την VHDL, η οποία έχει μια συγκεκριμένη δομή, η Verilog είναι «ενωμένη». Πιο συγκεκριμένα δεν υπάρχει κάποιος διαχωρισμός από τη δήλωση εισόδου/εξόδου με τη συμπεριφορά του κυκλώματος, όπως στην VHDL όπου, η οντότητα και η αρχιτεκτονική είναι δύο διακριτά τμήματα κώδικα. Επίσης, δεν υπάρχει η έννοια της βιβλιοθήκης στην Verilog, καθώς προστέθηκε πολύ αργότερα ως λειτουργικότητα στην System Verilog.

Το πρώτο βήμα στην υλοποίηση του κώδικα Verilog, είναι η δήλωση των σημάτων και μετά ο καθορισμός, αν αυτά είναι εισόδου και εξόδου. Η δήλωση γίνεται, είτε κατευθείαν στον καθορισμό της ενότητας (module) του κυκλώματος, είτε αμέσως εντός του κώδικα. Αυτό είναι το αντίστοιχο της οντότητας της VHDL.

```
module name (signals);          module name (input signals, output signals);
        input signals;
        output signals;

module example (x1, x2, x3, f);     module example (input x1, x2, x3, output f);
        input x1, x2, x3;
        output f;
```

Σχήμα 2.7 Παράδειγμα ορισμού μεταβλητών

Το όνομα που επιλέχθηκε είναι το example. Τα σήματα δηλώνονται στο «module» και αν είναι εισόδου ή εξόδου ορίζεται, είτε εντός της δήλωσης του module, είτε εκτός αμέσως μετά. Αν ένα σήμα είναι είσοδος ορίζεται «input», διαφορετικά ορίζεται «output». Σε αντίθεση με την VHDL, υπάρχουν μόνο δύο τύποι δεδομένων στην Verilog, τα nets και τα registers. Τα nets δεν μπορούν να αποθηκεύσουν μεταβλητές, αλλά χρησιμοποιούνται για την οδήγηση του κυκλώματος. Επιπλέον διακρίνονται σε πολλά διαφορετικά (π.χ. wire, tri, wor, trior, wand, triand, κ.λ.π.), με πιο συνηθισμένο το wire. Αντιθέτως, τα registers χρησιμοποιούνται για αποθήκευση, όπως κάποιος καταχωρητής και διακρίνονται σε (α) reg για περιγραφή λογικής, (β) integer για βρόγχους επαλήθευσης και υπολογισμούς, (γ) time για αποθήκευση χρόνου όταν χρησιμοποιείται κώδικας ελέγχου. Επίσης όλες οι είσοδοι σε κώδικα ελέγχου, πρέπει να ορίζονται ως «reg» για αρχικοποίηση. Όλοι οι τύποι μεταβλητών μπορούν να λάβουν τιμές 0, 1, X (αδιάφορο), Z (υψηλή αντίσταση) και η δήλωση τους μπορεί να γίνει μαζί ή ξεχωριστά, από τις δηλώσεις για είσοδο και έξοδο.

```
module name (signals);          module name (input wire signals, output reg signals);
        input wire signals;
        output reg signals;
```

Σχήμα 2.8 Δομή ορισμού μεταβλητών εισόδου και εξόδου

Κατόπιν, πρέπει να γίνει η υλοποίηση της αρχιτεκτονικής του κυκλώματος, όπου ορίζεται η λειτουργία του. Η υλοποίηση, έχει την δυνατότητα να πραγματοποιηθεί με

ποικίλους τρόπους, όπως περιγράφονται παρακάτω. Έτσι, για τους σκοπούς του παραδείγματος θα χρησιμοποιηθεί η εντολή «assign», στην οποία πραγματώνεται η χρήση συνεχόμενης ανάθεσης της λογικής παράστασης στην έξοδο. Η Verilog, όπως και η VHDL, υποστηρίζει τελεστές λογικών πράξεων (AND, OR, NOT, NAND, NOR, XOR, XNOR) με την αντίστοιχη προτεραιότητα στην NOT και κατάλληλη χρήση παρενθέσεων. Τέλος, η υλοποίηση σταματά με την εντολή «endmodule». Οι κώδικες των σχημάτων 2.7 και 2.9 υλοποιούν το κύκλωμα με λογική παράσταση

```
assign f = (x1 & x2) | (~x2 & x3);
endmodule
```

Σχήμα 2.9 Ορισμός λειτουργίας κυκλώματος και τέλος υλοποίησης

2.4 Μεθοδολογίες γραφής κώδικα

Η VHDL και η Verilog έχουν τρεις διαφορετικές διακριτές μεθοδολογίες, με τις οποίες μπορεί να γραφεί ο κώδικας. Παρακάτω γίνεται εφαρμογή των τριών αυτών μεθοδολογιών, για την υλοποίηση του κυκλώματος με την λογική παράσταση : $f = x_1 x_2 + \bar{x}_2 x_3$.

2.4.1 Μεθοδολογίες γραφής κώδικα VHDL

1. Η structural μεθοδολογία θεμελιώνεται στην μοντελοποίηση του κυκλώματος με πύλες (AND, OR, XOR, κλπ). Στη VHDL υπάρχει η δυνατότητα να πραγματοποιηθεί η υλοποίηση, είτε με την χρήση PACKAGES (πακέτων), είτε με την χρήση COMPONENTS (συνιστώσες). Παρακάτω ακολουθεί η υλοποίηση της structural μεθοδολογίας με χρήση πακέτων.

```
LIBRARY IEEE;

LIBRARY work;

USE IEEE.STD_LOGIC_1164.ALL;

USE work.ANDGATE_package.ALL;

USE work.ORGATE_package.ALL;



ENTITY example2 IS

  PORT (x1, x2, x3 :  IN   STD_LOGIC;

              f :  OUT  STD_LOGIC);

END example2;
```

```
ARCHITECTURE structural OF example2 IS

        SIGNAL z1,z2,Nx2: STD_LOGIC;
        BEGIN
        Nx2<=NOT(x2);
        AND1: ANDGATE PORT MAP(x1,x2,z1);
        AND2: ANDGATE PORT MAP(Nx2,x3,z2);
        OR0:  ORGATE  PORT MAP(z1,z2,f);
END structural;
```

Σχήμα 2.10 Περιγραφή του κυκλώματος $f = x_1 x_2 + \bar{x}_2 x_3$ με χρήση πυλών VHDL.

2. Η Dataflow μεθοδολογία μεταχειρίζεται την λογική έκφραση και την υλοποιεί με την χρήση λογικών τελεστών. Η dataflow μεθοδολογία εστιάζει στην ροή των δεδομένων από κάθε πύλη.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY example1 IS
  PORT (x1, x2, x3 : IN  STD_LOGIC;
              f : OUT STD_LOGIC);
END example1;

ARCHITECTURE dataflow OF example1 IS
   BEGIN
      f <=(x1 AND x2) OR ((NOT x2) AND x3);
END dataflow;
```

Σχήμα 2.11 Περιγραφή του κυκλώματος $f = x_1 x_2 + \bar{x}_2 x_3$ με χρήση ροής δεδομένων VHDL.

3. Η Behavioral μεθοδολογία βασίζεται στην περιγραφή συμπεριφοράς της οντότητας, με την υπηρεσία διαδικαστικών δηλώσεων (Sequential Statements). Πιο συγκεκριμένα η μεθοδολογία αυτή, παρομοιάζεται με προγραμματισμό κώδικα γλώσσας υψηλού επιπέδου, αφού χρησιμοποιεί κοινούς τρόπους στην σύνταξη και την χρήση. Πρόσθετα με τα υπόλοιπα, για να παραχθεί η οντότητα με την Behavioral μεθοδολογία, είναι αναγκαία η εκτέλεση της εντολής PROCESS.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;


ENTITY example3 IS
  PORT (x1, x2, x3 :  IN   STD_LOGIC;
            f :  OUT  STD_LOGIC);
END example3;


ARCHITECTURE Behavior OF example3 IS


   BEGIN
   PROCESS(x1,x2,x3)
        BEGIN
     IF(x1='0' AND x2='0' AND X3='1') THEN
                f<='1';
        ELSIF(x1='1' AND x2='0' AND X3='1') THEN
                f<='1';
        ELSIF(x1='1' AND x2='1' AND X3='0') THEN
                f<='1';
        ELSIF(x1='1' AND x2='1' AND X3='1') THEN
                f<='1';
```

```
        ELSE
                f<='0';
        END IF;
    END PROCESS;
END Behavior;
```

Σχήμα 2.12 Περιγραφή του κυκλώματος $f = x_1x_2 + \bar{x}_2x_3$ με χρήση διαδικαστικών δηλώσεων

2.4.2 Μεθοδολογίες γραφής κώδικα Verilog

1. Η μοντελοποίηση δομής ή πυλών (structural ή gate), όπου μοντελοποιείται το κύκλωμα με δεδομένες πύλες όπως OR, AND, αλλά και με την εισαγωγή περαιτέρω διαφορετικών λογικών στοιχείων ανάλογα με τις ανάγκες του κυκλώματος. Στη μοντελοποίηση δομής πραγματοποιούνται συνδέσεις μεταξύ τους για να συμπληρωθεί το κύκλωμα. Η μοντελοποίηση δομής αποτελεί το χαμηλότερο δυνατό επίπεδο αφαίρεσης.

```
module example1 (input x1, x2, x3, output f); //setting inputs and output
wire Sanda, Sandb, Snot; //setting wire values for gate level design
not (Snot, x2); // setting gate inputs and outputs
and (Sanda, x1,x2), (Sandb, x3, Snot);
or (f, Sanda, Sandb);
endmodule
```

Σχήμα 2.13 Περιγραφή του κυκλώματος $f = x_1x_2 + \bar{x}_2x_3$ με χρήση πυλών

2. Η μοντελοποίηση ροής δεδομένων (dataflow) όπου, χρησιμοποιούμε τη λογική παράσταση του κυκλώματος ως βάση για την υλοποίηση. Συνεπώς, υπάρχει μια ροή δεδομένων από είσοδο σε έξοδο, η οποία είναι δυνατή με τη χρήση συνεχόμενων δηλώσεων εκχώρησης (εντολή assign).

```
module example2 (input x1, x2, x3, output f); //setting inputs and output
assign f = (x1 & x2) | (~x2 & x3);
//assigning behavior to output as a logic function
endmodule
```

Σχήμα 2.14 Περιγραφή του κυκλώματος $f = x_1x_2 + \bar{x}_2x_3$ με χρήση ροής δεδομένων

3. Η μοντελοποίηση συμπεριφοράς του κυκλώματος γίνεται χρήση του πίνακα αληθείας, για να εξακριβωθεί η συμπεριφορά του. Στο επίπεδο συμπεριφοράς του κυκλώματος γίνεται χρήση διαδικαστικών δηλώσεων με λίστες

ευαισθησίας για τον έλεγχο της διαδικασίας (μπλοκ εντολών always και initial) και αποτελεί το υψηλότερο δυνατό επίπεδο αφαίρεσης.

```
module example2 (x1, x2, x3, f); //setting inputs and output
    input x1, x2, x3;
    output reg f; //setting reg data type for storing values
    always @(x1 or x2 or x3) // for every change in inputs the following happens
    begin
        if (x1==0 & x2==0 & x3==1)
            f = 1;
        else if (x1==1 & x2==0 & x3==1)
            f = 1;
          else if (x1==1 & x2==1 & x3==0)
            f = 1;
        else if (x1==1 & x2==1 & x3==1)
            f = 1;
        else
            f = 0;
    end
endmodule
```

Σχήμα 2.15 Περιγραφή του κυκλώματος $f = x_1x_2 + \bar{x}_2x_3$ με χρήση διαδικαστικών δηλώσεων

2.5 Περιγραφή σημάτων πολλών bit με χρήση VHDL

Η VHDL δίνει την δυνατότητα στον σχεδιαστή να μεταχειρίζεται σήματα πολλαπλών bit, που αφορούν τα σήματα εισόδου, εξόδου ή μεταβλητές. Στη δήλωση του ENTITY γίνεται η αντίστοιχη υλοποίηση, όπως περιγράφεται στο σχήμα 2.16.

```
ENTITY example IS

  PORT (x1, x2 :    IN   STD_LOGIC_VECTOR(4 DOWNTO 0);

                f :  OUT  STD_LOGIC_VECTOR(4 DOWNTO 0));

END example;
```

Σχήμα 2.16 Περιγραφή δηλώσεων πολλαπλών bit VHDL.

Υπάρχουν δύο τρόποι εισαγωγής τιμών στα σήματα πολλαπλών bit. Πιο συγκεκριμένα ο ένας αναφέρεται στην εκχώρηση της τιμής συνολικά, ενώ στον άλλον δίνεται η δυνατότητα να εκχωρηθεί κάθε bit ατομικά, όπως θα γινόταν αντίστοιχα με την χρήση array σε γλώσσα υψηλού επιπέδου. Στο σχήμα 2.17 είναι ορατή η υλοποίηση της εκχώρησης.

```
X1<=”0001”; --πρώτος τρόπος

X2(0)<=”1”; --δεύτερος τρόπος

X2(1)<=”0”;

X2(2)<=”1”;

X2(2)<=”0”;
```

Σχήμα 2.17 Επεξήγηση εκχώρησης μεταβλητών πολλαπλών bit VHDL.

2.6 Περιγραφή σημάτων πολλών bit με χρήση Verilog

Στα ψηφιακά συστήματα χρησιμοποιούνται και σήματα πολλών bit. Η Verilog κάνει την περιγραφή αυτή, μαζί με την περιγραφή εισόδου και εξόδου, χωρίς τη χρήση ξεχωριστής εντολής. Στο παρακάτω σχήμα (Σχήμα 2.18) τα A και B χωρίζονται σε 4 bit αντίστοιχα, δηλαδή A_0, A_1, A_2, A_3 και B_0, B_1, B_2, B_3, και μπορούμε να τους εκχωρήσουμε όποια τιμή τεσσάρων bit χρειάζεται.

```
module name (A, B);
input [3:0] A;
output [3:0] B;
A = 0;  B = 4’b1010;
```

```
module name (output [3:0] B, input [3:0] A);
    A = 0;
    B = 4’b1010;
```

Σχήμα 2.18 Περιγραφή δηλώσεων πολλαπλών bit

Ωστόσο, η εκχώρηση λειτουργεί διαφορετικά από ότι στην VHDL. Μπορεί να τεθεί μια μεταβλητή ως X = 0, αλλά ο πλήρης και σαφής τρόπος για την εκχώρηση μεταβλητών είναι X = 1’b0. Επομένως, ορίζεται πρώτα το μήκος της μεταβλητής (σε αυτή τη περίπτωση «1»), μετά το αριθμητικό σύστημα στο οποίο υπάγεται (σε αυτή τη περίπτωση δυαδικό) με επιλογές από δυαδικό (b), οκταδικό (o), δεκαδικό (d) και δεκαεξάδικο (h). Στο τέλος, γράφεται ο αριθμός που απαιτείται (στη συγκεκριμένη περίπτωση «0»). Παράλληλα, υπάρχουν επιλογές σε περίπτωση που χρειαστεί συμπλήρωμα με 2 χρησιμοποιώντας το «-», X και Z αν κάποιο τμήμα έχει υψηλή αντίσταση ή αν η τιμή δεν επηρεάζει την έξοδο, καθώς και χρήση “_” το οποίο αγνοείται. Παρακάτω παραθέτονται παραδείγματα (Σχήμα 2.19).

```
a = 1’b0; // a=0        b = 4’o12; // b = 1010          c = -4’b1; // c = 1111
a = 5’hf; // a=0000f   b= 8’bx01; // b=xxxxxx01       c = 3’bz; // c = zzz
```

Σχήμα 2.19 Επεξήγηση εκχώρησης μεταβλητών πολλαπλών bit

2.7 Εισαγωγή στις προσομοιώσεις

Στην εισαγωγή πραγματοποιήθηκε μια σύντομη επεξήγηση των λειτουργιών, που αποτελούν την χρήση συστημάτων CAD. Αρχικά κατά την σύνθεση, υλοποιείται η διεξαγωγή της μεταγλώττισης (compile) του κώδικα. Έπειτα, κατά την μεταγλώττιση

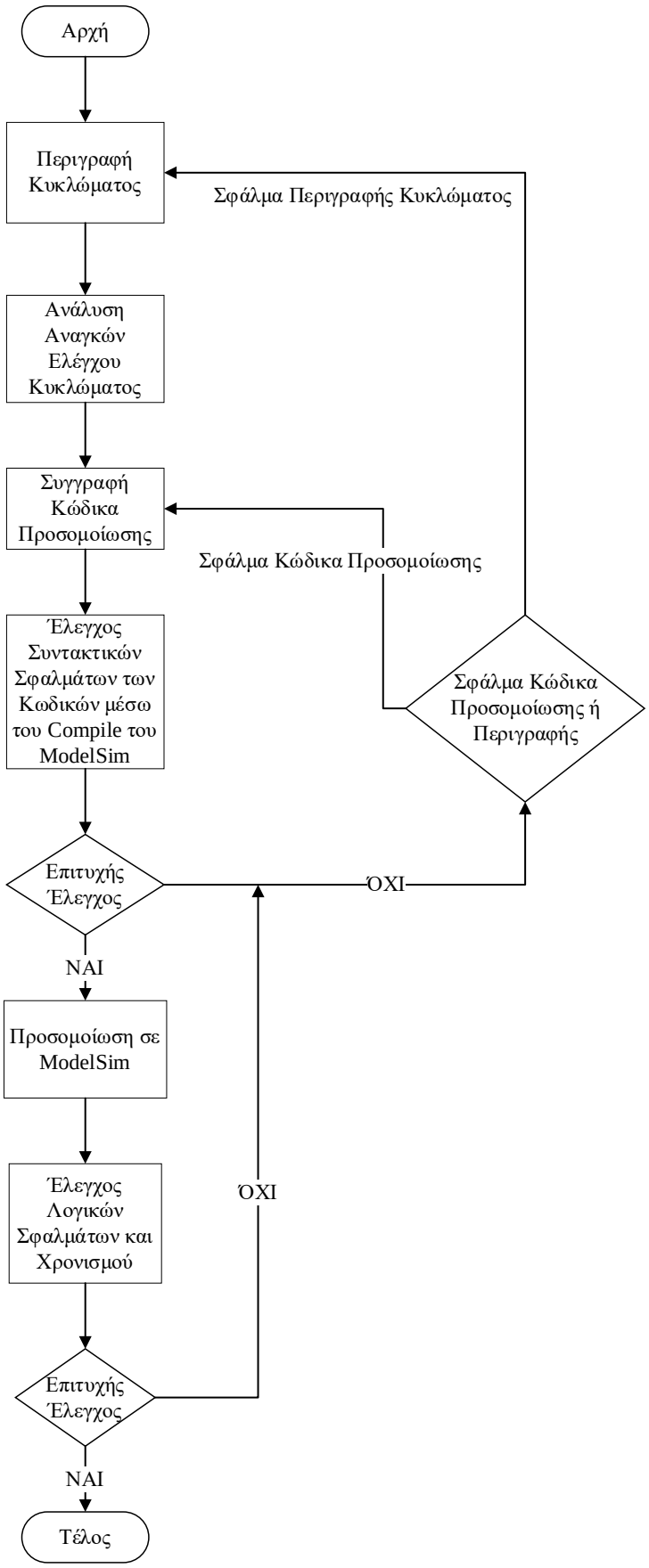
ο κώδικας της εκάστοτε γλώσσας περιγραφής υλικού, μετατρέπεται σε ένα σύνολο λογικών εκφράσεων. Το τελευταίο ελέγχεται ως προς την λειτουργικότητά του από λειτουργικούς προσομοιωτές. Μετέπειτα θα γίνει ανάλυση της δομής των κυκλωμάτων και του κώδικα προσομοίωσης (testbench). Στη συνέχεια, έπονται αναλυτικά εικόνες για κάθε κύκλωμα και ο κώδικας προσομοίωσης. Η μονάδα χρόνου των προσομοιώσεων είναι το νανοδευτερόλεπτο (ns) ως προεπιλογή της VHDL και της Verilog.

2.7.1 ModelSim

Οι προσομοιώσεις που θα λάβουν χώρο στα παρακάτω κεφάλαια θα πραγματοποιηθούν στο λογισμικό ModelSim. Το πρόγραμμα προσομοιώσεων (ModelSim) που προαναφέρθηκε, δημιουργήθηκε από την Mentor Graphics και αποτελεί ένα γραφικό περιβάλλον, το οποίο χρησιμοποιείται για την προσομοίωση γλωσσών περιγραφής υλικού. Η VHDL, η Verilog και η SystemC αποτελούν τις γλώσσες περιγραφής υλικού, που υποστηρίζει το ModelSim και περιέχει ενσωματωμένο «debugger» για την γλώσσα C. Επιπροσθέτως προσφέρει τη δυνατότητα στον άμεσα ενδιαφερόμενο, να το χρησιμοποιήσει ανεξάρτητα ή σε συνδυασμό με διάφορα άλλα προγράμματα όπως το PSIM. Παράλληλα με τα παραπάνω, μπορεί να λειτουργήσει αρμονικά με MATLAB ή Simulink, με την μια πλευρά να παρέχει αριθμητική προσομοίωση και την άλλη επαλήθευση της υλοποίησης του υλικού και των χαρακτηριστικών χρονισμού.

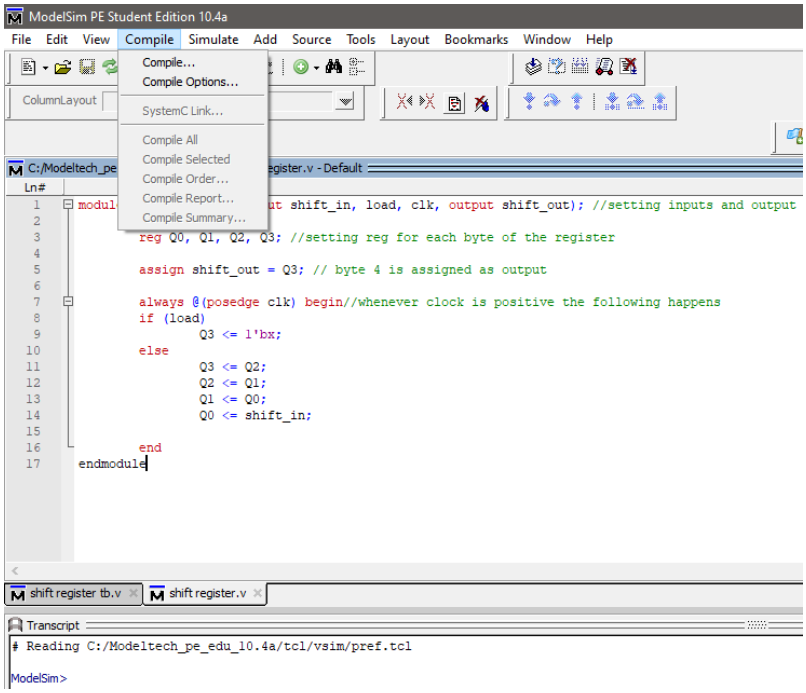
2.7.2 Προσομοίωση μέσω ModelSim

Η προσομοίωση στο ModelSim είναι μια απλή και εύκολη διαδικασία. Για κάθε περιγραφή κυκλώματος η οποία έχει γραφτεί σε μορφή κώδικα, απαιτείται και ο αντίστοιχος κώδικας προσομοίωσης (testbench). Ακόμη η συγγραφή κώδικα προσομοίωσης θα αναλυθεί εκτενώς, μετά το πέρας της υποενότητας. Η διαδικασία της προσομοίωσης θα παρουσιαστεί σε ακόλουθο διάγραμμα ροής. Έπειτα θα αναλυθεί σε βάθος επεξηγώντας εντολές εντός του ModelSim. Τέλος λαμβάνεται ως υπόθεση ότι οι κώδικες για την περιγραφή και τον έλεγχο του κυκλώματος είναι ήδη υλοποιημένοι και έτοιμοι για έλεγχο.

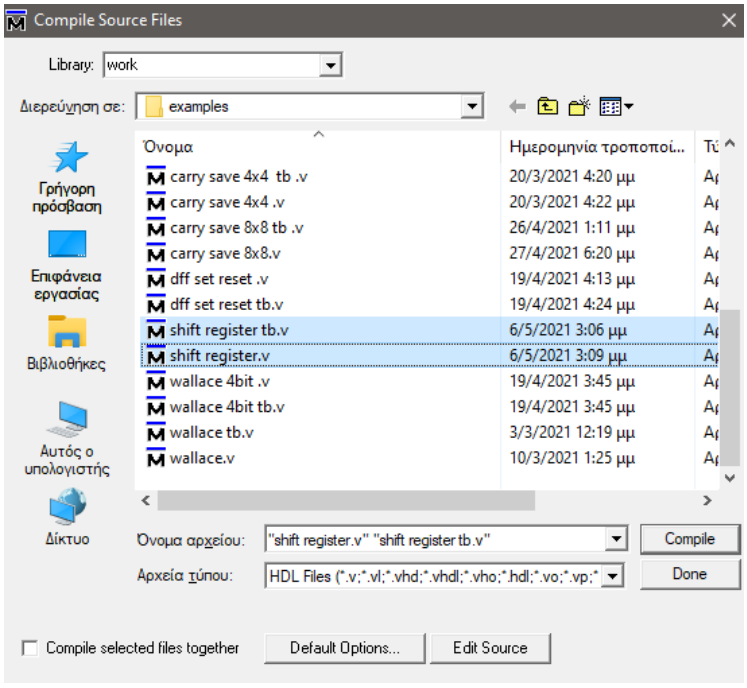


Σχήμα 2.20 Διαδικασία προσομοίωσης με ModelSim

Βήμα 1ο : Πριν την προσομοίωση θα πρέπει να έχει γίνει «compile» (μεταγλώττιση) του κώδικα και να μην βρίσκονται συντακτικά σφάλματα, αλλιώς δεν θα μπορεί να αρχίσει η προσομοίωση. Το compile γίνεται κάνοντας κλικ στην καρτέλα Compile → Compile…, επιλέγοντας τα επιθυμητά αρχεία και πατώντας Compile.

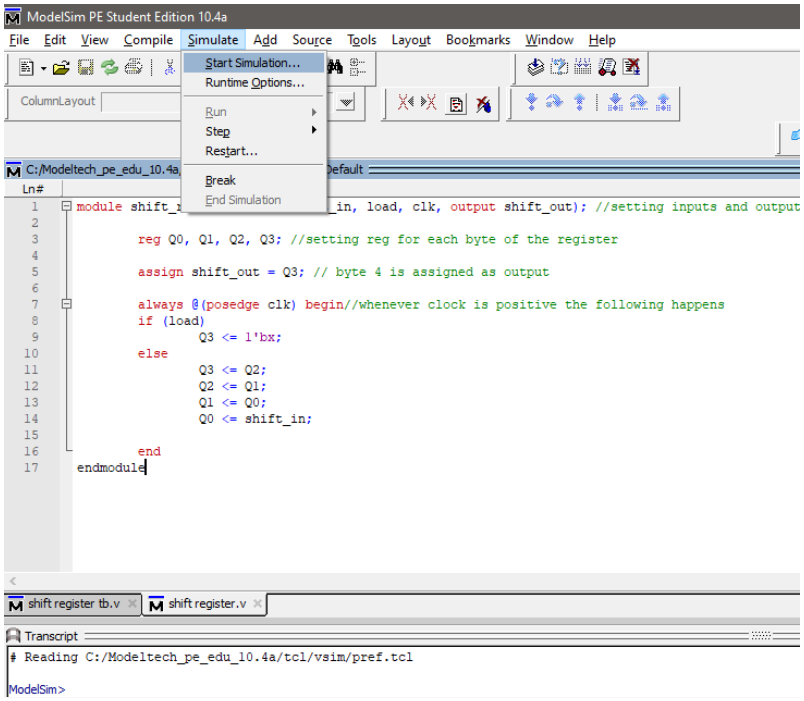


Σχήμα 2.21 Compile σε ModelSim



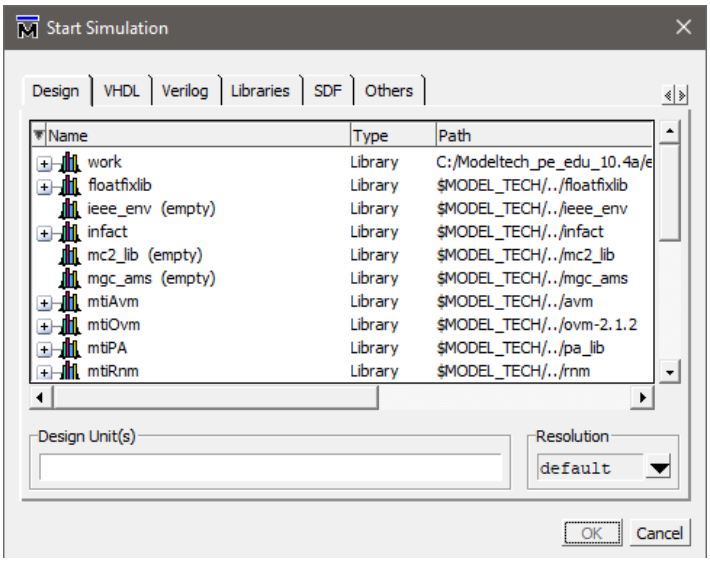
Σχήμα 2.22 Επιλογή αρχείων για Compile

Βήμα 2ο : Το ModelSim, θα εμφανίσει τυχόν συντακτικά λάθη που έχουν προκύψει. Για να ξεκινήσει η προσομοίωση γίνεται κλικ στην καρτέλα Simulate → Start Simulation…



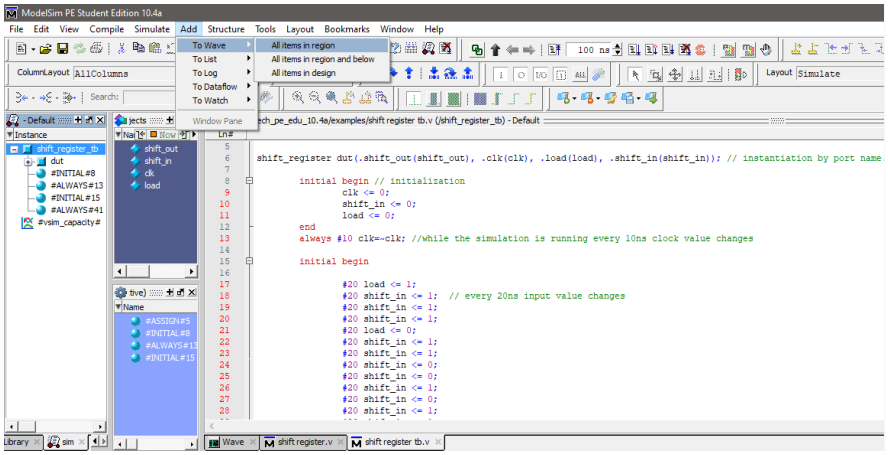
Σχήμα 2.23 Καρτέλα Simulate

Βήμα 3ο : Θα εμφανιστεί μια λίστα από φακέλους, όπου οι κώδικες από προεπιλογή αποθηκεύονται στον φάκελο work. Έτσι έχοντας ανοίξει τον φάκελο αυτό, θα πρέπει να επιλεχτεί ο κώδικας προσομοίωσης του κυκλώματος, το οποίο πρέπει να προσομοιωθεί και να ξεκινήσει η προσομοίωση κάνοντας κλικ στο OK.



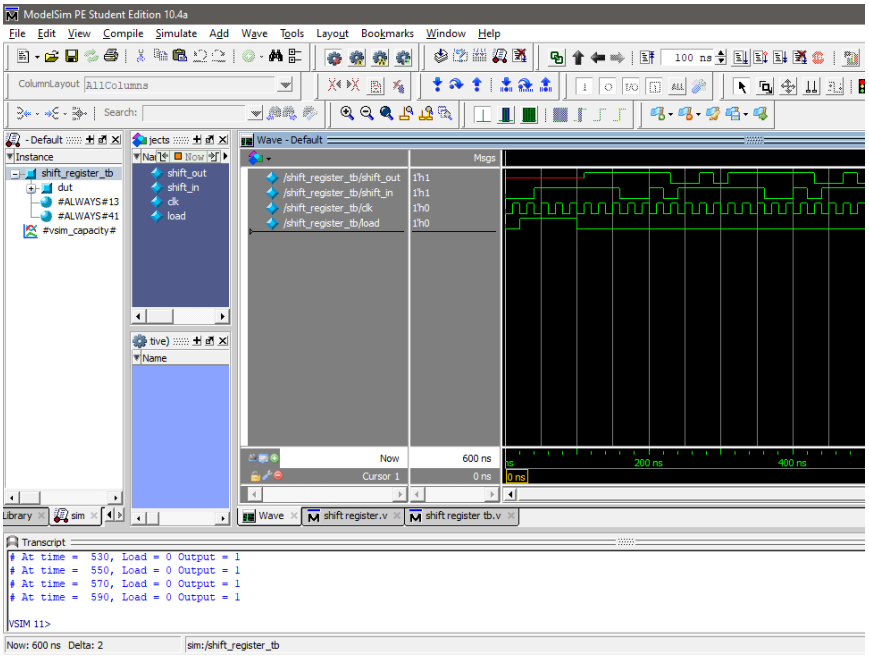
Σχήμα 2.24 Φάκελος Work στο πάνω τμήμα της εικόνας

Βήμα 4ο : Στο συγκεκριμένο παράδειγμα που δίνεται, θα γίνει η προσομοίωση ενός καταχωρητή ολίσθησης. Επομένως για να είναι δυνατή η απεικόνιση κυματομορφών, θα πρέπει να γίνει κλικ στην καρτέλα Add→To Wave→All items in region.



Σχήμα 2.25 Καρτέλα Add

Βήμα 5ο : Κατόπιν έναρξης της διαδικασίας αυτής, υπάρχουν επιλογές για προσομοίωση με συγκεκριμένα βήματα χρόνου (Run) ή συνεχόμενα (Run -all) με επιλογές για χειροκίνητη παύση. Στη προσομοίωση που αναπαριστάται στην εικόνα (2.6), εμφανίζονται επιπρόσθετα μηνύματα σχετικά με τιμές μεταβλητών. Αυτό μπορεί να ρυθμιστεί από τον κώδικα προσομοίωσης. Συμπληρωματικά, κατά την διάρκεια της προσομοίωσης υπάρχει η δυνατότητα τοποθέτησης γραμμών κατά μήκος της, ώστε να ληφθούν πιο ακριβείς μετρήσεις ή να επιτευχθούν διάφορες συγκρίσεις μεταξύ των τμημάτων της.



Σχήμα 2.26 Προσομοίωση σε ModelSim

2.7.3 Προσομοίωση μέσω ModelSim – Εναλλακτικοί μέθοδοι

Το ModelSim εκτός από την μεθοδολογία προσομοίωσης που αναφέρεται παραπάνω φέρει δύο ακόμα μεθόδους, οι οποίες επιτρέπουν την υλοποίηση της. Αναλυτικά η προσομοίωση μπορεί να υφίσταται είτε μέσω γραμμών εντολών εντός του ModelSim, είτε χρησιμοποιώντας αρχεία τύπου batch. Με άλλα λόγια, όταν κάποιος χρησιμοποιήσει το ModelSim, σύμφωνα με τον τρόπο που αναγράφεται στην προηγούμενη υποενότητα, για κάθε κουμπί που πατιέται, είτε για compile, είτε για προσομοίωση, είτε για οτιδήποτε άλλο, εκτελείται μια εντολή εσωτερικά του προγράμματος. Καταλήγοντας το ModelSim δύναται να λειτουργήσει μόνο με εντολές, σαν την συγγραφή κώδικα σε περιβάλλον DOS, όπως φαίνεται παρακάτω.

```
Modelsim> vlib rtl_work
Modelsim> vmap work rtl_work
Modelsim> vlog FA.v
Modelsim> vsim FA
Modelsim> add wave *
Modelsim> run 10 us
```

Σχήμα 2.27 Εντολές ModelSim με τις οποίες εκτελείται προσομοίωση κώδικα Verilog

Η δεύτερη, συνιστά μια πιο εκσυγχρονισμένη μέθοδο από την προαναφερθείσα, καθώς είναι πιο αυτοματοποιημένη με αρχεία τύπου batch. Τα αρχεία τύπου batch απαρτίζονται από εντολές, οι οποίες γράφονται σε ένα αρχείο .txt και τίθενται σε λειτουργία μέσω κάποιου παραθύρου DOS ή UNIX. Συνεπώς, η μέθοδος αυτή αποτελεί μια πιο σύντομη διαδικασία, αφού επιτρέπει την επαναληψιμότητα του εν λόγω κώδικα ModelSim αυτοματοποιημένα, με μια μόνο εκτέλεση του αρχείου. Όλες οι εντολές μπορούν να βρεθούν εδώ: https://www.microsemi.com/document-portal/doc_view/136660-modelsim-me-10-5c-reference-manual-for-libero-soc-v11-8. Ακολουθεί παράδειγμα δημιουργίας batch αρχείου, το οποίο φέρει λειτουργίες παρόμοιες με αυτές της παραπάνω υποενότητας, όπως compile, προσομοίωση και προσθήκη κυματομορφών.

```
\<install_dir>\modeltech\examples\counter.vhd
vlib work
vmap work work
vcom counter.vhd
<install_dir>\modeltech\examples\stim.do
add list -decimal *
do stim.do
write list counter.lst
vsim -do yourfile -wlf saved.wlf counter
vsim -view saved.wlf
view signals list wave
quit -f
```

Σχήμα 2.28 Παράδειγμα συγγραφής batch αρχείου για αρχείο VHDL

2.7.4 Δομή κώδικα προσομοίωσης VHDL

Η αναγκαιότητα της προσομοίωσης βρίσκει εφαρμογή, στη διευκόλυνση του σχεδιαστή για τον έλεγχο του κυκλώματος που έχει σχεδιαστεί. Συγκεκριμένα για τον έλεγχο λειτουργικότητας του κυκλώματος, ο αποτελεσματικότερος τρόπος είναι η υλοποίηση κώδικα προσομοίωσης (Testbench). Στο σχήμα 2.29 παρατηρείται η δομή του κώδικα προσομοίωσης με VHDL.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.all;

ENTITY Ftb IS

END Ftb;

ARCHITECTURE testF OF Ftb IS

SIGNAL x1, x2, x3, f: STD_LOGIC;

COMPONENT example1

 PORT(

 x1, x2, x3 : IN  STD_LOGIC;

        f : OUT STD_LOGIC);

END COMPONENT;

BEGIN

uut: example1 PORT MAP (x1=>x1,x2=>x2,x3=>x3,f=>f);

PROCESS

 BEGIN

 x1 <= '0'; x2 <= '0'; x3 <= '0'; WAIT FOR 20 ns;

 x1 <= '0'; x2 <= '0'; x3 <= '1'; WAIT FOR 20 ns;

 x1 <= '0'; x2 <= '1'; x3 <= '0'; WAIT FOR 20 ns;

 x1 <= '0'; x2 <= '1'; x3 <= '1'; WAIT FOR 20 ns;

 x1 <= '1'; x2 <= '0'; x3 <= '0'; WAIT FOR 20 ns;

 x1 <= '1'; x2 <= '0'; x3 <= '1'; WAIT FOR 20 ns;

 x1 <= '1'; x2 <= '1'; x3 <= '0'; WAIT FOR 20 ns;

 x1 <= '1'; x2 <= '1'; x3 <= '1'; WAIT FOR 20 ns;
```

```
 END PROCESS;

END testF;
```

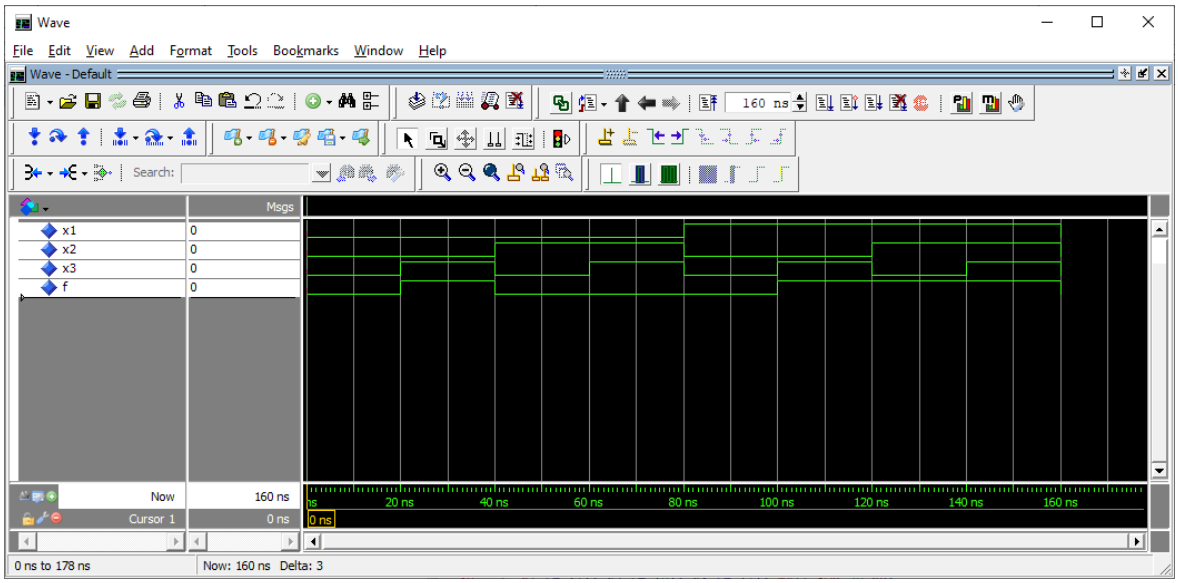
Σχήμα 2.29 Παράδειγμα testbench $f = x_1x_2 + \bar{x}_2x_3$ VHDL.

Η δομή του κώδικα προσομοίωσης συνάδει με τον κώδικα του κυκλώματος. Εντούτοις, υφίστανται εμφανείς διαφορές, όπως η έλλειψη σημάτων κατά την δήλωση της οντότητας, η ένταξη του κυκλώματος ως COMPONENT και η χρήση του sequential statement WAIT FOR. Αναλυτικότερα, δεν υπάρχουν σήματα για να δηλωθούν στο ENTITY, διότι δεν παράγεται κύκλωμα και δεν είναι απαραίτητα τα σήματα εισόδου και εξόδου. Ομοίως, η χρήση της εντολής COMPONENT είναι θεμελιώδης, αφού ενσωματώνεται το κύκλωμα προς προσομοίωση από το οποίο λαμβάνονται τα σήματα εισόδων, εξόδων και η λειτουργία όλου του κυκλώματος. Τέλος, τίθεται σε εφαρμογή η μεταχείριση της sequential statement WAIT FOR, με σκοπό την εκχώρηση τιμών στις εισόδους του κυκλώματος και την εναλλαγή τους με άλλες τιμές σε χρονικό διάστημα νανοδευτερολέπτων (ns).

Παρατήρηση

Ταυτόχρονα με την χρήση της εντολής PORT MAP, ολοκληρώνεται η ταυτοποίηση των σημάτων του κυκλώματος COMPONENT, με τα σήματα της προσομοίωσης.

Ο κώδικας προσομοίωσης παράγει αποτελέσματα, τα οποία είναι ορατά στο Waveform διάγραμμα που δημιουργείται κατά την προσομοίωση του κυκλώματος.



Σχήμα 2.30 Διάγραμμα Waveform προσομοίωσης $f = x_1x_2 + \bar{x}_2x_3$ σε VHDL.

2.7.5 Δομή κώδικα προσομοίωσης Verilog

Ο κώδικας προσομοίωσης, έχει παρόμοια δομή με τον κώδικα των κυκλωμάτων. Αρχικά γίνεται δήλωση μεταβλητών αλλά όχι με ίδιο τρόπο. Ο κώδικας εκκινεί με την εντολή «module» και δίνεται το όνομα του κώδικα προσομοίωσης. Οι είσοδοι δηλώνονται ως «reg» μεταβλητές, ενώ οι έξοδοι ως «wire» μεταβλητές και όχι ως input ή output.

```
module name();
    reg in1, in2, in3 ,… inx;
    wire out1, out2, …. outx;
module example_tb();
    reg x1, x2, x3; // inputs
    wire f; // output
```

Σχήμα 2.31 Παράδειγμα ορισμού μεταβλητών testbench

Στη συνέχεια ακολουθεί αρχικοποίηση μεταβλητών. Πρωτίστως πραγματώνεται η δήλωση του κυκλώματος το οποίο χρειάζεται έλεγχο. Μετέπειτα δηλώνεται ποιες μεταβλητές από τον κώδικα έλεγχου, αντιστοιχούν σε ποιες του κώδικα του κυκλώματος. Αφού πραγματοποιηθούν τα παραπάνω, αρχικοποιούνται οι τιμές που απαιτούνται στην προσομοίωση.

```
example dut(.in1(in1), .in2(in2), .in3(in3), .out1(out1),. Out2(out2); //initialization
    initial begin
        in1=1'b0;
        in2=1'b1;
        in3=1'b0;
    end
example dut(.x1(x1), .x2(x2), .x3(x3), .f(f)); //initialization
    initial begin
        x1=1'b0;
        x2=1'b0;
        x3=1'b0;
```

```
end
```

Σχήμα 2.32 Παράδειγμα αρχικοποίησης μεταβλητών

Τέλος θέτονται οι τιμές που χρειάζονται για την προσομοίωση του κυκλώματος. Οι τιμές τίθενται είτε αυτόματα, είτε χειροκίνητα με χρήση όποιου μπλοκ (initial ή always) επιθυμεί ο χρήστης. Παράλληλα με την εντολή «$monitor» στο παράδειγμα παρακάτω εμφανίζονται τα αποτελέσματα, αλλά μπορεί και να χρησιμοποιηθεί για άλλους σκοπούς. Ο κώδικας λήγει με την εντολή «endmodule».

```
initial begin
#5 x1=1'b1; x2=1'b0; x3=1'b0;          always #10 x1=~x1;
#5 x1=1'b1; x2=1'b0; x3=1'b1;          always #20 x2=~x2;
#5 x1=1'b1; x2=1'b1; x3=1'b0;           always #40 x3=~x3;
end
```

Σχήμα 2.33 Παράδειγμα εισόδου τιμών

```
always @(x1 or x2 or x3)
// for every change in inputs the following message appears
$monitor ("At time = %4d, X1=%d, X2=%d, X3=%d, Output = %d", $time, x1, x2, x3, f);
// message with extra info
endmodule
```

Σχήμα 2.34 Χρήση εντολής $monitor για ένδειξη τιμών και λήξη κώδικα

2.8 Περιγραφή συνδυαστικών κυκλωμάτων με VHDL και Verilog

Τα συνδυαστικά κυκλώματα αποτελούν ένα από τα δύο είδη κυκλωμάτων που χρησιμοποιούνται για την υλοποίηση ψηφιακών συστημάτων και συντίθενται από σύνολα λογικών πυλών. Στις πύλες αυτές εφαρμόζονται σήματα εισόδων και παράγονται συγκεκριμένα σήματα εξόδων, μετασχηματίζοντας έτσι την δυαδική πληροφορία που παρέχεται στα σήματα εισόδου. Τα σήματα εισόδων και εξόδων ερμηνεύονται ως δυαδικά σήματα, δηλαδή λογικά 1 και λογικά 0. Για δεδομένο αριθμό n εισόδων υπάρχουν ως αποτέλεσμα μέχρι 2^n δυνατοί έξοδοι. Αποτελεσματικά κατανοείται ότι για κάθε συνδυαστικό κύκλωμα μπορεί να παραχθεί ένας πίνακας αληθείας, που εξάγει τιμές εξόδων για δεδομένες τιμές εισόδων. Υπάρχει επίσης η δυνατότητα αναπαράστασης ακολουθιακών κυκλωμάτων με την χρήση λογικών συναρτήσεων. Ωστόσο τα συνδυαστικά κυκλώματα είναι απλοϊκά

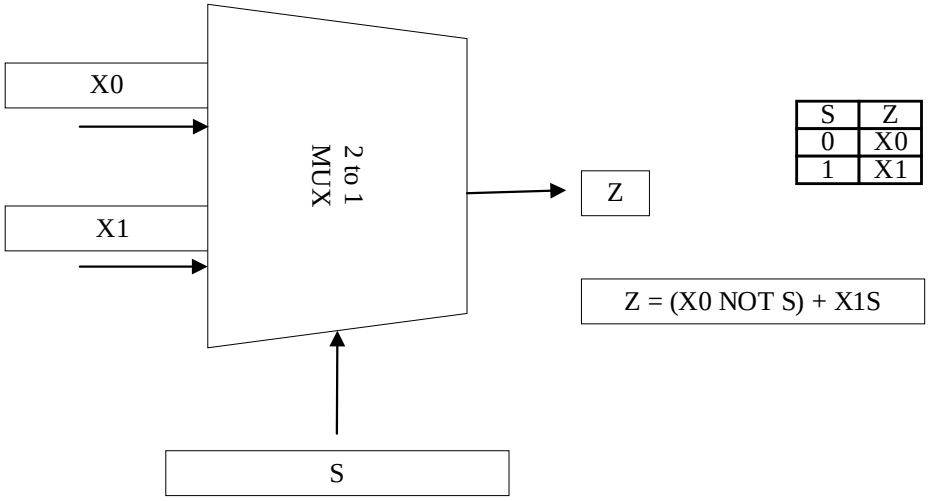
κάτι εμφανές στα διαγράμματα τους. Ένα διάγραμμα συνδυαστικού κυκλώματος περιέχει μόνο λογικές πύλες χωρίς βρόγχους ανάδρασης ή στοιχεία μνήμης. Τα παρακάτω κυκλώματα θα αναλυθούν με βάση είτε την λογική τους παράσταση είτε με βάση την συμπεριφορά τους, με σκοπό την ανάλυση των τρόπων αντιστοιχίσεων (assignments) που προσφέρουν οι VHDL και Verilog.

2.9 Περιγραφή με χρήση συνεχόμενης αντιστοίχισης (continuous assignment)

Ως συνεχόμενη αντιστοίχιση, στην Verilog αναφέρεται η χρήση της εντολής «assign», η οποία είναι ιδανική για την οδήγηση λογικών παραστάσεων. Με τον συγκεκριμένο τρόπο η τιμή της εξόδου αλλάζει κάθε φορά, που πραγματοποιείται οποιαδήποτε αλλαγή σε κάποια είσοδο. Σε επόμενες υλοποιήσεις θα εξεταστούν κυκλώματα υλοποιημένα με χρήση αυτής της μεθόδου.

2.10 Περιγραφή του 2 σε 1 πολυπλέκτη με VHDL και Verilog

Ο πολυπλέκτης 2 σε 1 είναι ένα πολύ διαδεδομένο ψηφιακό κύκλωμα, το οποίο εξάγει την ανάλογη έξοδο, σύμφωνα με όποια είσοδο έχει ενεργοποιημένη. Η πολυπλεξία ως διαδικασία αποτελεί τον συνδυασμό ενός ή πολλαπλών σημάτων και η μετάδοση τους σε ένα μοναδικό κανάλι. Ο πολυπλέκτης λειτουργεί με αυτή τη διαδικασία υπόψιν. Δηλαδή, μπορεί με φθηνό και αποδοτικό τρόπο να πραγματοποιεί πολυπλεξία με βάση την λογική. Ακολουθούν, ο πίνακας αληθείας και η λογική παράσταση.



S	Z
0	X0
1	X1

Σχήμα 2.35 Ο πολυπλέκτης 2 σε 1

Με x_0, x_1 συμβολίζονται οι είσοδοι δεδομένων, με s η είσοδος επιλογής και με z η έξοδός του κυκλώματος. Η περιγραφή του κυκλώματος θα γίνει με χρήση μοντελοποίησης ροής δεδομένων, δηλαδή βάσει της λογικής παράστασης του, $z = x_0\bar{s} + x_1 s$.

2.10.1 Περιγραφή του 2 σε 1 πολυπλέκτη με VHDL

Παρακάτω περιγράφεται ο πολυπλέκτης 2 σε 1 σε VHDL κώδικα.

```
LIBRARY ieee;

USE ieee.std_logic_1164.all;


ENTITY Mux2to1 IS

   PORT (x0, x1, s : IN  STD_LOGIC;

              z : OUT STD_LOGIC);

END Mux2to1;


ARCHITECTURE LogicFunc OF Mux2to1 IS

   BEGIN

          z<= (x0 AND (NOT s)) OR (x1 AND s);

END LogicFunc;
```

Σχήμα 2.36 Περιγραφή του πολυπλέκτη 2 σε 1 με συνεχόμενη αντιστοίχιση VHDL.

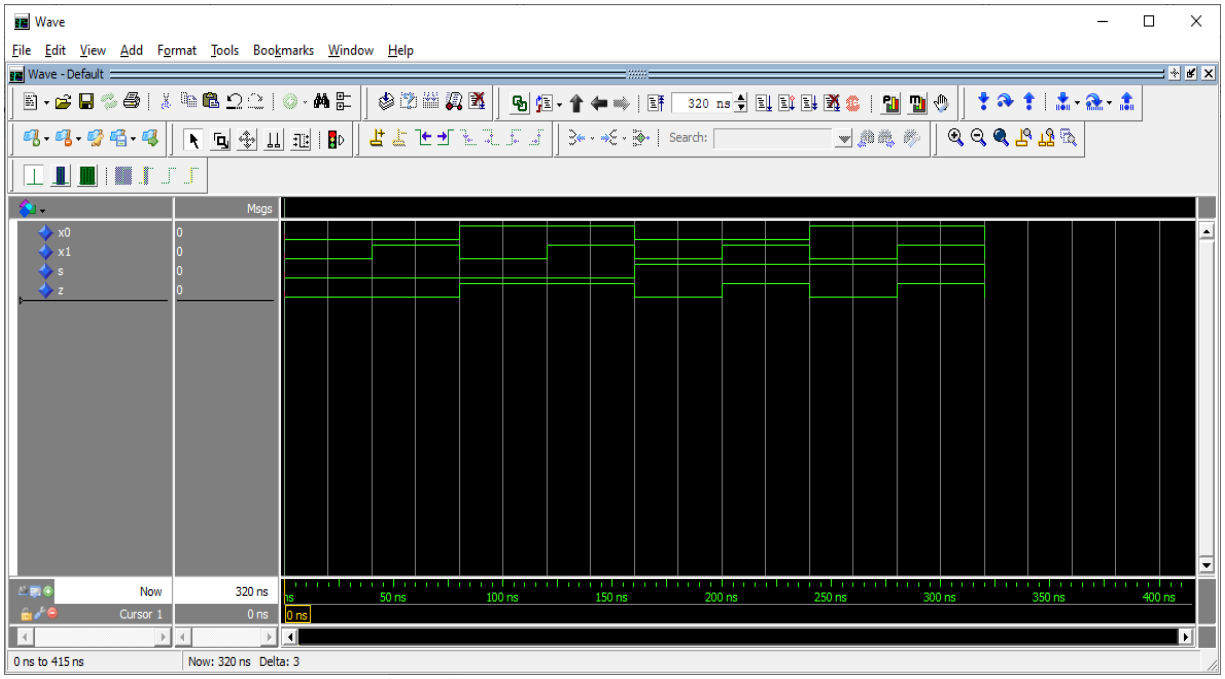
2.10.2 Προσομοίωση πολυπλέκτη 2 σε 1 με VHDL

Ακολουθεί ο κώδικας της προσομοίωσης (Testbench) του πολυπλέκτη 2 σε 1. Ο πολυπλέκτης δέχεται ως είσοδο όλες τις πιθανές τιμές που μπορεί να λάβει σε απόσταση 40 ns (νανοδευτερολέπτων) μεταξύ τους. Τέλος, τα αποτελέσματα για κάθε είσοδο είναι ορατά από το επακόλουθο Waveform διάγραμμα.

```
LIBRARY IEEE;

USE IEEE.STD_LOGIC_1164.ALL;


ENTITY Mux2to1tb IS

END Mux2to1tb;


ARCHITECTURE behavior OF Mux2to1tb IS


```

```
SIGNAL x0, x1, s, z : STD_LOGIC;


COMPONENT Mux2to1
PORT(x0, x1, s : IN  STD_LOGIC;
             z : OUT STD_LOGIC);
END COMPONENT;


 BEGIN


m1: Mux2to1 PORT MAP(x0=>x0,x1=>x1,s=>s,z=>z);


PROCESS
BEGIN


 s<='0';x0<='0';x1<='0'; WAIT FOR 40 ns;
 s<='0';x0<='0';x1<='1'; WAIT FOR 40 ns;
 s<='0';x0<='1';x1<='0'; WAIT FOR 40 ns;
 s<='0';x0<='1';x1<='1'; WAIT FOR 40 ns;
 s<='1';x0<='0';x1<='0'; WAIT FOR 40 ns;
 s<='1';x0<='0';x1<='1'; WAIT FOR 40 ns;
 s<='1';x0<='1';x1<='0'; WAIT FOR 40 ns;
 s<='1';x0<='1';x1<='1'; WAIT FOR 40 ns;


 END PROCESS;
END behavior;
```

Σχήμα 2.37 Testbench πολυπλέκτη 2 σε 1 VHDL.



Σχήμα 2.38 Προσομοίωση πολυπλέκτη 2 σε 1 VHDL.

2.10.3 Περιγραφή του 2 σε 1 πολυπλέκτη με Verilog

Η περιγραφή με Verilog βρίσκεται παρακάτω στο σχήμα 2.39.

```
module mux_2to1(x0, x1, s, z); //setting inputs and output
input x0, x1,s;
output z;
assign z = (x0 & ~s) | (x1 & s); //assigning behavior to output as a logic function
endmodule
```

Σχήμα 2.39 Περιγραφή του πολυπλέκτη 2 σε 1 με συνεχόμενη αντιστοίχιση

Επιπροσθέτως, άξια αναφοράς είναι μια διαφορετική υλοποίηση της περιγραφής του κυκλώματος, με χρήση μιας διαφορετικής μορφής της εντολής «assign», η οποία παρουσιάζεται στο παρακάτω σχήμα (Σχήμα 2.40). Απεναντίας με το παραπάνω, η συγκεκριμένη υλοποίηση κάνει χρήση του τελεστή «?» για να λειτουργήσει σαν εντολή «if», στην οποία αν το S είναι θετικό, η έξοδος είναι το X1 αλλιώς το X0.

```
module mux_2to1(input X0, X1, S, output Z);  //setting inputs and output
assign Q=(S)?X1:X0;//assigning behavior to output as an if statement function
endmodule
```

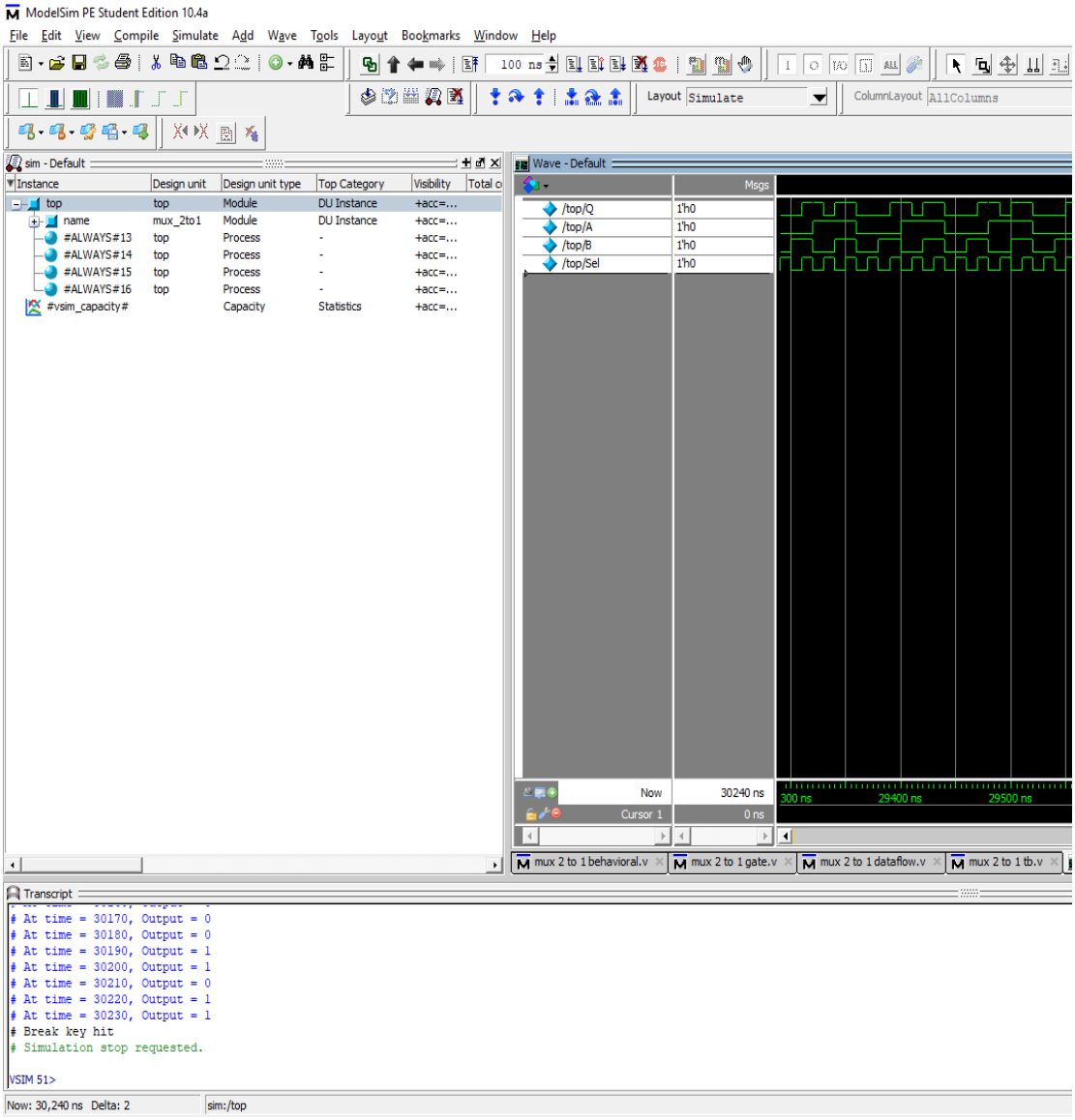
Σχήμα 2.40 Περιγραφή του πολυπλέκτη 2 σε 1 με εναλλακτική συνεχόμενη αντιστοίχιση

2.10.4 Προσομοίωση πολυπλέκτη 2 σε 1 με Verilog

Ο πολυπλέκτης 2 σε 1 δέχεται δύο εισόδους του ενός bit με αρχικοποίηση στο μηδέν. Ανάλογα με την τιμή της εισόδου επίτρεψης, εμφανίζει την είσοδο που αντιστοιχεί ως έξοδο. Ο πίνακας αληθείας του βρίσκεται στο σχήμα 2.41. Παρακάτω ακολουθούν το testbench και η προσομοίωση.

```
module mux_2to1_tb;  //setting inputs and output
    wire z;
    reg x0, x1, s;
    mux_2to1 dut(.x0(x0), .x1(x1), .z(z), .s(s)); // instantiation by port name.

    initial begin// initialization
       x0=1'b0;
       x1=1'b0;
       s=1'b0;
   end

   always #40 x0=~x0; // every set time in ns the value of inputs changes
   always #20 x1=~x1;
   always #10 s=~s;
   always@(x0 or x1 or s) // if any input changes the message appears

   $monitor("At time = %4d, Output = %b", $time, z); // message for extra info
    endmodule
```

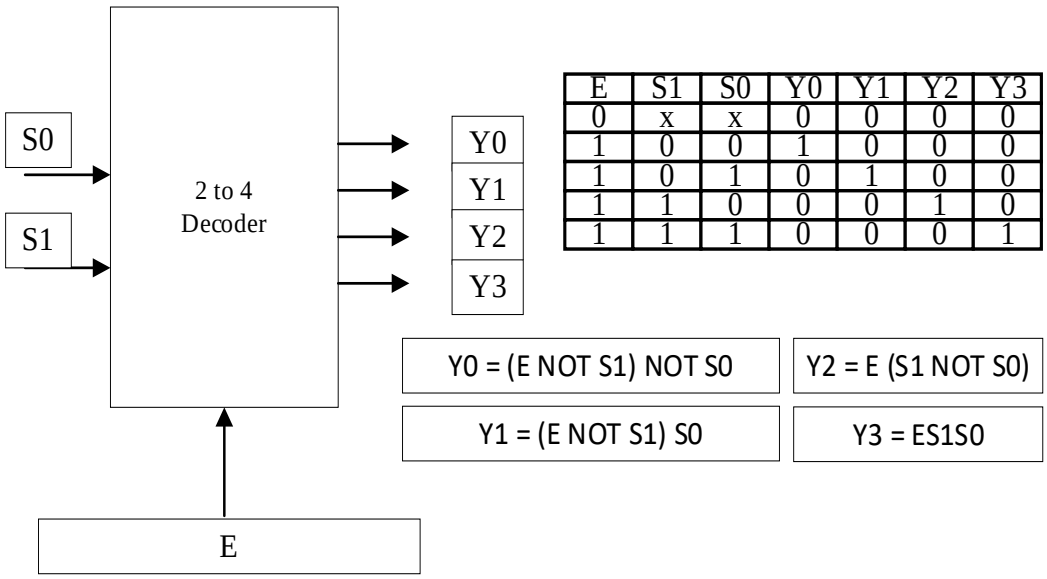
Σχήμα 2.41 Testbench πολυπλέκτη 2 σε 1



Σχήμα 2.42 Προσομοίωση πολυπλέκτη 2 σε 1

2.11 Περιγραφή αποκωδικοποιητή 2 σε 4 με VHDL και Verilog

Ο αποκωδικοποιητής 2 σε 4 με είσοδο ενεργοποίησης, είναι ένα ακόμα διαδεδομένο ψηφιακό κύκλωμα. Οι αποκωδικοποιητές είναι απαραίτητοι για εφαρμογές όπως πολυπλεξία δεδομένων, αποκωδικοποίηση διεύθυνσης μνήμης και οθόνη 7-segment. Στο ακόλουθο σχήμα δίνονται το λογικό σύμβολο, ο πίνακας αληθείας και οι λογικές παραστάσεις των εξόδων του κυκλώματος. Με s_0, s_1 εμφανίζονται οι είσοδοι, με e η είσοδος ενεργοποίησης και με y_0, y_1, y_2, y_3 οι έξοδοι.



E	S1	S0	Y0	Y1	Y2	Y3
0	x	x	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

Σχήμα 2.43 αποκωδικοποιητής 2 σε 4 με είσοδο ενεργοποίησης

2.11.1 Περιγραφή αποκωδικοποιητή 2 σε 4 με VHDL

Ακολουθεί ο κώδικας του αποκωδικοποιητή 2 σε 4 με VHDL.

```
LIBRARY ieee;

USE ieee.std_logic_1164.all;


ENTITY Dec_2_to_4 IS

 PORT (     e, s0, s1 : IN   STD_LOGIC;

       y0, y1, y2, y3 : OUT  STD_LOGIC);

END Dec_2_to_4;


ARCHITECTURE LogicFunc OF Dec_2_to_4 IS

 BEGIN

    y0<= ((NOT s1) AND (NOT s0) AND e );

    y1<= ((NOT s1) AND s0 AND e);

    y2<= (s1 AND (NOT s0) AND e);

    y3<= (s1 AND s0 AND e);

END LogicFunc;
```

Σχήμα 2.44 αποκωδικοποιητής 2 σε 4 με συνεχόμενη αντιστοίχιση VHDL.

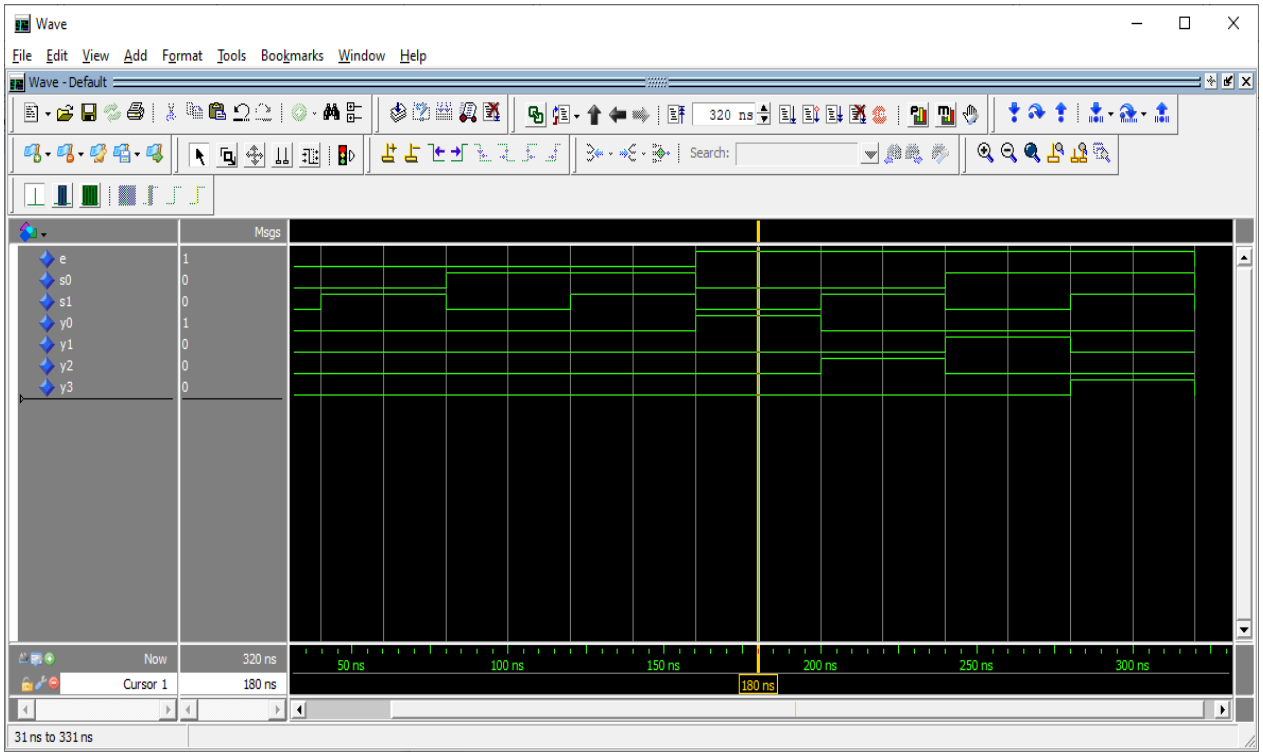
2.11.2 Προσομοίωση αποκωδικοποιητή 2 σε 4 με VHDL

Η προσομοίωση του αποκωδικοποιητή 2 σε 4 επιτυγχάνεται με το παρακάτω Testbench. Ο αποκωδικοποιητής λαμβάνει όλες τις πιθανές τιμές ως είσοδο για κάθε 40 ns και τα αποτελέσματα απεικονίζονται στο αντίστοιχο Waveform.

```
LIBRARY IEEE;

USE IEEE.STD_LOGIC_1164.ALL;


ENTITY Dec_2_to_4tb IS

END Dec_2_to_4tb;


ARCHITECTURE behavior OF Dec_2_to_4tb IS


SIGNAL e, s0, s1, y0, y1, y2, y3 : STD_LOGIC;


COMPONENT Dec_2_to_4

PORT (     e, s0, s1 : IN  STD_LOGIC;

     y0, y1, y2, y3 : OUT STD_LOGIC);

END COMPONENT;


 BEGIN


m1: Dec_2_to_4 PORT
MAP(e=>e,s0=>s0,s1=>s1,y0=>y0,y1=>y1,y2=>y2,y3=>y3);


PROCESS

BEGIN
```

```
e<='0';s0<='0';s1<='0'; WAIT FOR 40 ns;

e<='0';s0<='0';s1<='1'; WAIT FOR 40 ns;

e<='0';s0<='1';s1<='0'; WAIT FOR 40 ns;

e<='0';s0<='1';s1<='1'; WAIT FOR 40 ns;

e<='1';s0<='0';s1<='0'; WAIT FOR 40 ns;

e<='1';s0<='0';s1<='1'; WAIT FOR 40 ns;

e<='1';s0<='1';s1<='0'; WAIT FOR 40 ns;

e<='1';s0<='1';s1<='1'; WAIT FOR 40 ns;


 END PROCESS;

END behavior;
```

Σχήμα 2.45 Testbench αποκωδικοποιητή 2 σε 4 VHDL.



Σχήμα 2.46 Προσομοίωση αποκωδικοποιητή 2 σε 4 VHDL.

2.11.3 Περιγραφή αποκωδικοποιητή 2 σε 4 με Verilog

Στο παρακάτω σχήμα ακολουθεί η μοντελοποίηση του αποκωδικοποιητή 2 σε 4 με χρήση ροής δεδομένων, χρησιμοποιώντας Verilog.

```
module dec2_4 (input s0,s1,e, output y0,y1,y2,y3); // setting inputs and outputs
        assign y0= (~s1) & (~s0) & e; //assigning behavior to each output
        assign y1= (~s1) & s0 & e;
        assign y2= s1 & (~ s0) & e;
        assign y3= s1 & s0 & e;
endmodule
```

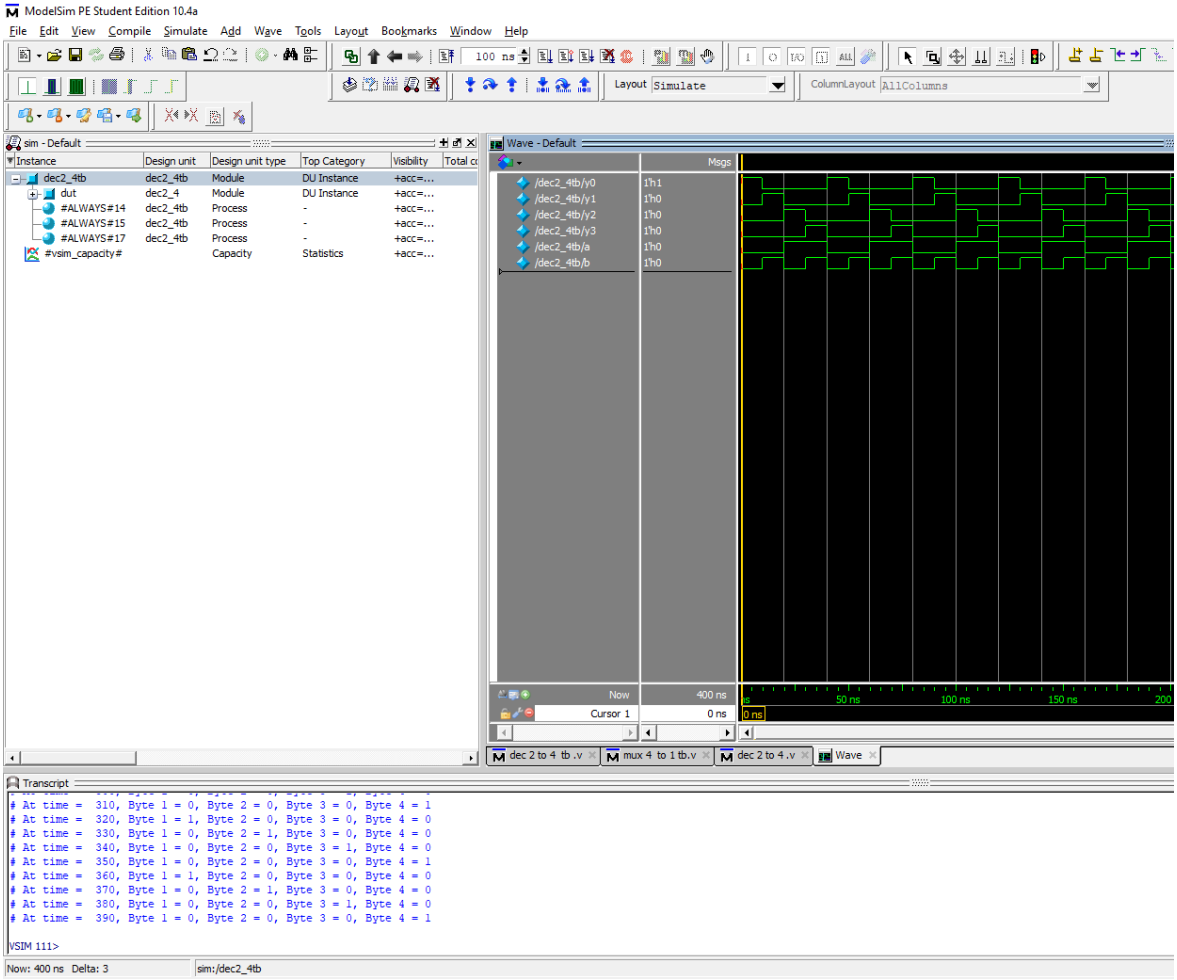
Σχήμα 2.47 αποκωδικοποιητής 2 σε 4 με συνεχόμενη αντιστοίχιση

2.11.4 Προσομοίωση αποκωδικοποιητή 2 σε 4 με Verilog

Ο αποκωδικοποιητής 2 σε 4 δέχεται δύο εισόδους μήκους ενός bit και έχει τέσσερις εξόδους μήκους ενός bit. Οι είσοδοι αρχικοποιούνται στο μηδέν και οι τιμές μεταξύ τους εναλλάσσονται κάθε είκοσι και δέκα νανοδευτερόλεπτα. Ο πίνακας αληθείας του βρίσκεται στο σχήμα 2.48. Παρακάτω ακολουθούν το testbench και η προσομοίωση.

```
module dec2_4tb(); //setting inputs and outputs

        wire y0,y1,y2,y3;

         reg a, b;

        dec2_4 dut(.y0(y0), .y1(y1), .y2(y2), .y3(y3), .a(a), .b(b)); //initialization

        initial begin

                        a <= 1'b0; b <= 1'b0;

        end

        always #20 a=~a; // inputs change value at designated time in ns

        always #10 b=~b;

always@(a or b) // for every change in the inputs the following message appears

$monitor("At time = %4d, Byte 1 = %d, Byte 2 = %d, Byte 3 = %d, Byte 4 = %d", $time, y0, y1, y2, y3); // message with extra info

endmodule
```

Σχήμα 2.48 Testbench αποκωδικοποιητή 2 σε 4



Σχήμα 2.49 Προσομοίωση αποκωδικοποιητή 2 σε 4

2.12 Περιγραφή με χρήση διαδικαστικής αντιστοίχισης (procedural assignment) με τις VHDL και Verilog

Η διαδικαστική αντιστοίχιση (procedural assignment) είναι η χρήση των διαδικασιών «PROCESS» για την VHDL και «always» ή «initial» για την Verilog, με σκοπό τη δημιουργία ενός μπλοκ, όπου θα γίνεται η εισαγωγή τιμών σε μεταβλητές. Η τιμή θα εισαχτεί στη μεταβλητή, κάποια στιγμή κατά την εκτέλεση της προσομοίωσης. Αυτό μπορεί να ελεγχθεί με εντολές, όπως «if-else», «case» και βρόγχους επανάληψης.

2.12.1 Περιγραφή του πολυπλέκτη 2 σε 1 με VHDL

Η χρήση της εντολής PROCESS επιτρέπει την δημιουργία του κυκλώματος του πολυπλέκτη 2 σε 1 με την χρήση Sequential Statement IF THEN. Παρακάτω παρατίθεται ο κώδικας του κυκλώματος σε VHDL.

```
LIBRARY ieee;
```

```
USE ieee.std_logic_1164.all;


ENTITY Mux_2_to_1 IS
   PORT (x0, x1, s : IN  STD_LOGIC;
               f : OUT STD_LOGIC);
END Mux_2_to_1;


ARCHITECTURE Behavioral OF mux_2_to_1 IS
BEGIN
  PROCESS ( x0, x1, s )
  BEGIN
    IF s = '0' THEN f <= x0 ;
    ELSE
       f <= x1 ;
    END IF ;
  END PROCESS ;
END Behavioral;
```

Σχήμα 2.50 Περιγραφή πολυπλέκτη 2 σε 1 με διαδικαστική αντιστοίχιση

2.12.2 Περιγραφή του πολυπλέκτη 2 σε 1 με Verilog

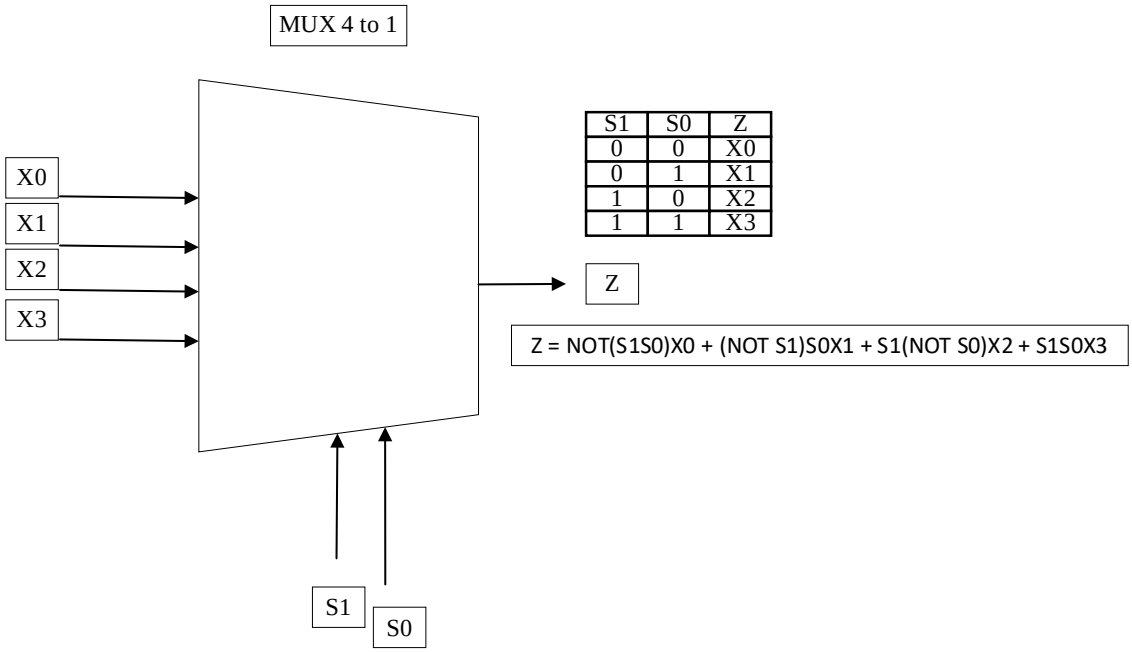
Παραπάνω στο σχήμα 2.35 έχει δοθεί η περιγραφή του πολυπλέκτη 2 σε 1 με συνεχόμενη αντιστοίχιση. Η διαδικαστική αντιστοίχιση χρησιμοποιεί τη διαδικασία «always» μέσα στην οποία πραγματώνεται η εντολής «if-else» ώστε να ολοκληρωθεί η περιγραφή. Η ιδιότητα της «always», είναι ότι αν πραγματοποιηθεί οποιαδήποτε αλλαγή σε κάποια από τις δηλωθέν τιμές της, στη συγκεκριμένη περίπτωση x0, x1, S, θα λειτουργεί το μπλοκ κώδικα, που βρίσκεται μεταξύ του begin και του end. Η εντολή if στον κώδικα του σχήματος 2.51 θέτει ότι αν το S είναι «1» η έξοδος Z θα είναι το x1, αλλιώς θα είναι το x0.

```
module mux_2to1( x0, x1, S, Z);  //setting inputs and output
input wire x0, x1, S;
output reg Q;
always @(x0 or x1 or S) begin  // if any input changes the  following happens
                if (S)
                        Z= x1;
                else
                        Z=x0;
        end
endmodule
```

Σχήμα 2.51 Περιγραφή του πολυπλέκτη 2 σε 1 με χρήση διαδικαστικής αντιστοίχισης

2.13 Περιγραφή του πολυπλέκτη 4 σε 1 με VHDL και Verilog

Ο πολυπλέκτης 4 σε 1 αποτελεί μια βελτίωση του 2 σε 1 με βάση την ίδια λογική. Στο σχήμα 2.52 δίνονται το λογικό σύμβολο, ο πίνακας αληθείας και η λογική παράσταση που υλοποιεί το κύκλωμα. Με x_0, x_1, x_3, x_4 συμβολίζονται οι είσοδοι με s_1, s_0 οι είσοδοι ελέγχου και με Z η έξοδος.



S1	S0	Z
0	0	X0
0	1	X1
1	0	X2
1	1	X3

Σχήμα 2.52 Πολυπλέκτης 4 σε 1

2.13.1 Περιγραφή του πολυπλέκτη 4 σε 1 με VHDL

Ο παρακάτω κώδικας VHDL περιγράφει τον πολυπλέκτη 4 σε 1. Η λογική της υλοποίησης παραμένει ίδια με την διαφορά ότι υπάρχει μια 2-bit είσοδο επίτρεψης, όπου για ανάλογη τιμή της εμφανίζεται η ανάλογη τιμή εισόδου στην έξοδο (παράδειγμα για s= 01 τότε το αποτέλεσμα είναι y).

```
LIBRARY ieee;
```

```
USE ieee.std_logic_1164.all;


ENTITY Mux_4_to_1 IS
 PORT (x0, x1, x2, x3 : IN   STD_LOGIC;
                s : IN   STD_LOGIC_VECTOR(1 DOWNTO 0);
                y : OUT  STD_LOGIC);
END Mux_4_to_1;


ARCHITECTURE behavior OF Mux_4_to_1 IS
          BEGIN
      WITH s SELECT
           y  <= x0 WHEN "00",
                 x1 WHEN "01",
                 x2 WHEN "10",
                 x3 WHEN OTHERS;
END behavior;
```

Σχήμα 2.53 Περιγραφή του πολυπλέκτη 4 σε 1 με χρήση διαδικαστικής αντιστοίχισης VHDL.

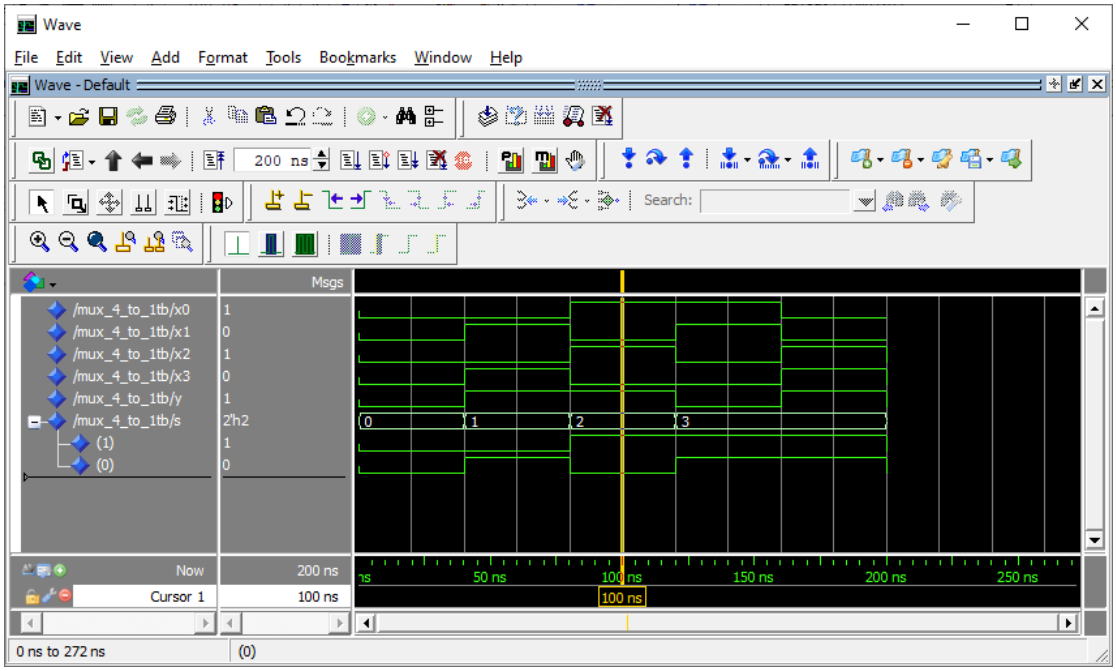
2.13.2 Προσομοίωση πολυπλέκτη 4 σε 1 με VHDL

Η προσομοίωση του πολυπλέκτη 4 σε 1 επιτυγχάνεται με Testbench, το οποίο λαμβάνει διάφορες εισόδους ανά 40 ns. Τα αποτελέσματα είναι εμφανείς στο επακόλουθο Waveform διάγραμμα.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;


ENTITY Mux_4_to_1tb IS
END Mux_4_to_1tb;
```

```
ARCHITECTURE behavior OF Mux_4_to_1tb IS


SIGNAL x0, x1, x2, x3, y : STD_LOGIC;
SIGNAL s : STD_LOGIC_VECTOR(1 DOWNTO 0);


COMPONENT Mux_4_to_1
PORT(x0, x1, x2, x3 : IN  STD_LOGIC;
             s : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
                y : OUT STD_LOGIC);
END COMPONENT;


 BEGIN


m1: Mux_4_to_1 PORT MAP(s=>s,x0=>x0,x1=>x1,x2=>x2,x3=>x3,y=>y);


PROCESS
BEGIN


 s<="00";x0<='0';x1<='0';x2<='0';x3<='0'; WAIT FOR 40 ns;
 s<="01";x0<='0';x1<='1';x2<='0';x3<='1'; WAIT FOR 40 ns;
 s<="10";x0<='1';x1<='0';x2<='1';x3<='0'; WAIT FOR 40 ns;
 s<="11";x0<='1';x1<='1';x2<='0';x3<='0'; WAIT FOR 40 ns;
 s<="11";x0<='0';x1<='0';x2<='1';x3<='1'; WAIT FOR 40 ns;


 END PROCESS;
END behavior;
```

Σχήμα 2.54 Testbench πολυπλέκτη 4 σε 1 VHDL.



Σχήμα 2.55 Προσομοίωση πολυπλέκτη 4 σε 1 VHDL.

2.13.3 Περιγραφή του πολυπλέκτη 4 σε 1 με Verilog

Στο σχήμα 2.56 παρουσιάζεται η περιγραφή του.

```
module mux_4to1 ( A, B, C, D, Sel0, Sel1, Q); //setting inputs and output
        input wire A, B, C, D, Sel0, Sel1;
        output reg Q;
always @ begin(A or B or C or D or Sel0, Sel1)
//if any input changes the following happens
        case (Sel0 | Sel1)
                2'b00 : Q <= A;
                2'b01 : Q <= B;
                2'b10 : Q <= C;
                2'b11 : Q <= D;
        endcase
end
endmodule
```

Σχήμα 2.56 Περιγραφή του πολυπλέκτη 4 σε 1 με χρήση διαδικαστικής αντιστοίχισης

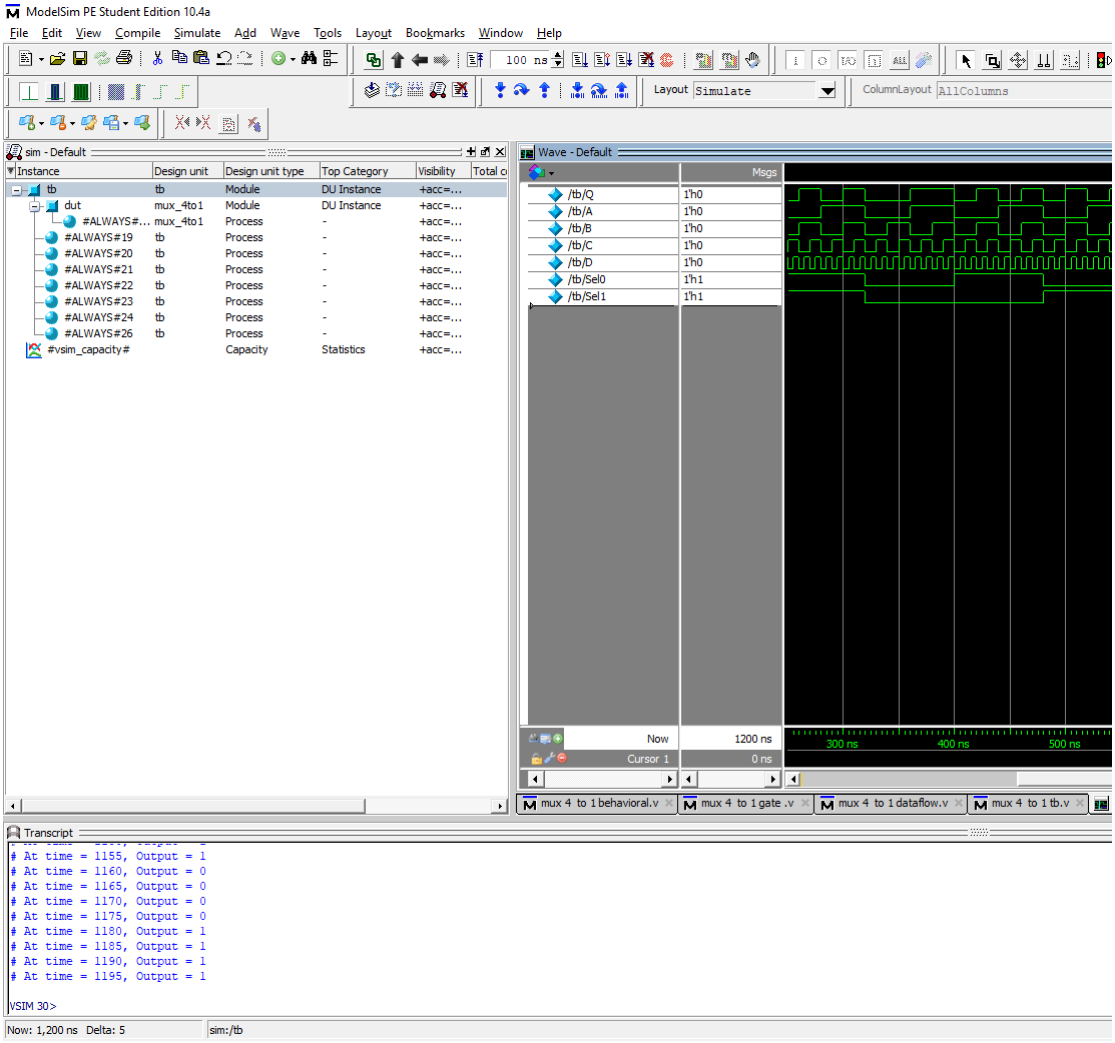
2.13.4 Προσομοίωση πολυπλέκτη 4 σε 1 με Verilog

Ο πολυπλέκτης 4 σε 1 λειτουργεί με τον ίδιο τρόπο που λειτουργεί και ο 2 σε 1 Βασική διαφορά είναι ότι δέχεται τέσσερις εισόδους και απαιτεί δύο εισόδους επίτρεψης για να τις διαχειριστεί. Οι είσοδοι και οι έξοδοι έχουν μήκος ένα bit και οι είσοδοι αρχικοποιούνται στο μηδέν. Εναλλαγές μεταξύ τιμών στις εισόδους γίνονται

κάθε πέντε, δέκα, είκοσι, σαράντα νανοδευτερόλεπτα, ενώ στις εισόδους επίτρεψης κάθε ογδόντα και εκατόν εξήντα νανοδευτερόλεπτα. Ο πίνακας αληθείας βρίσκεται στο σχήμα 2.57. Παρακάτω ακολουθούν το testbench και η προσομοίωση.

```
module mux_4to1_tb; //setting inputs and output
    wire  Q;
    reg  A, B, C, D, Sel0, Sel1;

mux_4to1 dut(.Q(Q), .A(A), .B(B), .C(C), .D(D), .Sel0(Sel0), .Sel1(Sel1));
    initial begin // initialization
    A=1'b0;
     B=1'b0;
    C=1'b0;
    D=1'b0;
    Sel0=1'b0;
    Sel1=1'b0;
    end

    always #40 A=~A; // every set time in ns the value of inputs changes
    always #20 B=~B;
    always #10 C=~C;
    always #5 D=~D;
   always #80 Sel0=~Sel0;
   always #160 Sel1=~Sel1;
always@(A or B or C or D or Sel0 or Sel1)
$monitor("At time = %4d, Output = %D", $time, Q); // message for extra info
endmodule
```

Σχήμα 2.57 Testbench πολυπλέκτη 4 σε 1



Σχήμα 2.58 Προσομοίωση πολυπλέκτη 4 σε 1

2.14 Περιγραφή του αποκωδικοποιητή 2 σε 4 με είσοδο επίτρεψης με VHDL και Verilog

Το κύκλωμα που περιγράφεται είναι το ίδιο με παραπάνω αλλά δέχεται και μια είσοδο επίτρεψης η οποία επιτρέπει την λειτουργία του κυκλώματος μόνο όταν αυτή λαμβάνει θετικές τιμές.

2.14.1 Περιγραφή του αποκωδικοποιητή 2 σε 4 με είσοδο επίτρεψης με VHDL

Ακολουθεί η περιγραφή του αποκωδικοποιητή 2 σε 4 με είσοδο επίτρεψης σε VHDL κώδικα.

```
LIBRARY ieee;

USE ieee.std_logic_1164.all;

ENTITY dec2to4Case IS
```

```
      PORT(s   : IN STD_LOGIC_VECTOR (1 DOWNTO 0);
           e   : IN STD_LOGIC;
           y   : OUT STD_LOGIC_VECTOR (3 DOWNTO 0));
END dec2to4Case;


ARCHITECTURE Behavior OF dec2to4Case IS
BEGIN
   PROCESS (s, e)
   BEGIN
     IF e = '1' THEN
       CASE s IS
         WHEN "00" => y <= "1000" ;
         WHEN "01" => y <= "0100" ;
         WHEN "10" => y <= "0010" ;
         WHEN OTHERS => y <= "0001" ;
     END CASE;
  ELSE
     y <= "0000" ;
  END IF ;
END PROCESS ;
END Behavior;
```

Σχήμα 2.59 Αποκωδικοποιητής 2 σε 4 με είσοδο επίτρεψης χρησιμοποιώντας διαδικαστική αντιστοίχιση VHDL.

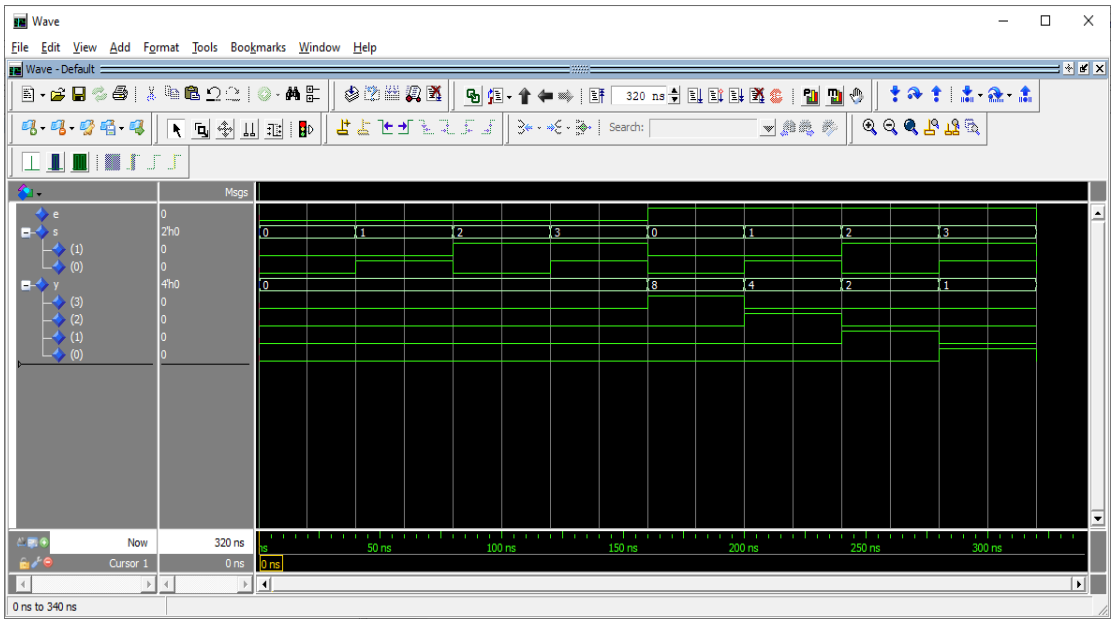
2.14.2 Προσομοίωση του αποκωδικοποιητή 2 σε 4 με είσοδο επίτρεψης με VHDL

Το παρακάτω Testbench είναι εκτελεί κάθε πιθανή είσοδο του αποκωδικοποιητή 2 σε 4 με είσοδο επίτρεψης. Τα αποτελέσματα είναι διακριτά από το αντίστοιχο Waveform διάγραμμα ανά 40 ns.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY dec2to4Casetb IS
END dec2to4Casetb;

ARCHITECTURE behavior OF dec2to4Casetb IS

SIGNAL e : STD_LOGIC;
SIGNAL s : STD_LOGIC_VECTOR(1 DOWNTO 0);
SIGNAL y : STD_LOGIC_VECTOR(3 DOWNTO 0);

COMPONENT dec2to4Case
PORT (e : IN  STD_LOGIC;
     s : IN  STD_LOGIC_VECTOR(1 DOWNTO 0);
     y : OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
END COMPONENT;

 BEGIN

m1: dec2to4Case PORT MAP(e=>e,s=>s,y=>y);

PROCESS
BEGIN

 e<='0';s<="00"; WAIT FOR 40 ns;
```

```
e<='0';s<="01"; WAIT FOR 40 ns;

e<='0';s<="10"; WAIT FOR 40 ns;

e<='0';s<="11"; WAIT FOR 40 ns;

e<='1';s<="00"; WAIT FOR 40 ns;

e<='1';s<="01"; WAIT FOR 40 ns;

e<='1';s<="10"; WAIT FOR 40 ns;

e<='1';s<="11"; WAIT FOR 40 ns;


 END PROCESS;

END behavior;
```

Σχήμα 2.60 Testbench αποκωδικοποιητή 2 σε 4 με είσοδο επίτρεψης VHDL.



Σχήμα 2.61 Προσομοίωση αποκωδικοποιητή 2 σε 4 με είσοδο επίτρεψης VHDL.

2.14.3 Περιγραφή του αποκωδικοποιητή 2 σε 4 με είσοδο επίτρεψης με Verilog

Η περιγραφή του αποκωδικοποιητή 2 σε 4 με είσοδο επίτρεψης χρησιμοποιώντας διαδικαστική αντιστοίχιση παρουσιάζεται παρακάτω στο σχήμα (Σχήμα 2.62).

```
module dec2_4 (s, y, e);// setting inputs and outputs
        input [1:0] s, e;
        output reg [3:0] y;
        always @(s or e) begin
                if (e) begin
                        if (s == 2'b00) begin
                                y = 4'b1000;
                        end
else if (s == 2'b01) begin
                                 y = 4'b0100;
end
                        else if (s == 2'b10) begin
                                 y = 4'b0010;
                          end
                        else
                        y = 4'b0001;
                        end
                else
                        y=0;
end
endmodule
```

Σχήμα 2.62 Αποκωδικοποιητής 2 σε 4 με είσοδο επίτρεψης χρησιμοποιώντας διαδικαστική αντιστοίχιση

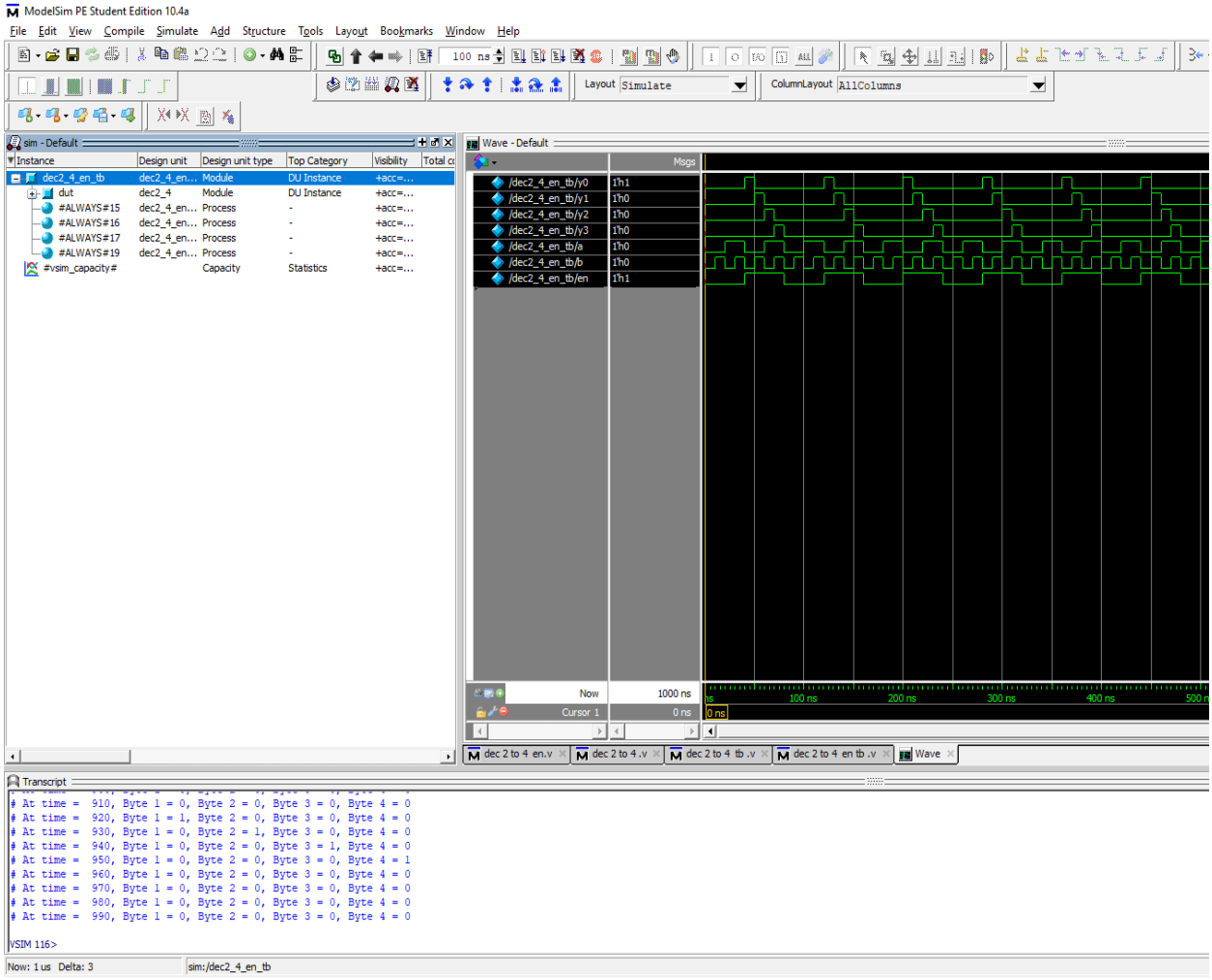
2.14.4 Προσομοίωση αποκωδικοποιητή 2 σε 4 με είσοδο επίτρεψης με Verilog

Ο αποκωδικοποιητής 2 σε 4 με είσοδο επίτρεψης δέχεται δύο εισόδους μήκους ενός bit και έχει τέσσερις εξόδους μήκους ενός bit. Επίσης δέχεται και την είσοδο επίτρεψης μήκους ενός bit. Παρατηρείται ότι ενώ η είσοδος επίτρεψης όμως έχει δηλωθεί με μήκος δύο bit δεν αλλάζει το τελικό αποτέλεσμα καθώς χρησιμοποιείται μόνο το λιγότερο σημαντικό από τα δύο. Οι είσοδοι αρχικοποιούνται στο μηδέν και οι τιμές μεταξύ τους εναλλάσσονται κάθε είκοσι και δέκα νανοδευτερόλεπτα ενώ της εισόδου επίτρεψης κάθε σαράντα. Ο πίνακας αληθείας του βρίσκεται στο σχήμα 2.43. Παρακάτω ακολουθούν το testbench και η προσομοίωση.

```
module dec2_4_en_tb();//setting inputs and outputs
        wire y0,y1,y2,y3;
        reg a, b, en;
dec2_4 dut(.y0(y0), .y1(y1), .y2(y2), .y3(y3), .a(a), .b(b), .en(en)); //initialization
        initial begin
                a <= 1'b0;
                b <= 1'b0;
```

```
                en <= 1'b0;
        end
        always #40 en=~en; // inputs change value at designated time in ns
        always #20 a=~a;
        always #10 b=~b;
always@(a or b or en) // for every change in the inputs the following message appears
$monitor("At time = %4d, Byte 1 = %d, Byte 2 = %d, Byte 3 = %d, Byte 4 = %d", $time, y0,
y1, y2, y3);
// message with extra info
endmodule
```

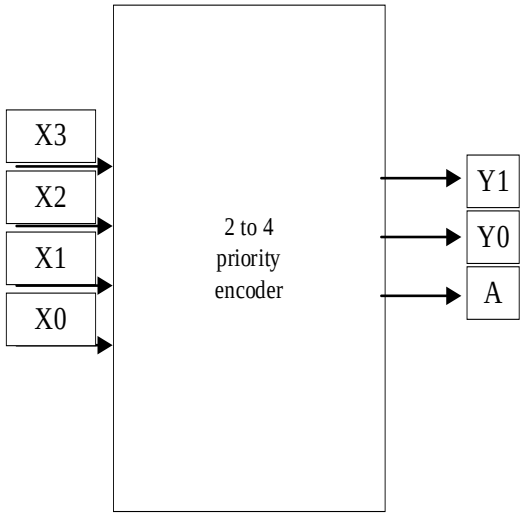
Σχήμα 2.63 Testbench αποκωδικοποιητή 2 σε 4 με είσοδο επίτρεψης



Σχήμα 2.64 Προσομοίωση αποκωδικοποιητή 2 σε 4 με είσοδο επίτρεψης

2.15 Περιγραφή του κωδικοποιητή προτεραιότητας 4 σε 2 με VHDL και Verilog

Στο σχήμα 2.65 δίνεται το λογικό σύμβολο και ο πίνακας αληθείας του κωδικοποιητή προτεραιότητας 4 σε 2.



X3	X2	X1	X0	Y1	Y0	A
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	X	0	1	1
0	1	X	X	1	0	1
1	X	X	X	1	1	1

Σχήμα 2.65 κωδικοποιητής προτεραιότητας 4 σε 2

2.15.1 Περιγραφή του κωδικοποιητή προτεραιότητας 4 σε 2 με VHDL

Παρακάτω απεικονίζεται ο κώδικας του κωδικοποιητή προτεραιότητας 4 σε 2 VHDL.

```
LIBRARY ieee;

USE ieee.std_logic_1164.all;


ENTITY prior_enc_4_to_2 IS

      PORT(x   : IN  STD_LOGIC_VECTOR (3 DOWNTO 0);

           y   : OUT STD_LOGIC_VECTOR (1 DOWNTO 0);

           z   : OUT STD_LOGIC);

END prior_enc_4_to_2;


ARCHITECTURE Behavioral OF prior_enc_4_to_2 IS

BEGIN

  PROCESS (x)

  BEGIN
```

```
      IF x(3) = '1' THEN
        y <= "11" ;
      ELSIF x(2) = '1' THEN
        y <= "10" ;
      ELSIF x(1) = '1' THEN
        y <= "01" ;
      ELSE
        y <= "00" ;
     END IF ;
  END PROCESS ;
   z <= '0' WHEN x = "0000" ELSE
        '1' ;
END Behavioral ;
```

Σχήμα 2.66 Περιγραφή του κωδικοποιητή προτεραιότητας 4 σε 2 με χρήση διαδικαστικής αντιστοίχισης VHDL.

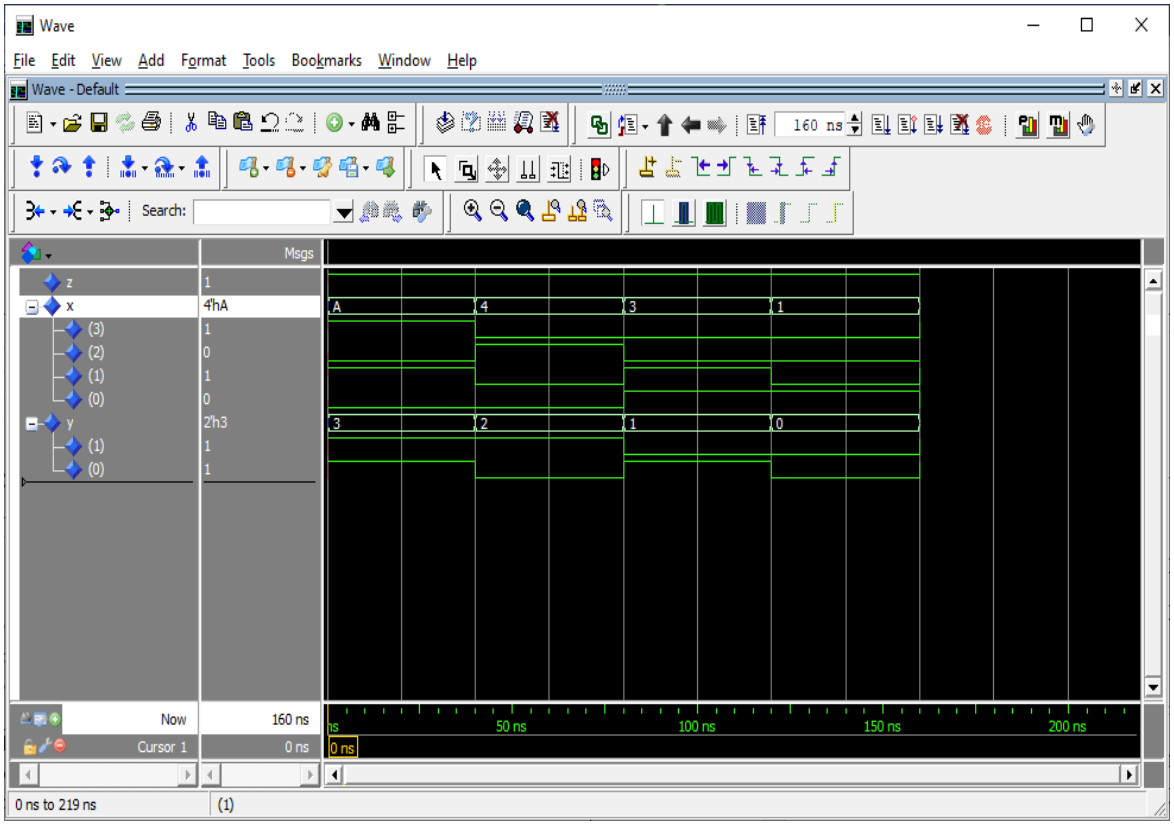
2.15.2 Προσομοίωση κωδικοποιητή προτεραιότητας 4 σε 2 με VHDL

Παρακάτω ακολουθεί προσομοίωση του κωδικοποιητή προτεραιότητας 4 σε 2 με VHDL. Η είσοδος αλλάζει ανά 40 ns με τα αποτελέσματα να είναι ορατά στο Waveform διάγραμμα (σχήμα 2.67).

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;


ENTITY prior_enc_4_to_2tb IS
END prior_enc_4_to_2tb;


ARCHITECTURE behavior OF prior_enc_4_to_2tb IS
```

```
SIGNAL z : STD_LOGIC;
SIGNAL y : STD_LOGIC_VECTOR(1 DOWNTO 0);
SIGNAL x : STD_LOGIC_VECTOR(3 DOWNTO 0);

COMPONENT prior_enc_4_to_2
PORT (x: IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
     z: OUT STD_LOGIC;
     y: OUT STD_LOGIC_VECTOR(1 DOWNTO 0));
END COMPONENT;

 BEGIN

m1: prior_enc_4_to_2 PORT MAP(x=>x,z=>z,y=>y);

PROCESS
BEGIN

 x<="1010"; WAIT FOR 40 ns;
 x<="0100"; WAIT FOR 40 ns;
 x<="0011"; WAIT FOR 40 ns;
 x<="0001"; WAIT FOR 40 ns;

 END PROCESS;
END behavior;
```

Σχήμα 2.67 Testbench κωδικοποιητή προτεραιότητας 4 σε 2 VHDL.



Σχήμα 2.68 Προσομοίωση κωδικοποιητή προτεραιότητας 4 σε 2 VHDL.

2.15.3 Περιγραφή του κωδικοποιητή προτεραιότητας 4 σε 2 με Verilog

Στο σχήμα 2.69 είναι η περιγραφή του με χρήση διαδικαστικής αντιστοίχισης της Verilog.

```
module priority_encoder(a, y); //setting output and inputs
 input [3:0]y;
 output reg [1:0]a;
always@(y) begin // if any input changes the following happens
        casex(y) // variable of case that treats x and z as don't care
                4'b0001:a = 2'b00;
                4'b001x:a = 2'b01;
                4'b01xx:a = 2'b10;
                4'b1xxx:a = 2'b11;
                default: $display("Error!");
        endcase
end
endmodule
```

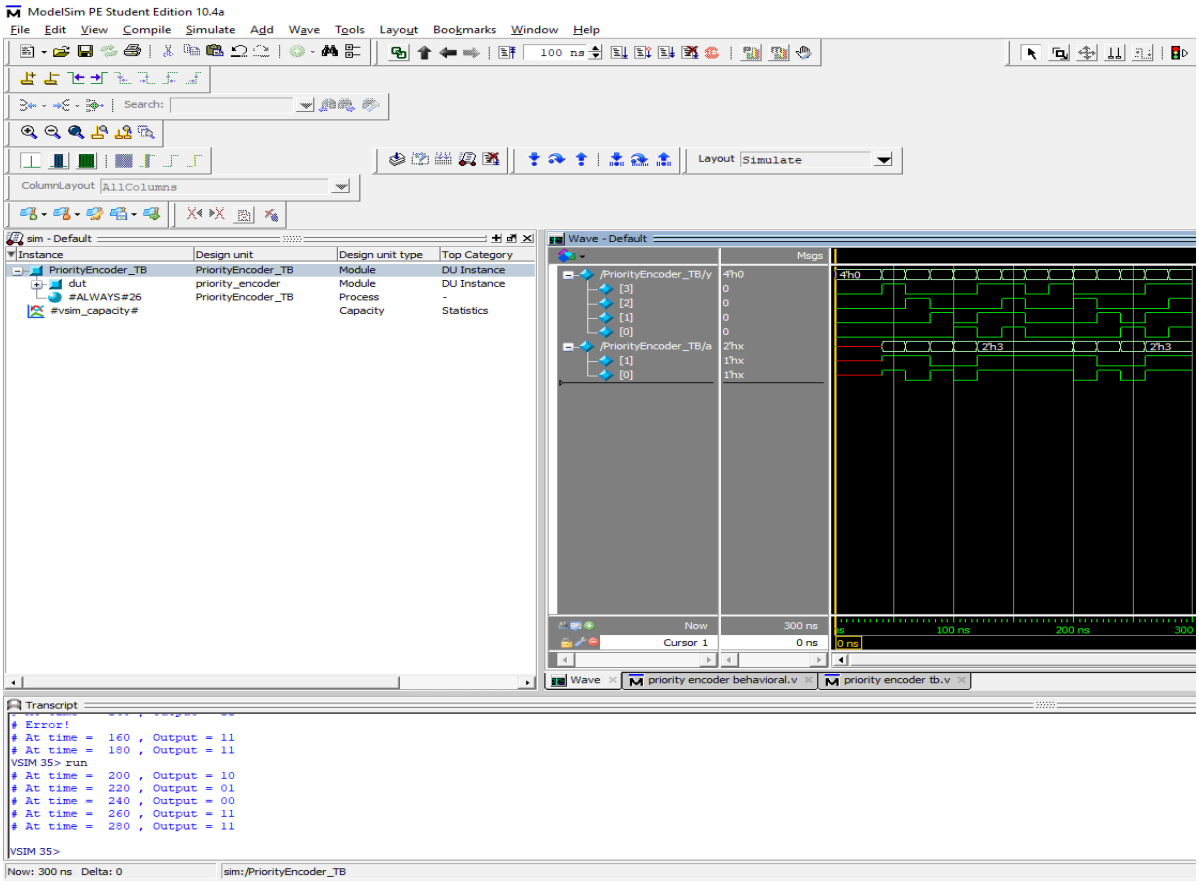
Σχήμα 2.69 Περιγραφή του κωδικοποιητή προτεραιότητας 4 σε 2 με χρήση διαδικαστικής αντιστοίχισης

2.15.4 Προσομοίωση κωδικοποιητή προτεραιότητας 4 σε 2 με Verilog

Ο κωδικοποιητής προτεραιότητας είναι ένα κύκλωμα με τέσσερις εισόδους. Ωστόσο σε αυτή την περιγραφή δίνονται με μια μεταβλητή των τεσσάρων bit. Οι είσοδοι είναι του ενός bit και αρχικοποιούνται στο μηδέν. Σε περίπτωση που έχει δύο και παραπάνω εισόδους ενεργοποιημένες, η έξοδος θα καθοριστεί από την είσοδο με την μεγαλύτερη προτεραιότητα. Ο πίνακας αληθείας βρίσκεται στο σχήμα 2.65. Παρακάτω ακολουθούν το testbench και η προσομοίωση.

```
module PriorityEncoder_TB; //setting output and inputs
        reg [3:0] y;
        wire [1:0] a;
        priority_encoder dut (.y(y), .a(a));  // instantiation by port name.
        initial begin // initialization
                y = 0;
                #20 y = 4'b0000; //certain values are used after 20ns each
                #20 y = 4'b1000;
                 #20 y = 4'b0100;
                 #20 y = 4'b0010;
                  #20 y = 4'b0001;
                  #20 y = 4'b1010;
                  #20 y = 4'b1111;
        end
        always@(y) // if any input changes the message appears
$monitor("At time = %4d , Output = %b", $time, a);
endmodule
```

Σχήμα 2.70 Testbench κωδικοποιητή προτεραιότητας 4 σε 2



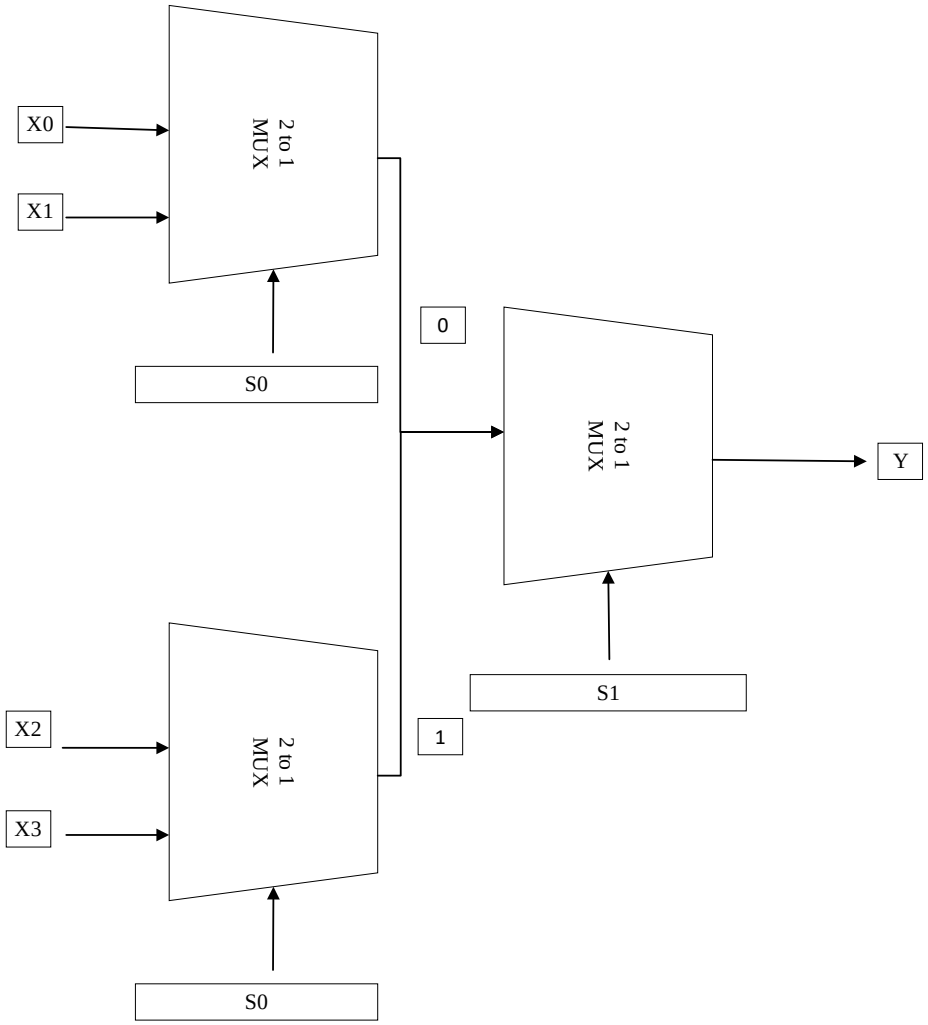
Σχήμα 2.71 Προσομοίωση κωδικοποιητή προτεραιότητας 4 σε 2

2.16 Δομική (structural) σχεδίαση συνδυαστικών κυκλωμάτων με VHDL και Verilog

Τα προηγούμενα παραδείγματα ήταν είτε βάση συμπεριφοράς του κυκλώματος (behavioral), είτε βάση της λογικής παράστασης τους ως ροή δεδομένων (dataflow). Αυτό έχει ως αποτέλεσμα να μην υλοποιούνται ως κυκλώματα. Παράλληλα, ένας άλλος τρόπος υλοποίησης που περιγράφτηκε παραπάνω είναι η μοντελοποίηση με βάση τη δομή. Ωστόσο, το παράδειγμα που αναπτύχθηκε, συμπεριελάμβανε μόνο απλές λογικές πύλες. Έτσι είναι δυνατό να δημιουργηθούν κυκλώματα και να συμπεριληφθούν στον κυρίως κώδικα ενός άλλου μεγαλύτερου κυκλώματος.

2.16.1 Δομική (structural) σχεδίαση ενός πολυπλέκτη 4 σε 1 με χρήση πολυπλεκτών 2 σε 1 με VHDL και Verilog

Στο παρακάτω σχήμα περιγράφεται η δομική σχεδίαση πολυπλέκτη 4 σε 1 με χρήση πολυπλεκτών 2 σε 1.



Σχήμα 2.72 Σχεδίαση πολυπλέκτη 4 σε 1 με χρήση πολυπλεκτών 2 σε 1

2.16.2 Δομική (structural) σχεδίαση ενός πολυπλέκτη 4 σε 1 με χρήση πολυπλεκτών 2 σε 1 με VHDL

Η δομική σχεδίαση ενός πολυπλέκτη 4 σε 1 με χρήση πολυπλεκτών 2 σε 1 μπορεί να επιτευχθεί με την δημιουργία κυκλώματος ως βιβλιοθήκη. Στο σχήμα 2.50 υπάρχει η υλοποίηση του πολυπλέκτη 2 σε 1. Παρακάτω ακολουθεί η υλοποίηση της βιβλιοθήκης πολυπλέκτη 2 σε 1.

```
LIBRARY ieee;

USE ieee.std_logic_1164.all;



PACKAGE Mux2to1_package IS

        COMPONENT Mux2to1

                PORT (x0, x1, s : IN  STD_LOGIC;
```

```
                    z : OUT STD_LOGIC);
    END COMPONENT;
END Mux2to1_package;
```

Σχήμα 2.73 Βιβλιοθήκη πολυπλέκτη 2 σε 1 VHDL.

Όπως φαίνεται στο ακόλουθο σχήμα γίνεται η χρήση της βιβλιοθήκης με τις εντολές «LIBRARY work;» και την «USE work.Mux2to1_package.all;»

```
LIBRARY ieee;
LIBRARY work;
USE ieee.std_logic_1164.all;
USE work.Mux2to1_package.all;

ENTITY Mux_4_to_1_const IS
PORT (x0, x1, x2, x3 : IN   STD_LOGIC;
        s0, s : IN   STD_LOGIC;
           z : OUT  STD_LOGIC);
END Mux_4_to_1_const;

ARCHITECTURE behavior OF Mux_4_to_1_const IS
    SIGNAL z1, z2 : STD_LOGIC;
BEGIN
    MUX1: Mux2to1 PORT MAP (x0,x1,s0,z1);
    MUX2: Mux2to1 PORT MAP (x2,x3,s0,z2);
    MUX3: Mux2to1 PORT MAP (z1,z2,s,z);
END behavior;
```

Σχήμα 2.74 Δομική σχεδίαση του πολυπλέκτη 4 σε 1 με χρήση 2 σε 1 VHDL.

2.16.3 Δομική (structural) σχεδίαση ενός πολυπλέκτη 4 σε 1 με χρήση πολυπλεκτών 2 σε 1 με Verilog

Στο σχήμα 2.75 δίνεται η δομική σχεδίαση με Verilog του πολυπλέκτη 4 σε 1, με χρήση σαν στοιχείο σχεδίασης τον πολυπλέκτη 2 σε 1. Για να γίνει πιο σαφές, η υλοποίηση ξεκινάει με την δήλωση δυο διαφορετικών module, ένα για την υλοποίηση του κυρίως κυκλώματος πολυπλέκτη 4 σε 1 και ένα για το υποκύκλωμα του πολυπλέκτη 2 σε 1. Το module του υποκυκλώματος δομείται με τη χρήση των βασικών πυλών, που χρειάζονται για την υλοποίηση. Οι πύλες αυτές είναι ενσωματωμένες στη Verilog, αλλά μπορούμε να τις υλοποιήσουμε και μεμονωμένα. Αρχικά, εισάγεται η έξοδος, και στη συνέχεια οι είσοδοι στον ορισμό της πύλης. Επίσης, πρέπει να οριστούν κάποιες μεταβλητές «wire» για την μεταξύ τους επικοινωνία. Έπειτα, χρησιμοποιείται η ίδια λογική και στο κυρίως κύκλωμα, αλλά αντί για τις πύλες χρησιμοποιούμε ως βάση το υποκύκλωμα που έχουμε φτιάξει.

```
module mux4to1_using_mux2to1(output y, input x0,x1,ix2,ix3,s1,s0);
        wire mux1,mux2;
        mux2to1 mux_1 (mux1,x0, x1, s1);
        mux2to1 mux_2 (mux2, x2, x3,s1);
        mux2to1 mux_3(y, mux1, mux2, s0);
endmodule
module mux2to1(output out, input A, B, Sel);
        wire and_A,and_B,Selout;
        not ( Selout, Sel);
        and (and_A, A, Selout), (and_B, B, Sel);
        or (out, and_A, and_B);
endmodule
```

Σχήμα 2.75 Δομική σχεδίαση του πολυπλέκτη 4 σε 1 με χρήση 2 σε 1

2.17 Διαδικαστικές Εντολές

2.17.1 Εντολή always

Η διαδικαστική εντολή «always» της Verilog, θέτει ένα μπλοκ στο εσωτερικό του οποίου ο κώδικας τρέχει συνέχεια κατά την διάρκεια της προσομοίωσης. Για να ενεργοποιηθεί έχει μια συγκεκριμένη λίστα ευαισθησίας (sensitivity list), όπου αν πραγματοποιηθούν αλλαγές οπουδήποτε σε αυτήν, ενεργοποιείται το μπλοκ. Επίσης, άξιο αναφοράς είναι ότι αν η εντολή χρησιμοποιηθεί χωρίς λίστα ευαισθησίας, τότε ο κώδικας του μπλοκ της «τρέχει» συνέχεια κατά την προσομοίωση. Βρόγχοι if-else, case και βρόγχοι επανάληψης μπορούν να λειτουργήσουν μόνο μέσα σε μπλοκ always.

```
always @ (sensitivity list) begin
if statements
case statements
loop statements
end
```
Σχήμα 2.76 Επεξήγηση μπλοκ «always»

2.17.2 Εντολή PROCESS

Παρόμοια με την εντολή «always» είναι η χρήση της εντολής PROCESS για την VHDL. Αναλυτικότερα, η χρήση της PROCESS επιτρέπει στα εκάστοτε statements να «τρέχουν» διαδοχικά. Επιπλέον, χρησιμοποιούνται μεταβλητές ως κάτοχοι θέσεων (Place Holders), ώστε να λαμβάνουν τις εντολές που τους εκχωρούνται άμεσα. Παρακάτω παρατίθενται κάποιοι από τους βρόχους που μπορούν να χρησιμοποιηθούν σε μια PROCESS.

```
BEGIN
    PROCESS (PLACE HOLDERS)
    BEGIN
IF statements
CASE statements
WITH/SELECT statements
WHEN/SELECT statements
FOR LOOP statements
WHILE LOOP statements
```
Σχήμα 2.77 Επεξήγηση εντολής «PROCESS»

2.18 Περιγραφή συνδυαστικών κυκλωμάτων με χρήση της if-else με VHDL και Verilog

Ακολουθεί η μορφή της εντολής IF στη VHDL γλώσσα.

```
IF (προϋπόθεση) THEN                --Μόνο μία προϋπόθεση--
    --κώδικας σε VHDL--
END
```

```
IF (προϋπόθεση 1) THEN                --Παραπάνω από μία προϋπόθεση--
        --κώδικας σε VHDL--
ELSIF(προϋπόθεση 2) THEN
        --κώδικας σε VHDL--
END


IF (προϋπόθεση) THEN                  --Μόνο μία προϋπόθεση και κάλυψη
        --κώδικας σε VHDL--           υπολοίπων καταστάσεων χωρίς όρους--
ELSE
        --κώδικας σε VHDL--
END
```

Σχήμα 2.78 Επεξήγηση βρόγχου «if-else» VHDL

Η γενική μορφή της εντολής είναι για την Verilog ακολουθεί παρακάτω.

```
 if (expression) begin          */ Μόνο μια εντολή if
        statements;                     if (expression)
 end                                            statements; /*
 else if (expression) begin
        statements;                     */ Μόνο μια if-else
 end                                    if (expression)
else begin                              statements;
    statements;                         else
end                                     statements; */
```

Σχήμα 2.79 Επεξήγηση βρόγχου «if-else» Verilog

2.18.1 Περιγραφή πολυπλέκτη 2 σε 1

Ακολουθεί περιγραφή του πολυπλέκτη 2 σε 1 με χρήση εντολής if σε VHDL.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;


ENTITY Mux_2_to_1 IS
   PORT (x0, x1, s : IN  STD_LOGIC;
```

```
            f : OUT STD_LOGIC);
END Mux_2_to_1;


ARCHITECTURE Behavioral OF mux_2_to_1 IS
BEGIN
  PROCESS ( x0, x1, s )
  BEGIN
    IF s = '0' THEN f <= x0 ;
    ELSE
       f <= x1 ;
    END IF ;
  END PROCESS ;
END Behavioral;
```

Σχήμα 2.80 Περιγραφή πολυπλέκτη 2 σε 1 με εντολή if-else με VHDL.

Ακολουθεί περιγραφή του πολυπλέκτη 2 σε 1 με χρήση εντολής if σε Verilog.

```
module mux_2to1(input X0, X1, S, output reg Z);  //setting inputs and output
always@(X0 or X1 or S) begin
if (S)
        Z = X1;
else
        Z = X0;
end
endmodule
```

Σχήμα 2.81 Περιγραφή πολυπλέκτη 2 σε 1 με εντολή if-else με Verilog

Παρατηρείται ότι η έξοδος Z έχει δηλωθεί ως reg. Αυτό έχει γίνει για να μπορεί να αποθηκεύει την τιμή που της δίνεται.

2.18.2 Περιγραφή του κωδικοποιητή προτεραιότητας 4 σε 2 με VHDL και Verilog

Παρακάτω δίνεται η περιγραφή του κωδικοποιητή προτεραιότητας 4 σε 2 με χρήση της εντολής if-else.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;


ENTITY prior_enc_4_to_2 IS
      PORT(x   : IN  STD_LOGIC_VECTOR (3 DOWNTO 0);
           y   : OUT STD_LOGIC_VECTOR (1 DOWNTO 0);
           z   : OUT STD_LOGIC);
END prior_enc_4_to_2;
ARCHITECTURE Behavioral OF prior_enc_4_to_2 IS
BEGIN
  PROCESS (x)
  BEGIN
     IF x(3) = '1' THEN
       y <= "11" ;
     ELSIF x(2) = '1' THEN
       y <= "10" ;
     ELSIF x(1) = '1' THEN
       y <= "01" ;
     ELSE
       y <= "00" ;
    END IF ;
  END PROCESS ;
   z <= '0' WHEN x = "0000" ELSE
        '1' ;
END Behavioral ;
```

Σχήμα 2.82 Περιγραφή του κωδικοποιητή προτεραιότητας 4 σε 2 με χρήση εντολής if με VHDL.

Παρακάτω δίνεται η περιγραφή του κωδικοποιητή προτεραιότητας 4 σε 2 με χρήση της εντολής if-else.

```
module priority_encoder_if(a, y, clk); //setting output and inputs
        input [3:0]y;
        input clk;
        output reg [1:0]a;
        always@(y) begin // if any input changes the following happens
                if (y[3])
                        a <= 2'b00;
                else if (y[2])
                        a <= 2'b01;
                else if (y[1])
                        a <= 2'b10;
                else if(y[0])
                        a <= 2'b11;
                else
                        a <= 2'b00;
        end
endmodule
```

Σχήμα 2.83 Περιγραφή του κωδικοποιητή προτεραιότητας 4 σε 2 με χρήση εντολής if με Verilog

2.19 Περιγραφή κυκλωμάτων με χρήση εντολής case με VHDL και Verilog

Η γενική μορφή της VHDL case είναι η ακόλουθη

```
CASE (μεταβλητή) IS

WHEN statement => (μεταβλητή 1) <=

WHEN OTHERS statement => (μεταβλητή 2) <= (εκχώρηση) ;

END CASE;
```

Σχήμα 2.84 Επεξήγηση εντολής «case» VHDL

Η γενική μορφή της Verilog case είναι η ακόλουθή:

```
case(expression)
        value: statement;
        value: statement;
endcase
```

Σχήμα 2.85 Επεξήγηση εντολής «case» Verilog

Οι εντολές casex και casez υλοποιούνται με τον ακριβώς ίδιο τρόπο. Η μόνη διαφορά είναι ότι η casez επιτρέπει και τη χρήση μιας μεταβλητής ακόμα και αν περιέχει Z (υψηλή αντίσταση) ως μεταβλητή “don’t care”, ενώ η casex επιτρέπει τη χρήση μιας μεταβλητής ακόμα και αν περιέχει X και Z με αντίστοιχο τρόπο.

2.19.1 Περιγραφή του πολυπλέκτη 2 σε 1

Ακολουθεί η περιγραφή του πολυπλέκτη 2 σε 1 με χρήση εντολής case με VHDL.

```
LIBRARY ieee;

USE ieee.std_logic_1164.all;



ENTITY mux2to1Case IS

   PORT (x0, x1, s : IN  STD_LOGIC;

               z : OUT STD_LOGIC);

END mux2to1Case;



ARCHITECTURE Behavior OF mux2to1Case IS

BEGIN

   PROCESS ( x0, x1, s )

   BEGIN

      CASE s IS

         WHEN '0' => z <= x0 ;

         WHEN OTHERS => z <= x1 ;

      END CASE;

   END PROCESS;

END Behavior;
```

Σχήμα 2.86 Περιγραφή πολυπλέκτη 2 σε 1 με χρήση εντολής case με VHDL.

Ακολουθεί η περιγραφή του πολυπλέκτη 2 σε 1 με χρήση εντολής case με Verilog.

```
module mux_2to1_using_case(input X0, X1, S, output reg Z);
always@(X0 or X1 or S) begin
case (S)
          1'b0 : Z <= X0;
          1'b1 : Z <= X1;
endcase
end
endmodule
```

Σχήμα 2.87 Περιγραφή πολυπλέκτη 2 σε 1 με χρήση εντολής case με Verilog

2.19.2 Περιγραφή του αποκωδικοποιητή 2 σε 4

Στο παρακάτω σχήμα δίνεται η περιγραφή του αποκωδικοποιητή 2 σε 4 με χρήση εντολής if και case εμφωλευμένη μέσα στην if σε VHDL.

```
LIBRARY ieee;

USE ieee.std_logic_1164.all;


ENTITY dec2to4Case IS

      PORT(s   : IN STD_LOGIC_VECTOR (1 DOWNTO 0);

           e   : IN STD_LOGIC;

           y   : OUT STD_LOGIC_VECTOR (3 DOWNTO 0));

END dec2to4Case;


ARCHITECTURE Behavior OF dec2to4Case IS

BEGIN

   PROCESS (s, e)

   BEGIN

     IF e = '1' THEN

       CASE s IS

        WHEN "00" => y <= "1000" ;

        WHEN "01" => y <= "0100" ;

        WHEN "10" => y <= "0010" ;

        WHEN OTHERS => y <= "0001" ;

     END CASE;

  ELSE

     y <= "0000" ;

  END IF ;

END PROCESS ;
```

```
END Behavior;
```

Σχήμα 2.88 περιγραφή του αποκωδικοποιητή 2 σε 4 με χρήση case και if με VHDL.

Αντίστοιχα, ο παρακάτω κώδικας περιγράφει το ίδιο σε Verilog.

```
module dec2_4 (s, y, e);
        input [1:0] wire s, e;
        output [3:0] reg y;
        reg [2:0] z;
        always @(s or e) begin
                z <= s and e;
                if (e)
                        casex (z)
                                3'b100 : y <= 4'b1000;
                                3'b101 : y <= 4'b0100;
                                3'b110 : y <= 4'b0010;
                                3'b111 : y <= 4'b0001;
                        endcase
                else
                        y<=0;
        end
endmodule
```

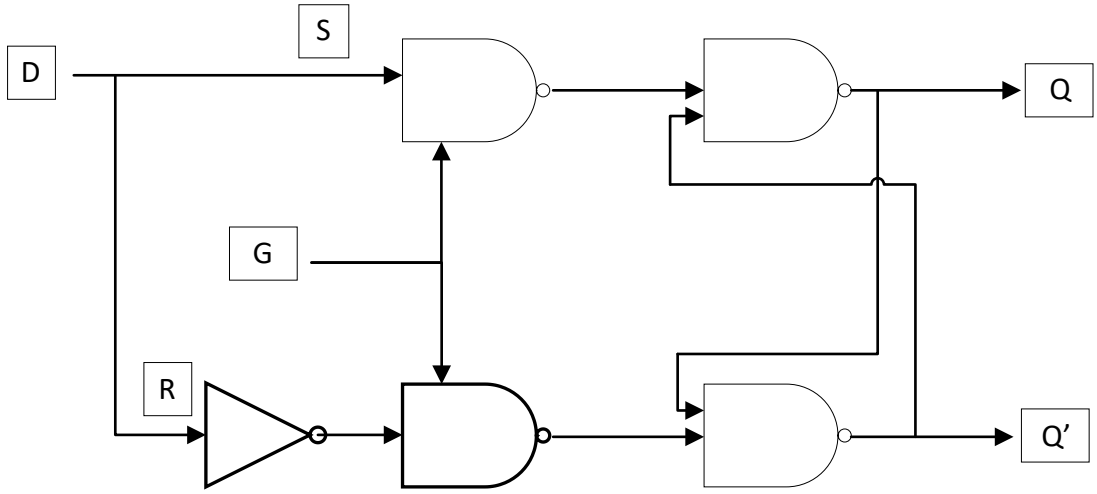
Σχήμα 2.89 περιγραφή του αποκωδικοποιητή 2 σε 4 με χρήση case και if με Verilog

2.20 Περιγραφή ακολουθιακών κυκλωμάτων με τις γλώσσες VHDL και Verilog

Παραπάνω έγιναν αναφορές στα συνδυαστικά κυκλώματα που αποτελούν έναν από τους δύο τρόπους υλοποίησης ψηφιακών κυκλωμάτων. Στην ενότητα αυτή, παρουσιάζεται πληθώρα ακολουθιακών κυκλωμάτων, όπως τα φλιπ-φλοπ και οι καταχωρητές. Τα ακολουθιακά κυκλώματα έχουν την ιδιότητα να γνωρίζουν προηγούμενες καταστάσεις ή τιμές εισόδου του κυκλώματος. Σε αντίθεση με τα συνδυαστικά κυκλώματα περιέχουν και στοιχεία μνήμης. Επίσης οι έξοδοι του μπορούν να χρησιμοποιηθούν εκ νέου ως τιμές εισόδου στο κύκλωμα, πάλι σε αντίθεση με τα συνδυαστικά όπου, οι έξοδοι είναι μόνο μιας συγκεκριμένης χρονικής στιγμής. Διαχωρίζονται σε δύο κατηγόριες βάσει του πως αλληλεπιδρούν με κύκλωμα ρολογιού, τα σύγχρονα και τα ασύγχρονα. Τα σύγχρονα ακολουθιακά κυκλώματα αλληλεπιδρούν με το ρολόι ανά συγκεκριμένα χρονικά διαστήματα επιτυγχάνοντας μια μορφή συγχρονισμού. Τα ασύγχρονα ακολουθιακά κυκλώματα μπορούν να έχουν αλλαγές οποιαδήποτε στιγμή χωρίς κάποιο συγκεκριμένο χρονισμό. Ακολουθούν η περιγραφή του μανδαλωτή D (D latch) και των ακμοπυροδούμενων φλιπ-φλοπ με την Verilog. Έπειτα θα υλοποιηθούν κυκλώματα καταχωρητών και απαριθμητών.

2.20.1 Περιγραφή του μανδαλωτή D Latch με χρήση VHDL και Verilog

Η περιγραφή του D Latch δίνεται στο παρακάτω σχήμα. Οι δυο είσοδοι του είναι, μια D, μια G και μια έξοδος Q. Η έξοδος ορίζεται με χρήση της εντολής if-else. Η είσοδος G λειτουργεί σαν είσοδος επίτρεψης δηλαδή, όταν είναι θετική τότε η έξοδος Q λαμβάνει την τιμή της εισόδου D. Σε περίπτωση που δεν είναι θετική η τιμή της εξόδου Q παραμένει η ίδια.



Σχήμα 2.90 Δομή ενός D-Latch

2.20.1.1 Περιγραφή του μανδαλωτή D Latch με χρήση VHDL

Παρακάτω περιγράφεται ο μανδαλωτής D Latch με χρήση VHDL κώδικα.

```
LIBRARY ieee;

USE ieee.std_logic_1164.all;

ENTITY D_latch IS

    PORT (D, G : IN  STD_LOGIC;

          Q : OUT STD_LOGIC);

END D_latch;


ARCHITECTURE behavior OF D_latch IS

BEGIN

     PROCESS (D, G)

     BEGIN

     IF G='1' THEN
```

```
        Q<=D;
    END IF;
    END PROCESS;
END Behavior;
```

Σχήμα 2.91 Περιγραφή D Latch με VHDL.

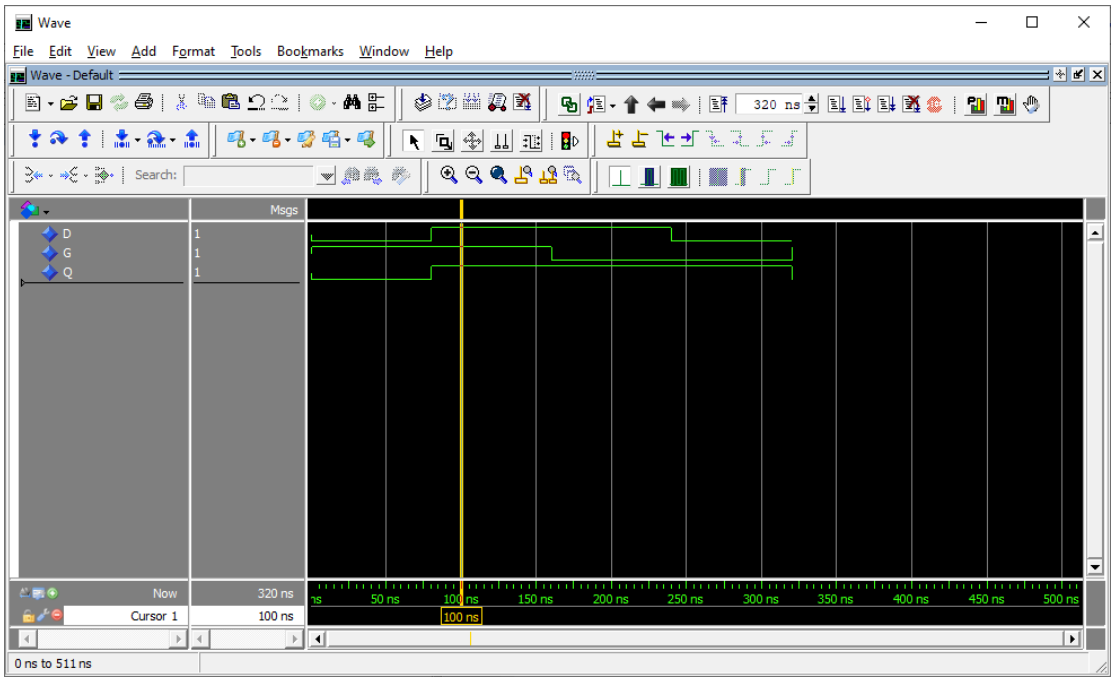
2.20.1.2 Προσομοίωση μανδαλωτή D Latch με VHDL

Ακολουθεί το Testbench που προσομοιώνει τον μανδαλωτή D Latch. Στην προσομοίωση λαμβάνονται όλες οι πιθανές τιμές εισόδου κάθε 80 ns. Τέλος, τα αποτελέσματα απεικονίζονται στο αντίστοιχο διάγραμμα Waveform.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY D_latchtb IS
END D_latchtb;

ARCHITECTURE behavior OF D_latchtb IS

SIGNAL  D, G, Q : STD_LOGIC;

COMPONENT D_latch
PORT (D, G : IN  STD_LOGIC;
      Q : OUT STD_LOGIC);
END COMPONENT;

 BEGIN

m1: D_latch PORT MAP(G=>G,D=>D,Q=>Q);
```

```
PROCESS
BEGIN

 G<='1';D<='0';WAIT FOR 80 ns;
 G<='1';D<='1';WAIT FOR 80 ns;
 G<='0';D<='1';WAIT FOR 80 ns;
 G<='0';D<='0';WAIT FOR 80 ns;

 END PROCESS;
END behavior;
```

Σχήμα 2.92 Testbench μανδαλωτή D Latch VHDL.



Σχήμα 2.93 Προσομοίωση μανδαλωτή D Latch VHDL.

2.20.1.3 Περιγραφή του μανδαλωτή D Latch με χρήση Verilog

Η εντολή if-else, στην οποία γίνεται ο υπολογισμός, τοποθετείται σε μπλοκ always με λίστα ευαισθησίας τις εισόδους D και Q. Παρακάτω ακολουθεί η περιγραφή χρησιμοποιώντας Verilog.

```
module d_latch (input d, g, output reg Q); //setting reg output and inputs
always @(g or d) begin //for every change in input the following happens
        if (g)
                Q <= d;
        end
endmodule
```

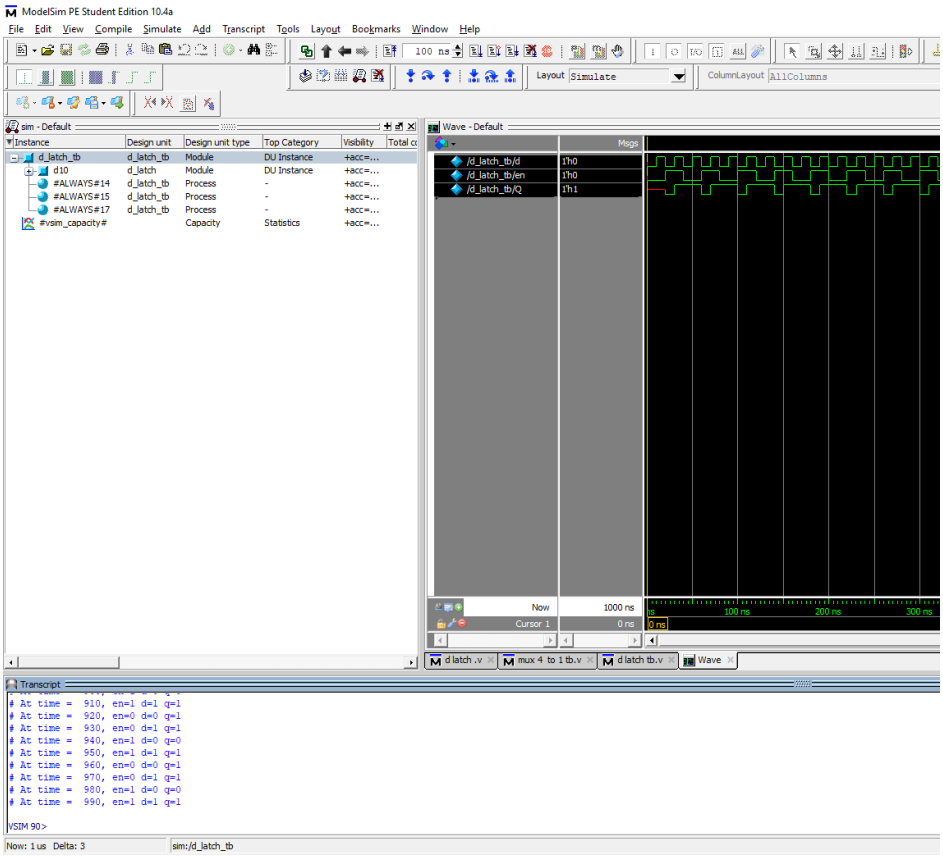
Σχήμα 2.94 Περιγραφή D Latch με Verilog

2.20.1.4 Προσομοίωση μανδαλωτή D Latch με Verilog

Στο κύκλωμα του D Latch υπάρχουν δύο είσοδοι και μια έξοδος. Η μία είσοδος αποτελεί τα σήματα δυαδικών ψηφιών που εισέρχονται στο κύκλωμα και η άλλη αποτελεί είσοδο επίτρεψης. Οι είσοδοι είναι του ενός bit και αρχικοποιούνται στο μηδέν. Η έξοδος είναι το σήμα που εισέρχεται στο κύκλωμα με δεδομένο ότι είναι ενεργοποιημένη η είσοδος επίτρεψης. Ακολουθούν το testbench και η προσομοίωση.

```
module d_latch_tb; //setting inputs and output

  reg d, en;

  wire Q;

  d_latch d10 (.d (d), .en(en), .Q(Q)); // instantiation by port name.

  initial begin // initialization

        d = 1'b0;

        en <= 1'b0;

  end

  always #10 d=~d; // every set time in ns the value of inputs changes

  always #20 en=~en;

  always@(d or en) // if any input changes the message appears

$monitor ("At time = %4d, en=%0b d=%0b q=%0b", $time, en ,d ,Q); // message for extra info

endmodule
```

Σχήμα 2.95 Testbench μανδαλωτή D Latch

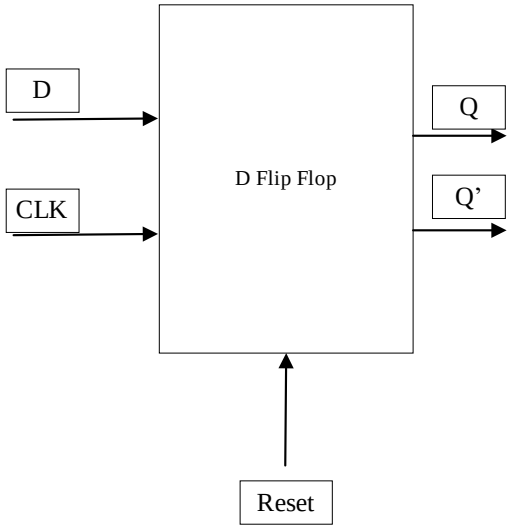


Σχήμα 2.96 Προσομοίωση μανδαλωτή D Latch

2.21 Περιγραφή φλιπ-φλοπ με χρήση VHDL και Verilog

2.21.1 Περιγραφή θετικά ακμοπυροδοτούμενου D φλιπ-φλοπ με ασύγχρονη είσοδο μηδενισμού με χρήση VHDL και Verilog

Στο παρακάτω σχήμα 2.97 παρουσιάζεται το λογικό σύμβολο και ο πίνακας αληθείας του D φλιπ-φλοπ που πυροδοτείται με τις θετικές ακμές παλμών του ρολογιού και διαθέτει ασύγχρονη είσοδο μηδενισμού που ενεργοποιείται με 0.



Reset	D	Q'	Q
0	x	x	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Σχήμα 2.97 Θετικά ακμοπυροδοτούμενο D φλιπ-φλοπ με ασύγχρονη είσοδο μηδενισμού

2.21.1.1 Περιγραφή θετικά ακμοπυροδοτούμενου D φλιπ-φλοπ με ασύγχρονη είσοδο μηδενισμού με χρήση VHDL

Ακολουθεί η περιγραφή του θετικά ακμοπυροδοτούμενου D φλιπ-φλοπ με ασύγχρονη είσοδο μηδενισμού σε VHDL.

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


ENTITY D_flip_flopA IS
 PORT (D, Clr, Clk : IN  STD_LOGIC;
              Q : OUT STD_LOGIC);
END D_flip_flopA;


ARCHITECTURE behavior OF D_flip_flopA IS
BEGIN
     PROCESS (Clr, Clk)
     BEGIN
        IF Clr= '0' THEN
          Q<='0';
        ELSIF Clk'EVENT AND Clk='1' THEN
          Q<=D;
       END IF;


     END PROCESS;
END Behavior;
```

Σχήμα 2.98 Περιγραφή με VHDL του θετικά ακμοπυροδοτούμενου D φλιπ-φλοπ με ασύγχρονη είσοδο μηδενισμού

2.21.1.2 Περιγραφή θετικά ακμοπυροδοτούμενου D φλιπ-φλοπ με ασύγχρονη είσοδο μηδενισμού με χρήση Verilog

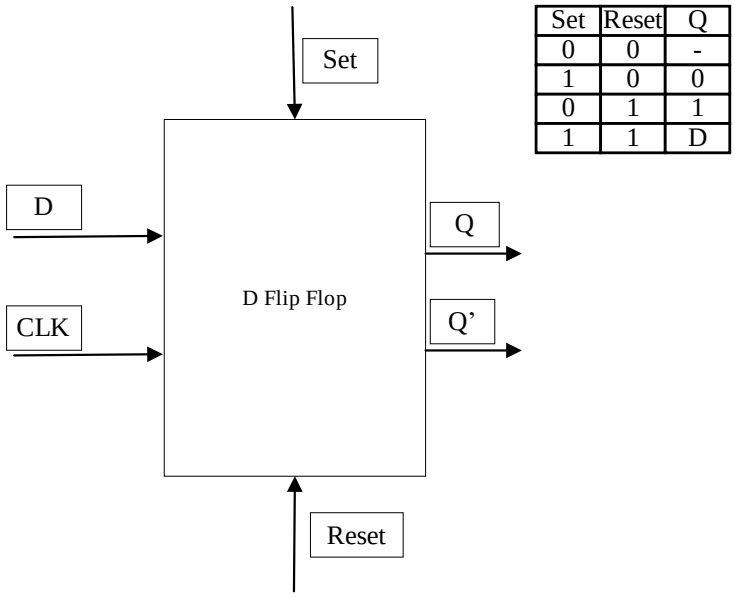
Παρακάτω ακολουθεί η περιγραφή του D φλιπ-φλοπ με χρήση Verilog.

```
module dff_reset (D, clk, reset, Q);//setting reg output and inputs
        input clk, D, reset;
        output reg Q;

        always@(posedge clk)// whenever clock is 1 this happens
        begin
                if (reset == 0)
                        Q <= 1'b0;
                else
                        Q <= D;
        end
endmodule
```

Σχήμα 2.99 Περιγραφή με Verilog του θετικά ακμοπυροδοτούμενου D φλιπ-φλοπ με ασύγχρονη είσοδο μηδενισμού

2.21.2 Περιγραφή θετικά ακμοπυροδοτούμενου D φλιπ-φλοπ με ασύγχρονες εισόδους μηδενισμού και θέσης με χρήση VHDL και Verilog

Παρακάτω στο σχήμα 2.100 δίνεται το λογικό σύμβολο και ο χαρακτηριστικός πίνακας του D φλιπ-φλοπ με πυροδότηση στις θετικές ακμές των παλμών του ρολογιού και διάθεση ασύγχρονων εισόδων μηδενισμού και θέσης με ενεργοποίηση στο 0.



Set	Reset	Q
0	0	-
1	0	0
0	1	1
1	1	D

Σχήμα 2.100 Θετικά ακμοπυροδοτούμενου D φλιπ-φλοπ με ασύγχρονες εισόδους μηδενισμού και θέσης

2.21.2.1 Περιγραφή θετικά ακμοπυροδοτούμενου D φλιπ-φλοπ με ασύγχρονες εισόδους μηδενισμού και θέσης με χρήση VHDL

Ο παρακάτω κώδικας σε VHDL περιγράφει τον θετικά ακμοπυροδοτούμενου D φλιπ-φλοπ με ασύγχρονες εισόδους μηδενισμού και θέσης

```
LIBRARY ieee;

USE ieee.std_logic_1164.ALL;

ENTITY D_flip_flop IS

    PORT (D, Clr, Prt, Clk : IN     STD_LOGIC;

                Q   : OUT  STD_LOGIC);

END D_flip_flop;


ARCHITECTURE behavioral OF D_flip_flop IS

BEGIN

      PROCESS (Clr , Prt, Clk)

      BEGIN

         IF Clr= '0' THEN  Q<='0';

         ELSIF Prt='0' THEN  Q<='1';

         ELSIF Clk'EVENT AND Clk='1' THEN

           Q<=D;

        END IF;

      END PROCESS;

END Behavioral;
```

Σχήμα 2.101 Περιγραφή θετικά ακμοπυροδοτούμενου D φλιπ-φλοπ με ασύγχρονες εισόδους μηδενισμού και θέσης με χρήση VHDL

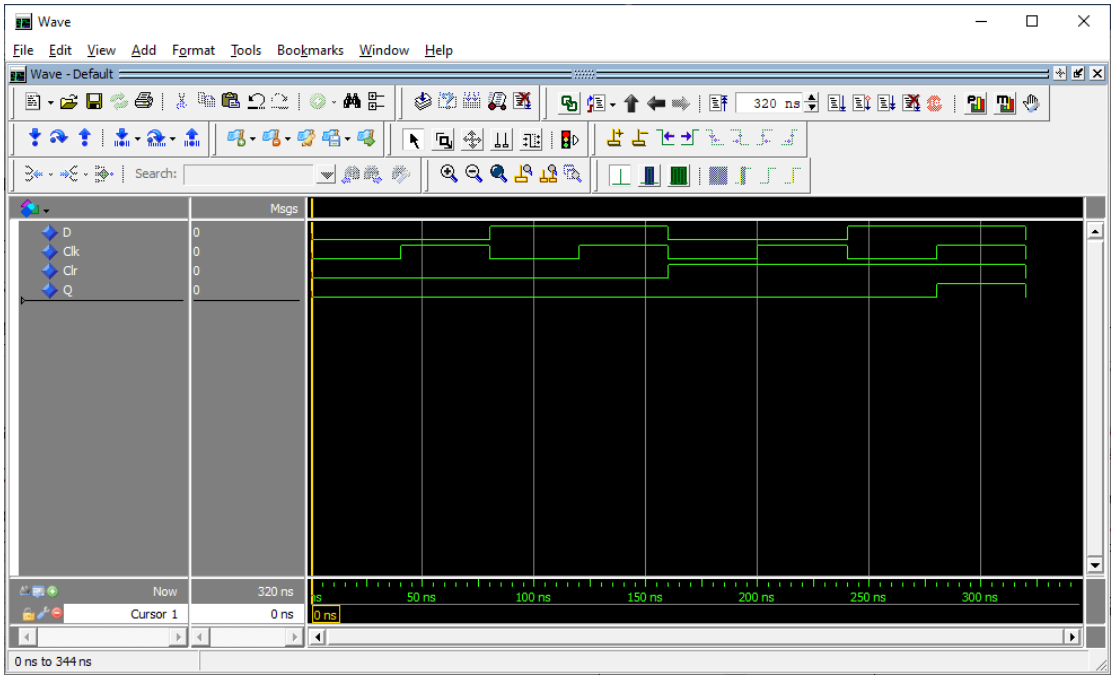
2.21.2.2 Προσομοίωση θετικά ακμοπυροδοτούμενου D φλιπ-φλοπ με VHDL

Η προσομοίωση του θετικά ακμοπυροδοτούμενου D φλιπ-φλοπ περιγράφεται με τον παρακάτω VHDL κώδικα. Παρατηρείται ότι για κάθε 40 ns ο θετικά ακμοπυροδοτούμενος D φλιπ-φλοπ λαμβάνει όλες τις πιθανές τιμές εισόδου. Τέλος, τα αποτελέσματα της προσομοίωσης είναι ορατά στο ακόλουθο διάγραμμα Waveform.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY D_flip_flopAtb IS
END D_flip_flopAtb;

ARCHITECTURE behavior OF D_flip_flopAtb IS

SIGNAL D, Clk, Clr, Q : STD_LOGIC;

COMPONENT D_flip_flopA
PORT (Clk, Clr, D : IN  STD_LOGIC;
            Q : OUT STD_LOGIC);
END COMPONENT;

 BEGIN
  m1: D_flip_flopA PORT MAP(Clk=>Clk,Clr=>Clr,D=>D,Q=>Q);

PROCESS
BEGIN
 Clr<='0';Clk<='0';D<='0'; WAIT FOR 40 ns;
 Clr<='0';Clk<='1';D<='0'; WAIT FOR 40 ns;
 Clr<='0';Clk<='0';D<='1'; WAIT FOR 40 ns;
 Clr<='0';Clk<='1';D<='1'; WAIT FOR 40 ns;
 Clr<='1';Clk<='0';D<='0'; WAIT FOR 40 ns;
```

```
Clr<='1';Clk<='1';D<='0'; WAIT FOR 40 ns;

Clr<='1';Clk<='0';D<='1'; WAIT FOR 40 ns;

Clr<='1';Clk<='1';D<='1'; WAIT FOR 40 ns;

END PROCESS;

END behavior;
```

Σχήμα 2.102 Testbench θετικά ακμοπυροδοτούμενου D φλιπ-φλοπ με ασύγχρονες εισόδους μηδενισμού και θέσης με VHDL.



Σχήμα 2.103 Προσομοίωση θετικά ακμοπυροδοτούμενου D φλιπ-φλοπ με ασύγχρονες εισόδους μηδενισμού και θέσης με VHDL.

2.21.2.3 Περιγραφή θετικά ακμοπυροδοτούμενου D φλιπ-φλοπ με ασύγχρονες εισόδους μηδενισμού και θέσης με χρήση Verilog

Η περιγραφή του παραπάνω σχήματος με Verilog είναι στο παρακάτω σχήμα.

```
module dff_clr_ptr (clk, D, set, Q, reset); //setting reg output and inputs
        input clk, D, set, reset;
         output reg Q;
always@(posedge clk)begin // whenever clock is 1 the following happens
                    if (set == 0 & reset == 0) begin
                              Q <= 1'bx;
                    end
```

```
                else if (set == 0 & reset == 1)begin
                        Q <= 1'b1;
                end
                else if (set == 1 & reset == 0)begin
                        Q <= 1'b0;
                end
                else begin
                        Q <= D;
                end
        end
endmodule
```

Σχήμα 2.104 Περιγραφή θετικά ακμοπυροδοτούμενου D φλιπ-φλοπ με ασύγχρονες εισόδους μηδενισμού και θέσης με χρήση Verilog

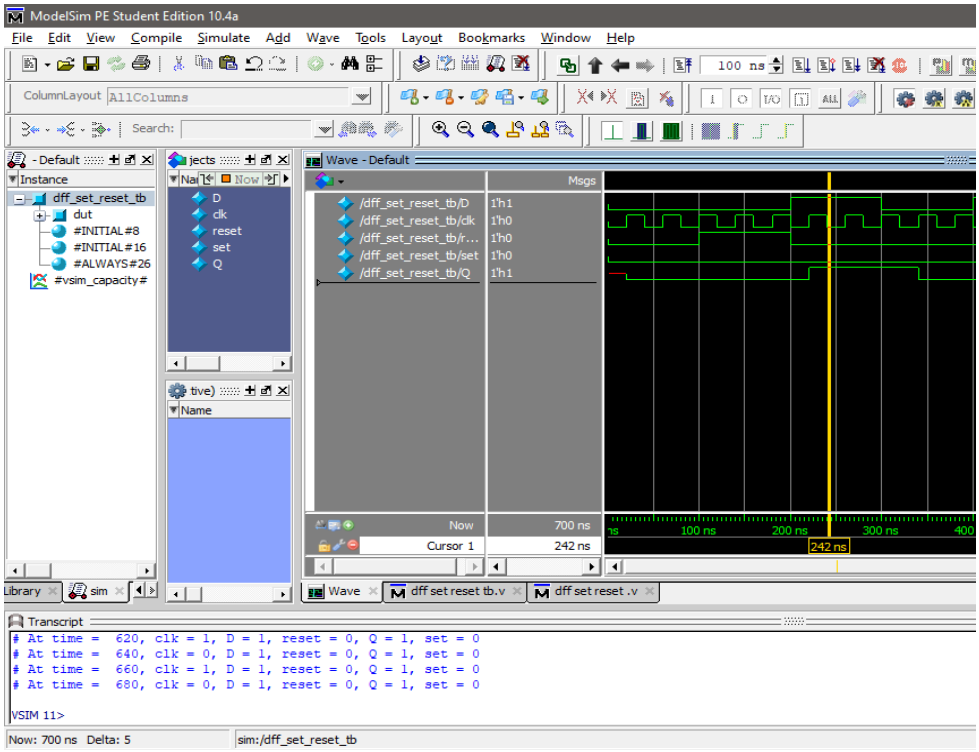
2.21.2.4 Προσομοίωση θετικά ακμοπυροδοτούμενου D φλιπ-φλοπ με Verilog

Το θετικά ακμοπυροδοτούμενο D φλιπ-φλοπ αποτελεί κύκλωμα, στο οποίο με θετικό παλμό ρολογιού, η τιμή της εισόδου D θα γίνει η τιμή της εξόδου. Οι είσοδοι έχουν μήκος ένα bit και αρχικοποιούνται στο μηδέν. Κατόπιν θα ακολουθήσει η προσομοίωση του D φλιπ-φλοπ, με ασύγχρονα στοιχεία μηδενισμού και θέσης. Ο πίνακας αληθείας βρίσκεται στο σχήμα 2.97. Παρακάτω ακολουθούν το testbench και η προσομοίωση.

```
module dff_set_reset_tb; //setting inputs and output

        reg D, clk, reset, set;

        wire Q;

dff_set_reset dut(.Q(Q), .set(set), .reset(reset), .D(D), .clk(clk));// instantiation      initial
begin // initialization

                clk = 0;

                D = 0;

                set = 0;

                reset = 0;

forever #10 clk = ~clk; // as long the simulation is running clock changes every 10ns

        end

        initial begin

                #100; reset=1; D <= 0; //values change at specified times
```

```
            #100; reset=0; D <= 1;

            #100; D <= 0;

            #100; set = 0; D <= 1;

            #100; set = 1; D <= 0;

            #100; set = 0; D <= 1;

            #100; D <= 0;

            #100; D <= 1;

        end

        always@(posedge clk) // whenever clock is 1 the message appears

$monitor("At time = %4d, clk = %b, D = %b, reset = %b, Q = %b, set = %b", $time, clk, D, reset, Q, set); // message for extra info

endmodule
```

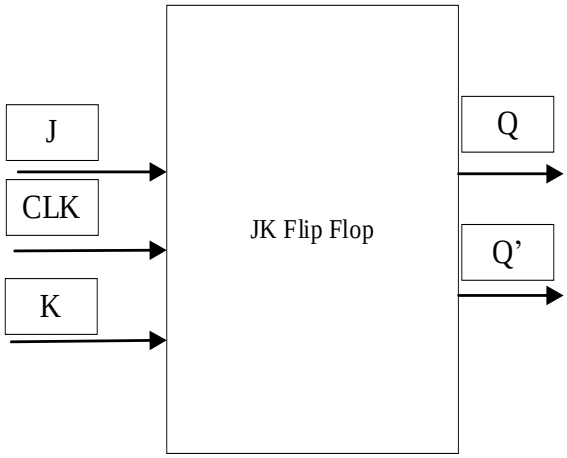
Σχήμα 2.105 Testbench θετικά ακμοπυροδοτούμενου D φλιπ-φλοπ με ασύγχρονες εισόδους μηδενισμού και θέσης με Verilog



Σχήμα 2.106 Προσομοίωση θετικά ακμοπυροδοτούμενου D φλιπ-φλοπ με ασύγχρονες εισόδους μηδενισμού και θέσης με Verilog

2.21.3 Περιγραφή θετικά ακμοπυροδοτούμενου JK φλιπ-φλοπ με VHDL και Verilog

Το θετικά ακμοπυροδοτούμενο JK φλιπ-φλοπ, λειτουργεί παρόμοια με το SR φλιπ-φλοπ καθώς, στην ουσία αποτελεί εξέλιξη του. Η βασική διαφορά μεταξύ τους είναι ότι το JK φλιπ-φλοπ μπορεί να έχει παράλληλα θετικές τιμές και στις δύο εισόδους του, σε αντίθεση με το SR φλιπ-φλοπ.



CLK	J	K	Q	Q'
1	0	0	Q	Q'
1	0	1	0	1
1	1	0	1	0
1	1	1	-	-

Σχήμα 2.107 Θετικά ακμοπυροδοτούμενο JK φλιπ-φλοπ

2.21.3.1 Περιγραφή θετικά ακμοπυροδοτούμενου JK φλιπ-φλοπ με VHDL

Ο παρακάτω κώδικας περιγράφει τον θετικά ακμοπυροδοτούμενο JK φλιπ-φλοπ σε VHDL.

```
LIBRARY ieee;

USE ieee.std_logic_1164.all;


ENTITY JK_flipflop IS

PORT (J, K, Clk : IN STD_LOGIC;

        Q : OUT STD_LOGIC);

END JK_flipflop;


ARCHITECTURE behavior OF JK_flipflop IS

    SIGNAL S: STD_LOGIC;

    SIGNAL JK: STD_LOGIC_VECTOR(1 downto 0);

BEGIN

JK<= J&K;
```

```
        PROCESS (Clk)
        BEGIN
        IF Clk'EVENT AND Clk='1' THEN
        CASE JK IS
                        WHEN "11" => S<=NOT S;
                        WHEN "01" => S<='0';
                        WHEN "10" => S<='1';
                        WHEN OTHERS => S<=S;
                END CASE;
        END IF;
END PROCESS;
Q<=S;
END behavior;
```

Σχήμα 2.108 Περιγραφή θετικά ακμοπυροδοτούμενου JK φλιπ-φλοπ με VHDL.

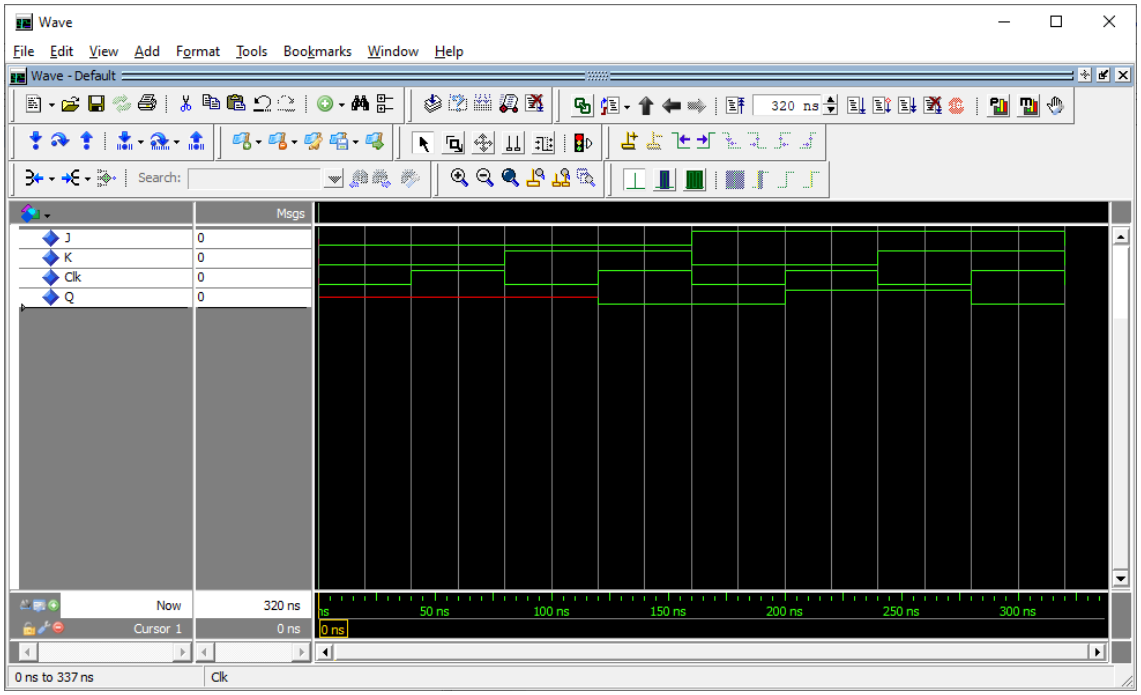
2.21.3.2 Προσομοίωση θετικά ακμοπυροδοτούμενου JK φλιπ-φλοπ με VHDL

Ο ακόλουθος κώδικας είναι ένα Testbench του θετικά ακμοπυροδοτούμενου JK φλιπ-φλοπ σε VHDL. Επιπροσθέτως, κάθε πιθανή τιμή εισόδου έχει εισαχθεί στο κύκλωμα σε διαστήματα 40 ns με τα αποτελέσματα να είναι φαίνονται στο ακόλουθο Waveform διάγραμμα.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;


ENTITY JK_flipfloptb IS
END JK_flipfloptb;


ARCHITECTURE behavior OF JK_flipfloptb IS
```

```
SIGNAL J, K, Clk, Q : STD_LOGIC;


COMPONENT JK_flipflop
PORT (Clk, J, K : IN  STD_LOGIC;
          Q : OUT STD_LOGIC);
END COMPONENT;


 BEGIN


m1: JK_flipflop PORT MAP(Clk=>Clk, J=>J,K=>K,Q=>Q);


PROCESS
BEGIN
 Clk<='0';J<='0';K<='0'; WAIT FOR 40 ns;
 Clk<='1';J<='0';K<='0'; WAIT FOR 40 ns;
 Clk<='0';J<='0';K<='1'; WAIT FOR 40 ns;
 Clk<='1';J<='0';K<='1'; WAIT FOR 40 ns;
 Clk<='0';J<='1';K<='0'; WAIT FOR 40 ns;
 Clk<='1';J<='1';K<='0'; WAIT FOR 40 ns;
 Clk<='0';J<='1';K<='1'; WAIT FOR 40 ns;
 Clk<='1';J<='1';K<='1'; WAIT FOR 40 ns;
 END PROCESS;
END behavior;
```

Σχήμα 2.109 Testbench θετικά ακμοπυροδοτούμενου JK φλιπ-φλοπ με VHDL.



Σχήμα 2.110 Προσομοίωση θετικά ακμοπυροδοτούμενου JK φλιπ-φλοπ με VHDL.

2.21.3.3 Περιγραφή θετικά ακμοπυροδοτούμενου JK φλιπ-φλοπ με Verilog

Ακολουθεί η περιγραφή του θετικά ακμοπυροδοτούμενου JK φλιπ-φλοπ με χρήση Verilog στο σχήμα (2.111).

```
module jkff(input clk, j, k, output reg Q, Qbar); //setting inputs and outputs
        always@(posedge clk) begin //whenever clock is 1 the following happens
                if(k == 0 & j == 0) begin
                        Q <= Q;
                        Qbar <= Qbar;
                end
                else if(k == 0 & j == 1) begin
                        Q <= 1;
                        Qbar <= 0;
                end
                else if(k == 1 & j == 0) begin
                        Q <= 0;
                        Qbar <= 1;
                end
                else if(k == 1 & j == 1) begin
                        Q <= Qbar;
                        Qbar <= ~Qbar;
                end
        end
  endmodule
```

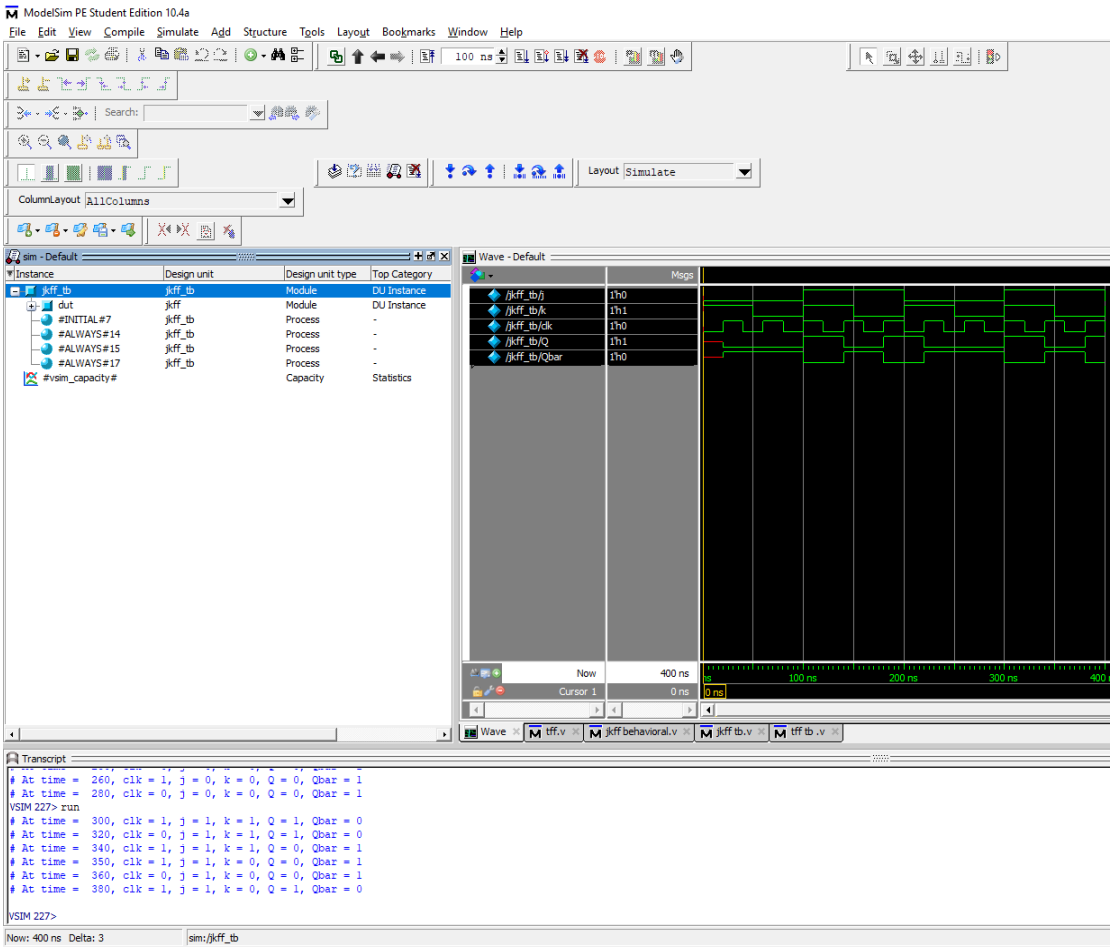
Σχήμα 2.111 Περιγραφή θετικά ακμοπυροδοτούμενου JK φλιπ-φλοπ με Verilog

2.21.3.4 Προσομοίωση θετικά ακμοπυροδοτούμενου JK φλιπ-φλοπ με Verilog

Στο JK φλιπ-φλοπ με δεδομένο θετικό ρολόι, αν οι τιμές των εισόδων είναι ίδιες, τότε για λογικό «0» δεν υπάρχει αλλαγή στην έξοδο. Αντίθετα με λογικό «1» η έξοδος αντιστρέφεται. Σε άλλη περίπτωση η τιμή της εξόδου αλλάζει ανάλογα με την τιμή των εισόδων. Όλες οι είσοδοι έχουν μήκος ένα bit και αρχικοποιούνται στο μηδέν. Ο πίνακας αληθείας βρίσκεται στο σχήμα 2.104. Παρακάτω ακολουθούν το testbench και η προσομοίωση.

```
module jkff_tb();//setting inputs and output
        reg j, k, clk;
        wire Q, Qbar;
 jkff dut(.Q(Q), .Qbar(Qbar), .j(j), .k(k), .clk(clk));// instantiation by port name.
        initial begin // initialization
                clk=0;
                j=0;
                k=1;
 forever #20 clk = ~clk;// As long the simulation is running clock changes every 20ns
        end
        always #100 j= ~j; // every set time in ns the value of inputs changes
        always #50 k=~k;
        always @(j or k)// if any input changes the message appears
        $monitor("At time = %4d, clk = %b, j = %b, k = %b, Q = %b, Qbar = %b",
 $time, clk, j, k, Q, Qbar); // message for extra info
 endmodule
```

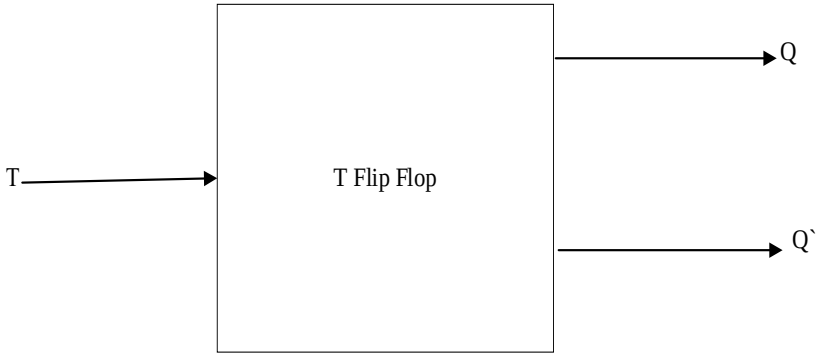
Σχήμα 2.112 Testbench θετικά ακμοπυροδοτούμενου JK φλιπ-φλοπ με Verilog



Σχήμα 2.113 Προσομοίωση θετικά ακμοπυροδοτούμενου JK φλιπ-φλοπ με Verilog

2.21.4 Περιγραφή θετικά ακμοπυροδοτούμενου T φλιπ-φλοπ με τις γλώσσες VHDL και Verilog

Το σχήμα 2.114 που ακολουθεί απεικονίζει το λογικό σύμβολο και τον λογικό πίνακα του θετικά ακμοπυροδοτούμενου T φλιπ-φλοπ. Η υλοποίηση του είναι παράγωγο της ένωσης των εισόδων του T φλιπ-φλοπ.



T	Q'
0	Q
1	~Q

Σχήμα 2.114 Δομή και πίνακας αληθείας T φλιπ-φλοπ

2.21.4.1 Περιγραφή θετικά ακμοπυροδοτούμενου Τ φλιπ-φλοπ με την γλώσσα VHDL

Στο θετικά ακμοπυροδοτούμενο Τ φλιπ-φλοπ, όταν η τιμή του κυκλώματος ωρολογίου του είναι θετική, τότε η έξοδος λαμβάνει την τιμή της εισόδου. Επειδή δεν μπορούσε να ολοκληρωθεί η προσομοίωση στο ModelSim, χρησιμοποιήθηκε το στοιχείο μηδενισμού. Παρακάτω στο σχήμα 2.107 βρίσκεται ο κώδικας της υλοποίησης.

```
LIBRARY ieee;

USE ieee.std_logic_1164.all;

ENTITY T_flip_flop IS

PORT (  T, Clk, Rst : IN  STD_LOGIC;

                Q : OUT STD_LOGIC);

END T_flip_flop;

ARCHITECTURE bhv OF T_flip_flop IS

        SIGNAL S: STD_LOGIC;

BEGIN

        PROCESS (Clk)

        BEGIN

         IF (Rst='1') THEN

           S<='0';

         ELSIF (rising_edge(Clk)) THEN

           IF T ='1' THEN

                 S <= NOT S;

           END IF;

         END IF;

          Q<=S;

        END PROCESS;

END bhv;
```

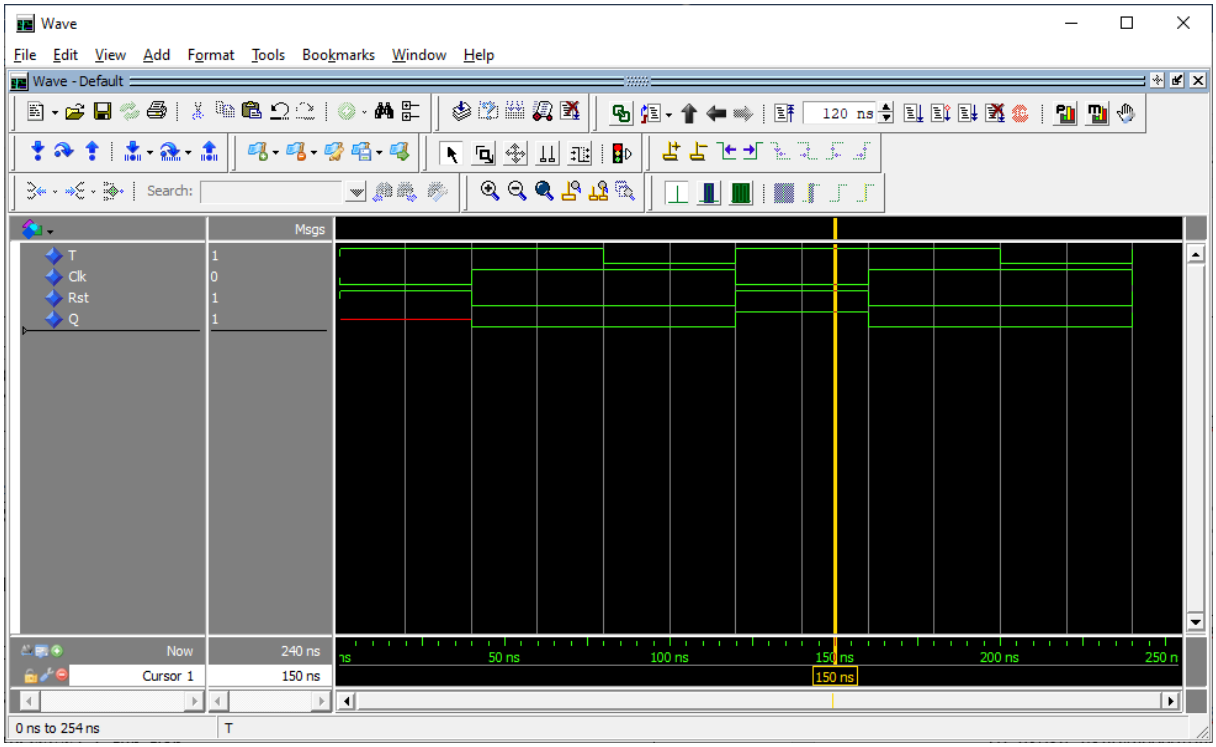
Σχήμα 2.115 Περιγραφή θετικά ακμοπυροδοτούμενου Τ φλιπ-φλοπ με VHDL.

2.21.4.2 Προσομοίωση θετικά ακμοπυροδοτούμενου T φλιπ-φλοπ με την γλώσσα VHDL

Ακολουθεί η προσομοίωση του θετικά ακμοπυροδοτούμενου T φλιπ-φλοπ σε VHDL κώδικα. Γίνεται εισαγωγή νέων τιμών στην είσοδο ανά 40 ns τα αποτελέσματα των οποίων είναι εμφανείς στο παρακάτω Waveform.

```
LIBRARY IEEE;

USE IEEE.STD_LOGIC_1164.ALL;

ENTITY T_flip_floptb IS

END T_flip_floptb;


ARCHITECTURE behavior OF T_flip_floptb IS

SIGNAL T, Clk, Rst, Q : STD_LOGIC;


COMPONENT T_flip_flop

PORT (Clk, T, Rst : IN  STD_LOGIC;

            Q : OUT STD_LOGIC);

END COMPONENT;

 BEGIN

  m1: T_flip_flop PORT MAP(Clk=>Clk, Rst=>Rst, T=>T,Q=>Q);

PROCESS

BEGIN

 Rst<='1';Clk<='0';T<='1'; WAIT FOR 40 ns;

 Rst<='0';Clk<='1';T<='1'; WAIT FOR 40 ns;

 Rst<='0';Clk<='1';T<='0'; WAIT FOR 40 ns;

 END PROCESS;

END behavior;
```

Σχήμα 2.116 Testbench θετικά ακμοπυροδοτούμενου T φλιπ-φλοπ με VHDL.



Σχήμα 2.117 Προσομοίωση θετικά ακμοπυροδοτούμενου T φλιπ-φλοπ με VHDL.

2.21.4.3 Περιγραφή θετικά ακμοπυροδοτούμενου T φλιπ-φλοπ με την γλώσσα Verilog

Ο παρακάτω κώδικας Verilog περιγράφει το θετικά ακμοπυροδοτούμενο T φλιπ-φλοπ.

```
module tff(input clk, reset, t, output reg Q);//setting reg output and inputs
        always @(posedge clk)//for positive change clock the following happens
                if (reset)
//IMPORTANT NOTE RESET IS USED DUE TO PROBLEMS WITH MODELSIM
                        Q <= 1'b0;
                else if (t)
                        Q <= ~Q;
endmodule
```

Σχήμα 2.118 Περιγραφή θετικά ακμοπυροδοτούμενου T φλιπ-φλοπ με Verilog

2.21.4.4 Προσομοίωση θετικά ακμοπυροδοτούμενου T φλιπ-φλοπ με την γλώσσα Verilog

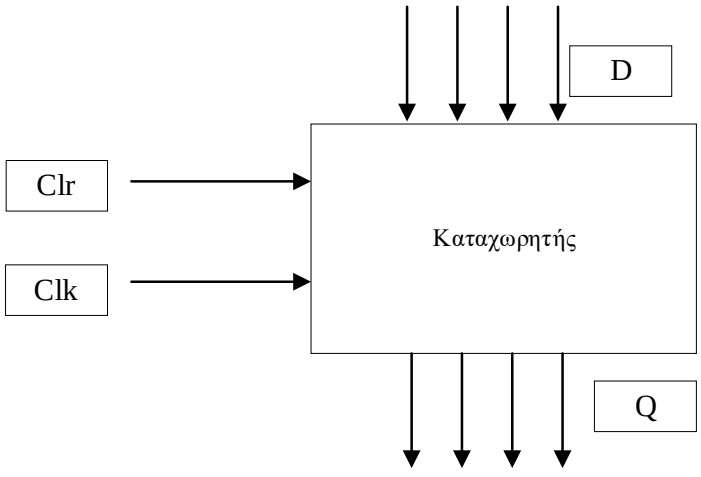
Το κύκλωμα δέχεται δύο εισόδους, μια για τον μηδενισμό και μια για το σήμα που εισέρχεται. Επίσης δέχεται και σήμα ρολογιού. Υπάρχει μια έξοδος και είναι το σήμα εξόδου που βγαίνει από το κύκλωμα. Οι είσοδοι αρχικοποιούνται στο μηδέν και αντιστροφές γίνονται στο ρολόι ανά δέκα, στην είσοδο αν είκοσι και στο σήμα μηδενισμού ανά εκατό νανοδευτερόλεπτα.

```
module tff_tb(); //setting inputs and output
        reg clk, t, reset;
        wire Q;
tff dut(.clk(clk), .t(t), .Q(Q), .reset(reset)); // instantiation by port name.
        initial begin // initialization
                clk = 0;
                t = 0;
                reset =1;
forever #10 clk = ~clk; //while the simulation is running every 10ns clock value changes
        end
        always #20 t = ~t; // every set time in ns the value of inputs changes
        always #100 reset = ~reset;
        always @(t) // if any input changes the message appears
$monitor ("At time t=%4d, t=%d Q=%d", $time , t, Q); // message for extra info
endmodule
```

Σχήμα 2.119 Testbench θετικά ακμοπυροδοτούμενου T φλιπ-φλοπ με Verilog

2.22 Περιγραφή καταχωρητή 4bit με χρήση με VHDL και Verilog

Ο καταχωρητής αποτελείται από πλήθος φλιπ-φλοπ τα οποία είναι συγχρονισμένα με το ίδιο ρολόι. Χρησιμοποιούνται για αποθήκευση μεταβλητών με σκοπό αυτές να διαβαστούν αργότερα. Δέχεται μια είσοδο D μαζί με κύκλωμα ρολογιού και σήμα μηδενισμού. Έχει μια έξοδο Q. Στο σχήμα 2.120 δίνεται το λογικό σύμβολο του κυκλώματος.



Σχήμα 2.120 Καταχωρητής 4bit με ασύγχρονη είσοδο μηδενισμού

2.22.1 Περιγραφή καταχωρητή 4bit με χρήση με VHDL

Παρακάτω περιγράφεται ο 4bit καταχωρητής σε VHDL κώδικα.

```
 LIBRARY ieee;
USE ieee.std_logic_1164.all;


ENTITY Reg_4 IS
PORT (X : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
Clr, Clk : IN STD_LOGIC;
Z : OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
END Reg_4;


ARCHITECTURE Behavioral OF Reg_4 IS
SIGNAL Q: STD_LOGIC_VECTOR(3 DOWNTO 0);
BEGIN
        PROCESS (Clr, Clk)
        BEGIN
                IF Clr = '1' THEN
                        Q <= "0000";
                ELSIF Clk'EVENT AND Clk='1' THEN
                        Q <= X;
                END IF;
        END PROCESS;


        Z<=Q;
END Behavioral;
```

Σχήμα 2.121 Περιγραφή καταχωρητή 4bit με ασύχρονη είσοδο μηδενισμού VHDL.

2.22.2 Περιγραφή καταχωρητή 4bit με χρήση με Verilog

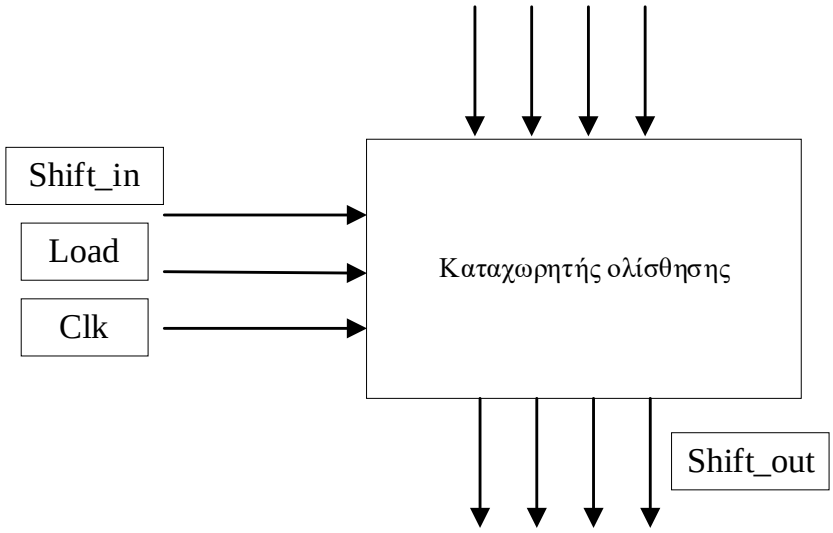
Στα σχήμα 2.121 δίνεται η περιγραφή σε Verilog.

```
module register_4bit (input [3:0] D, input clk, clr, output reg [3:0] Q);
  always @(posedge clk) //whenever clock is positive the following happens
        if (clr)
                Q <= 0;
        else
                Q <= D;
endmodule
```

Σχήμα 2.122 Περιγραφή καταχωρητή 4bit με ασύχρονη είσοδο μηδενισμού

2.23 Περιγραφή καταχωρητή ολίσθησης 4bit με παράλληλη φόρτωση με VHDL και Verilog

Ο καταχωρητής ολίσθησης λειτουργεί παρόμοια με τον κανονικό καταχωρητή αλλά δίνει έξοδο μόνο το bit που βγαίνει εκτός με την χρήση της ολίσθησης. Δέχεται είσοδο με ολίσθηση προς τα αριστερά μαζί με σήμα ρολογιού και φόρτωσης. Ακολουθεί το λογικό σύμβολο του καταχωρητή ολίσθησης 4bit στο σχήμα 2.123.



Σχήμα 2.123 Καταχωρητής ολίσθησης 4bit με παράλληλη φόρτωση

2.23.1 Περιγραφή καταχωρητή ολίσθησης 4bit με παράλληλη φόρτωση με VHDL

Ακολουθεί η περιγραφή του καταχωρητή ολίσθησης 4bit με παράλληλη φόρτωση με VHDL.

```
LIBRARY ieee;

USE ieee.std_logic_1164.all;



ENTITY Shift_4 IS
```

```
PORT (          X : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
        Sin, Ld, Clk : IN STD_LOGIC;
                Z : OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
END Shift_4;


ARCHITECTURE Behavioral OF Shift_4 IS
        SIGNAL Q:STD_LOGIC_VECTOR (3 DOWNTO 0);
BEGIN
        PROCESS (Sin, Clk)
        BEGIN
        IF Clk'EVENT AND Clk='1' THEN
                IF LD='1' THEN
                        Q <= X;
                ELSE
                        Q(0)<=Q(1);
                        Q(1)<=Q(2);
                        Q(2)<=Q(3);
                        Q(3)<= Sin;
                END IF;
        END IF;
        END PROCESS;
Z<=Q;
END Behavioral;
```

Σχήμα 2.124 Περιγραφή ολίσθησης 4bit με παράλληλη φόρτωση VHDL.

2.23.2 Περιγραφή καταχωρητή ολίσθησης 4bit με παράλληλη φόρτωση με Verilog

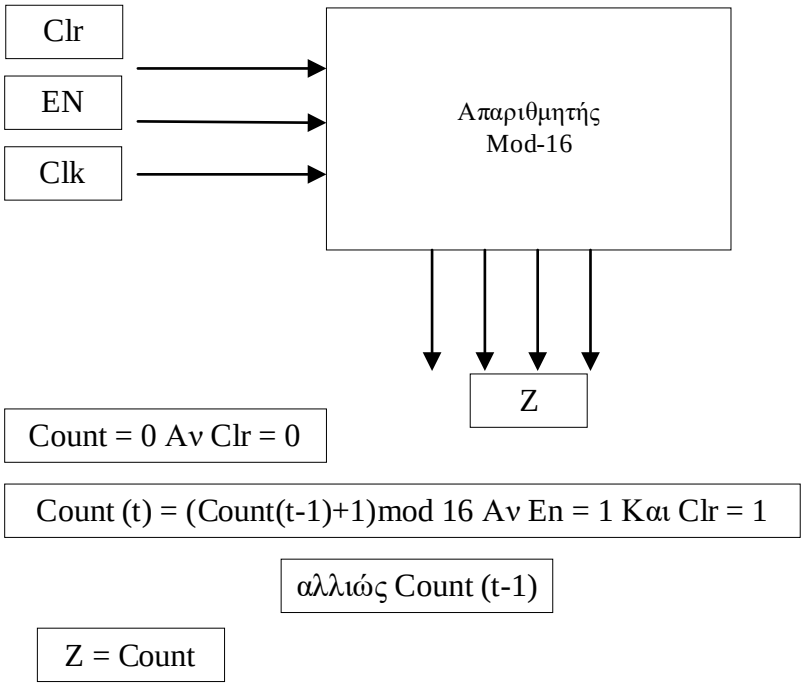
Παρακάτω αναλύεται ο καταχωρητής ολίσθησης τεσσάρων Bit με Verilog.

```
module shift_register(input shift_in, load, clk, output shift_out);
        reg Q0, Q1, Q2, Q3; //setting reg for each byte of the register
        assign shift_out = Q3; // byte 4 is assigned as output
always @(posedge clk) begin//whenever clock is positive the following happens
        if (load)
                Q3 <= 1'bx;
        else
                Q3 <= Q2;
                Q2 <= Q1;
                Q1 <= Q0;
                Q0 <= shift_in;
        end
  endmodule
```

Σχήμα 2.125 Περιγραφή ολίσθησης 4bit με παράλληλη φόρτωση

2.24 Περιγραφή με αύξοντα απαριθμητή των 4bit με χρήση VHDL και Verilog

Στο παρακάτω σχήμα δίνεται το λογικό σύμβολο και περιγράφεται η λειτουργία ενός αύξοντα απαριθμητή των 4bit(mod-16) με ασύγχρονη είσοδο μηδενισμού και είσοδο ενεργοποίησης.



Σχήμα 2.126 Απαριθμητής 4bit με ασύγχρονη είσοδο μηδενισμού και είσοδο ενεργοποίησης

2.24.1 Περιγραφή με αύξοντα απαριθμητή των 4bit με χρήση VHDL

Η περιγραφή που ακολουθεί είναι του αύξοντα απαριθμητή των 4bit με χρήση VHDL.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;

ENTITY up_count_4 IS
PORT (Clr, Clk, En : IN STD_LOGIC;
                Z : OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
END up_count_4;

ARCHITECTURE behavioral OF up_count_4 IS
        SIGNAL Count : STD_LOGIC_VECTOR(3 DOWNTO 0);
BEGIN
        PROCESS (Clk, Clr)
        BEGIN
        IF Clr='0' THEN
                Count <= "0000";
        ELSIF Clk'EVENT AND Clk='1' THEN
                IF En='1' THEN
                        Count <= Count+1;
                ELSE
                        Count <= Count;
                END IF;
        END IF;
        END PROCESS;
Z<=Count;
END behavioral;
```

Σχήμα 2.127 Περιγραφή αύξοντα απαριθμητή 4bit με ασύγχρονη είσοδο μηδενισμού και είσοδο ενεργοποίησης με VHDL.

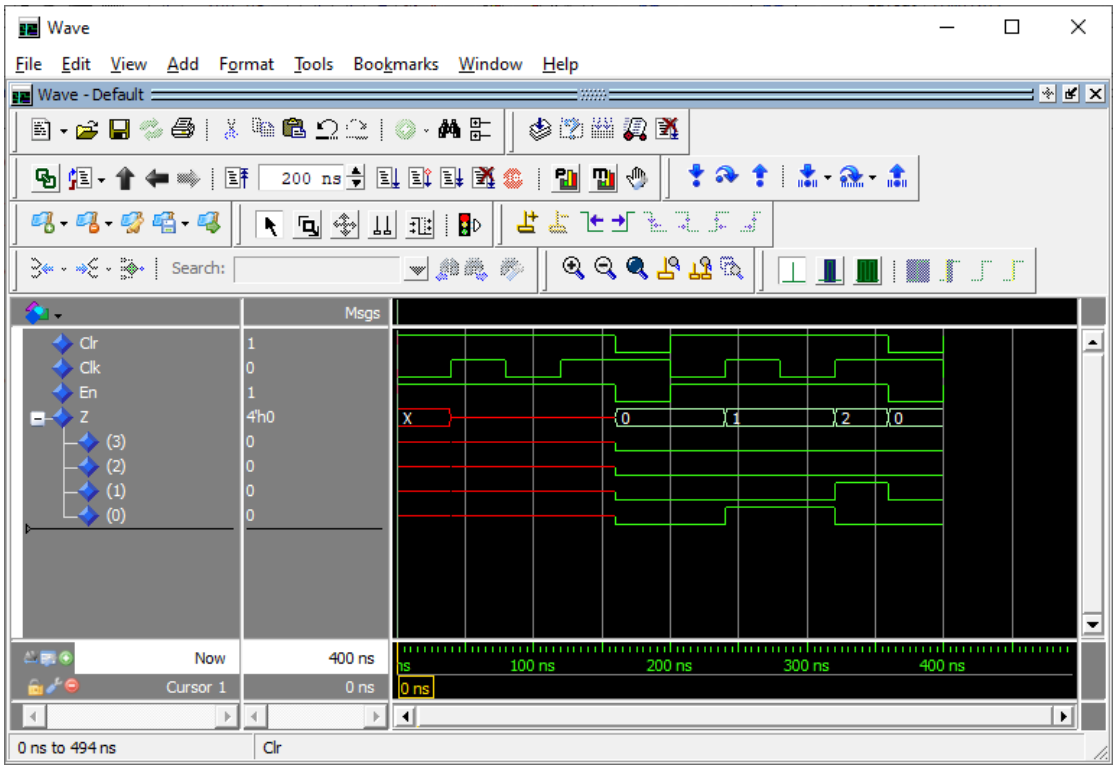
2.24.2 Προσομοίωση αύξοντα απαριθμητή 4bit με VHDL

Ακολουθεί το Testbench του αύξοντα απαριθμητή 4bit με VHDL. Όταν το «En» έχει την τιμή '1' και η τιμή του «Clk» αλλάζει από '0' σε '1' τότε ο απαριθμητής αυξάνεται κατά 1. Στην περίπτωση που το «En» είναι '0' τότε η τιμή του απαριθμητή μένει σταθερή ενώ όταν το «Clr» έχει την τιμή '0' τότε ο απαριθμητής μηδενίζει. Τα αποτελέσματα αυτά είναι ορατά στο ακόλουθο Waveform διάγραμμα.

```
LIBRARY IEEE;

USE IEEE.STD_LOGIC_1164.ALL;


ENTITY up_count_4tb IS

END up_count_4tb;


ARCHITECTURE behavior OF up_count_4tb IS


SIGNAL Clr, Clk, En : STD_LOGIC;

SIGNAL        Z : STD_LOGIC_VECTOR(3 DOWNTO 0);


COMPONENT up_count_4

PORT (Clr, Clk, En : IN  STD_LOGIC;

              Z : OUT STD_LOGIC_VECTOR(3 DOWNTO 0));

END COMPONENT;


 BEGIN


m1: up_count_4 PORT MAP(En=>En,Clr=>Clr,Clk=>Clk,Z=>Z);
```

```
PROCESS
BEGIN
  En<='1';Clk<='0';Clr<='1';WAIT FOR 40 ns;
 En<='1';Clk<='1';Clr<='1';WAIT FOR 40 ns;
 En<='1';Clk<='0';Clr<='1';WAIT FOR 40 ns;
 En<='1';Clk<='1';Clr<='1';WAIT FOR 40 ns;
 En<='0';Clk<='1';Clr<='0';WAIT FOR 40 ns;

 END PROCESS;
END behavior;
```

Σχήμα 2.128 Testbench αύξοντα απαριθμητή 4bit VHDL.



Σχήμα 2.129 Προσομοίωση αύξοντα απαριθμητή 4bit VHDL.

2.24.3 Περιγραφή με αύξοντα απαριθμητή των 4bit με χρήση Verilog

Η σχεδίαση με Verilog του παραπάνω σχήματος 2.117 δίνεται παρακάτω στο σχήμα 2.130. Δημιουργείται μια έξτρα μεταβλητή «counter_up», η οποία λειτουργεί για αποθήκευση κατά την αύξηση. Αντίστοιχη λειτουργία είχε και στο προηγούμενο

κύκλωμα η μεταβλητή «Q3». Τίθεται always μπλοκ με λίστα ευαισθησίας θετική άνοδο ρολογιού και εισόδου μηδενισμού. Στο εσωτερικό του μπλοκ θέτουμε βρόγχο if-else όπου για 0 στην εισόδου μηδενισμού η έξοδος είναι 0.

```
module up_counter(input clk, clr, en, output[3:0] Z); //1bit inputs and 4bit output
        reg [3:0] counter_up; // assigning reg datatype
        always @(posedge clk) begin
        // for either positive clock execute the following
                if(clr==0)
                        counter_up <= 4'd0;
                else if(en==1)
                        counter_up <= counter_up + 4'd1;
        end
assign Z = counter_up; // assigning reg value to output
endmodule
```

Σχήμα 2.130 Περιγραφή αύξοντα απαριθμητή 4bit με ασύγχρονη είσοδο μηδενισμού και είσοδο ενεργοποίησης με Verilog.

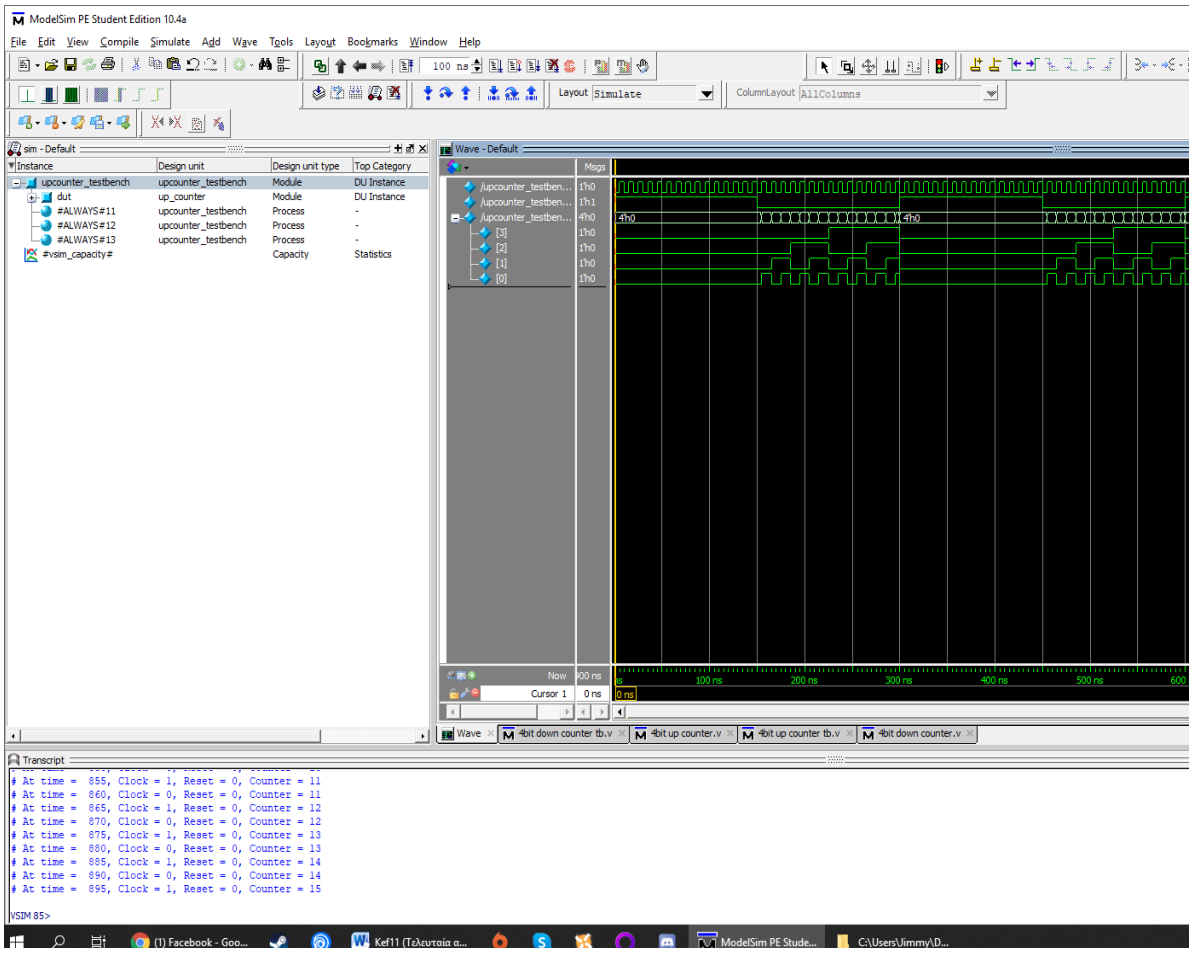
2.24.4 Προσομοίωση αύξοντα απαριθμητή 4bit με Verilog

Ο αύξοντας απαριθμητής τεσσάρων bit αποτελεί ένα κύκλωμα, το οποίο όσο η είσοδος επίτρεψης του είναι θετική, για κάθε θετικό παλμό του ρολογιού αυξάνει η έξοδος του κατά ένα. Στο συγκεκριμένο κύκλωμα περιέχεται και ασύγχρονη είσοδος μηδενισμού. Το αντίστοιχο κύκλωμα φθίνοντα απαριθμητή, έχει τον ίδιο ακριβώς κώδικά προσομοίωσης. Παρακάτω ακολουθούν το testbench και η προσομοίωση.

```
module upcounter_testbench();

        reg clk, reset; //setting inputs

        wire [3:0] counter; //setting output

        up_counter dut(clk, reset, counter); //initialization

        initial begin

                clk=0;

                reset =1;

        end

        always #5 clk=~clk; // clock is set to change value every 5ns

        always #150 reset=~reset;
```

```
        always @(clk)//for every change in clock the following happens

$monitor("At time = %4d, Clock = %d, Reset = %d, Counter = %d", $time, clk, reset, counter);

  endmodule
```

Σχήμα 2.131 Testbench αύξοντα απαριθμητή 4bit Verilog.



Σχήμα 2.132 Προσομοίωση αύξοντα απαριθμητή 4bit Verilog

2.25. Περιγραφή φθίνοντα απαριθμητή 4bit με χρήση VHDL και Verilog

Ένας σύγχρονος απαριθμητής των 4bit, όπου για κάθε θετικό παλμό του ρολογιού αν η τιμή του load είναι «1» τότε ο απαριθμητής επαναφέρεται στην αρχική του κατάσταση, ενώ αλλιώς μειώνεται κατά 1.

2.25.1 Περιγραφή φθίνοντα απαριθμητή 4bit με χρήση VHDL

Ακολουθεί ο κώδικας περιγραφής σε VHDL.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;


ENTITY down_count_4 IS
PORT (Clk, Ld: IN  STD_LOGIC;
        Z: OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
END down_count_4;
ARCHITECTURE behavioral OF down_count_4 IS
        SIGNAL Count: STD_LOGIC_VECTOR(3 DOWNTO 0);
BEGIN
        PROCESS (Clk)
        BEGIN
       IF Ld='1' THEN
              Count <= "1111";
       ELSIF Clk'EVENT AND Clk='1' THEN
              Count <= Count - 1;
          END IF;
        END PROCESS;
  Z<=Count;
END behavioral;
```

Σχήμα 2.133 Περιγραφή φθίνοντα απαριθμητή 4bit VHDL.

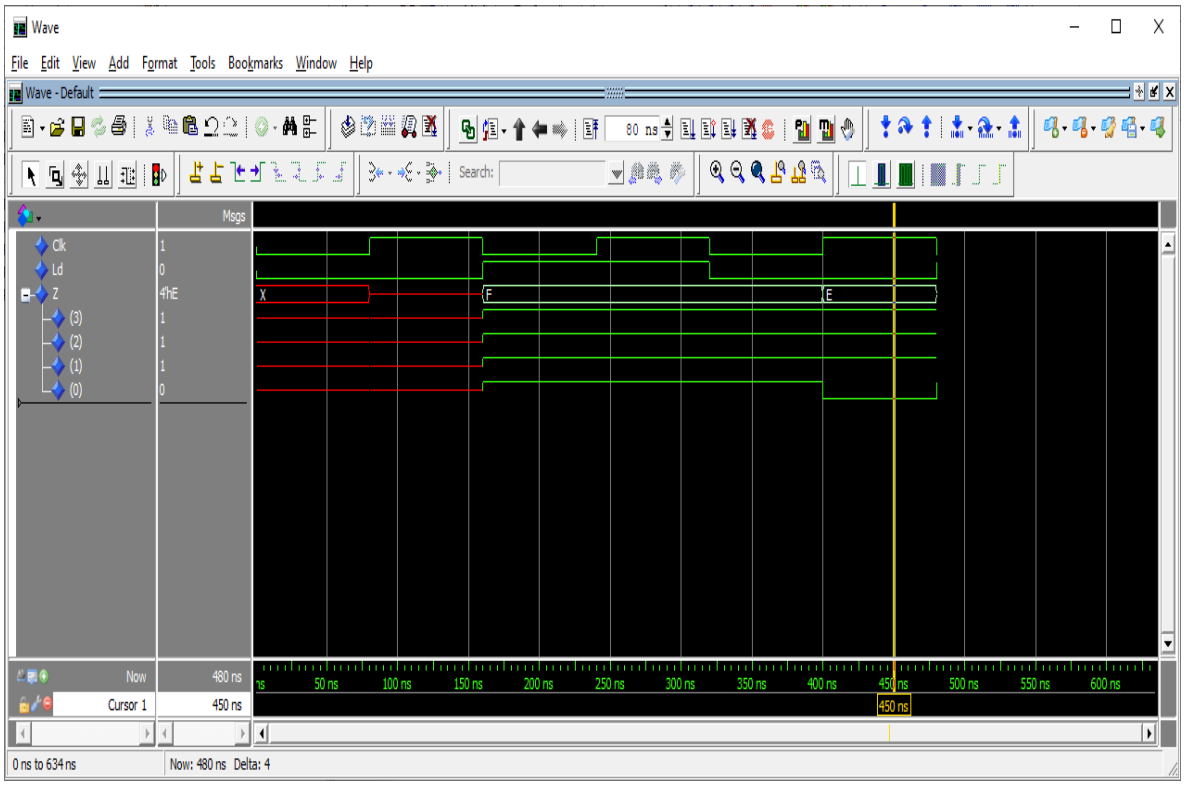
2.25.2 Προσομοίωση φθίνοντα απαριθμητή με χρήση VHDL

Στο σχήμα που ακολουθεί περιγράφεται η προσομοίωση του φθίνοντα απαριθμητή σε VHDL. Όταν γίνεται η αλλαγή του ρολογιού (Clk) από 0 σε 1 και η τιμή της μεταβλητής εισόδου Ld είναι 0 τότε ο απαριθμητής μειώνεται κατά 1. Το αποτέλεσμα φαίνεται στο παρακάτω Waveform. Παρατηρείται ότι για τα πρώτα 160 ns δεν υπάρχει αποτέλεσμα και αυτό είναι λογικό διότι δεν έχει γίνει η αλλαγή του Clock από 0 σε 1.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY down_count_4tb IS
END down_count_4tb;

ARCHITECTURE behavior OF down_count_4tb IS

SIGNAL Clk, Ld : STD_LOGIC;
SIGNAL Z : STD_LOGIC_VECTOR(3 DOWNTO 0);

COMPONENT down_count_4
PORT (Clk, Ld : IN  STD_LOGIC;
        Z : OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
END COMPONENT;

 BEGIN

m1: down_count_4 PORT MAP(Clk=>Clk,Ld=>Ld,Z=>Z);

PROCESS
BEGIN

 Clk<='0';Ld<='0';WAIT FOR 80 ns;
 Clk<='1';Ld<='0';WAIT FOR 80 ns;
 Clk<='0';Ld<='1';WAIT FOR 80 ns;
```

```
Clk<='1';Ld<='1';WAIT FOR 80 ns;

END PROCESS;

END behavior;
```

Σχήμα 2.134 Testbench φθίνοντας απαριθμητή 4bit VHDL.



Σχήμα 2.135 Προσομοίωση φθίνοντα απαριθμητή 4bit

2.25.3 Περιγραφή φθίνοντα απαριθμητή 4bit με χρήση Verilog

Παρακάτω στο σχήμα 2.136 περιγράφεται με Verilog.

```
module down_counter(input clk, load, output [3:0] Z); //1bit inputs and 4bit output
        reg [3:0] counter_down; // assigning reg datatype
        always @(posedge clk) begin
 // for either positive clock or reset execute the following

                if (load==1)
                        counter_down <= 4'hf;
                else
                        counter_down <= counter_down - 4'd1;
        end
```

```
assign Z = counter_down; // assigning reg value to output
endmodule
```

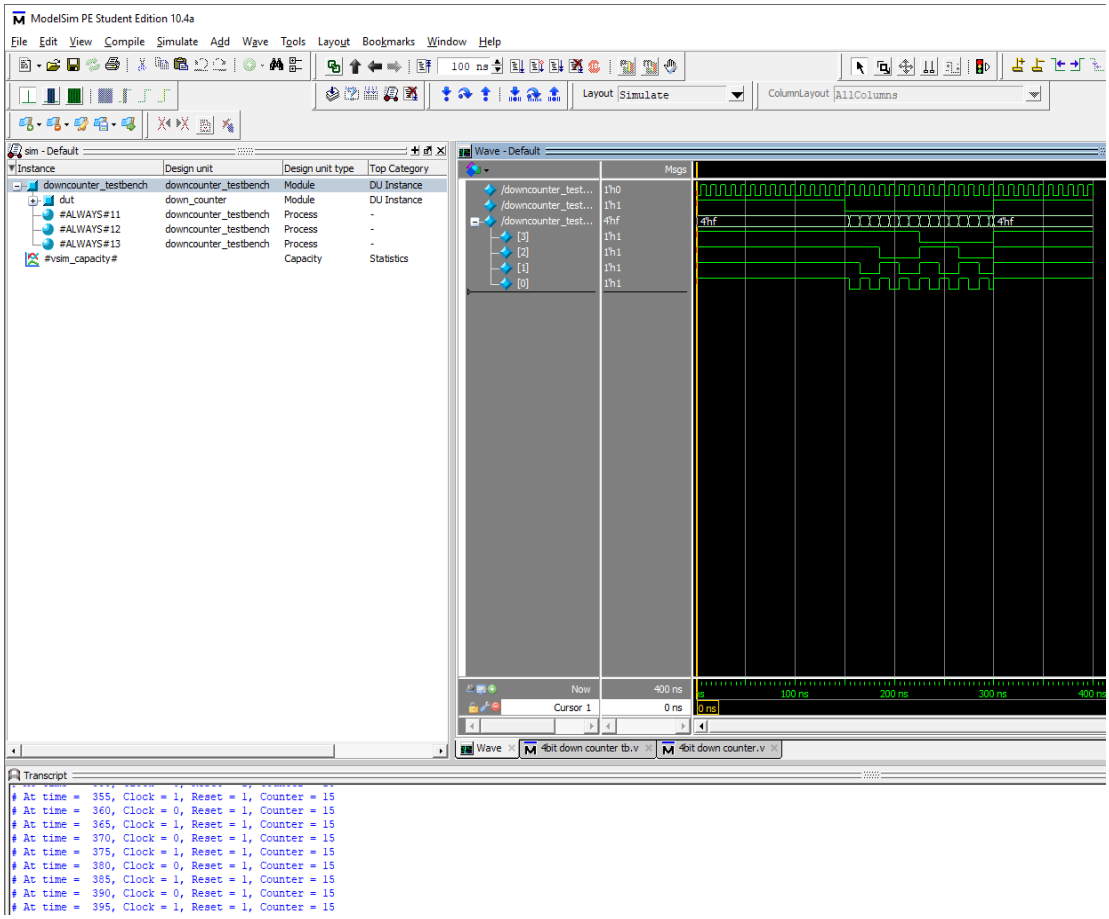
Σχήμα 2.136 Περιγραφή φθίνοντα απαριθμητή 4bit

2.25.4 Προσομοίωση φθίνοντα απαριθμητή με χρήση Verilog

Ο φθίνοντας απαριθμητής τεσσάρων bit αποτελεί ένα κύκλωμα, το οποίο όσο η είσοδος επίτρεψης του είναι θετική, για κάθε θετικό παλμό του ρολογιού μειώνεται η έξοδος του κατά ένα. Στο συγκεκριμένο κύκλωμα περιέχεται και ασύγχρονη είσοδος μηδενισμού. Το αντίστοιχο κύκλωμα αύξοντα απαριθμητή, έχει τον ίδιο ακριβώς κώδικά προσομοίωσης. Παρακάτω ακολουθούν το testbench και η προσομοίωση

```
module downcounter_testbench();
reg clk, reset; //setting inputs
wire [3:0] counter; //setting output
down_counter dut(clk, reset, counter); //initialization
initial begin
        clk=0;
        reset=1;
end
always #5 clk=~clk; // clock is set to change value every 5ns
always #150 reset=~reset;
always @(clk) //for every change in clock the following happens
$monitor("At time = %4d, Clock = %d, Reset = %d, Counter = %d", $time, clk, reset, counter);
endmodule
```

Σχήμα 2.137 Testbench φθίνοντας απαριθμητή 4bit



Σχήμα 2.138 Προσομοίωση φθίνοντα απαριθμητή 4bit

3. Αριθμητικά κυκλώματα με VHDL και Verilog

3.1 Εισαγωγή

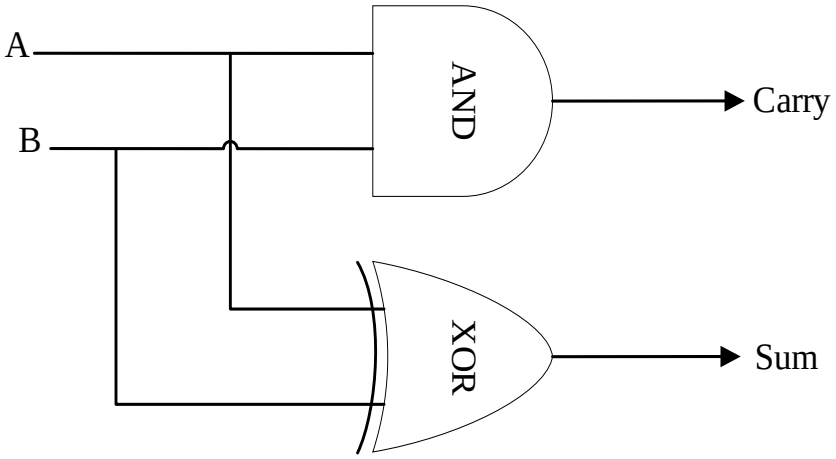
Στο προηγούμενο κεφάλαιο έγινε μελέτη συνδυαστικών και ακολουθιακών κυκλωμάτων όπου επεξηγήθηκαν οι βασικές αρχές και μεθοδολογίες της σύνταξης του κώδικα των VHDL και Verilog, μαζί με προσομοιώσεις των κυκλωμάτων. Σε αυτό το κεφάλαιο ακολουθεί ανάλυση αριθμητικών κυκλωμάτων όπως αθροιστές και πολλαπλασιαστές. Έπειτα από την ανάλυση του κάθε κυκλώματος θα ακολουθήσει η προσομοίωση του εν λόγω κυκλώματος

3.2 Σχεδίαση αθροιστών με VHDL και Verilog

Οι αθροιστές αποτελούν πυλώνα των σύγχρονων ψηφιακών κυκλωμάτων. Έχουν πληθώρα χρήσεων από απλά κυκλώματα μέχρι και υλοποίηση τμημάτων αριθμητικών λογικών μονάδων και επεξεργαστών. Αποτελούν επίσης σημαντικό τμήμα αυτής της διπλωματικής καθώς, πέρα από την κατανόηση και τα παραδείγματα για τα οποία χρησιμοποιούνται, είναι σημαντικό τμήμα των αρχιτεκτονικών των πολλαπλασιαστών που θα υλοποιηθούν μετέπειτα. Υπάρχουν διάφορα είδη αθροιστών όπως ο αθροιστής με χρήση διάδοσης κρατουμένου. Παρακάτω θα ακολουθήσει η περιγραφή διάφορων αθροιστών με χρήση της Verilog.

3.2.1 Περιγραφή ημιαθροιστή με VHDL και Verilog

Ο ημιαθροιστής αποτελεί την βασική μονάδα αθροιστή που δέχεται δύο εισόδους και εμφανίζει ως αποτέλεσμα είτε ένα άθροισμα είτε ένα κρατούμενο. Αποτελεί τον πιο βασικό αθροιστή με στοιχεία σε κάθε διαφορετική υλοποίηση αθροιστή. Έστω οι είσοδοι a, b και οι έξοδοι το άθροισμα sum και το κρατούμενο carry. Η υλοποίηση του μπορεί να πραγματωθεί με την χρήση λογικών παραστάσεων αλλά και με πληθώρα άλλων τρόπων όπως με χρήση μοντελοποίησης δομής. Οι λογικές παραστάσεις του ημιαθροιστή στις οποίες βασίζεται η υλοποίηση, για άθροισμα και κρατούμενο είναι αντίστοιχα sum = a $\oplus$ b και carry = a * b.



Σχήμα 3.1 Κύκλωμα ημιαθροιστή

3.2.1.1 Περιγραφή ημιαθροιστή με VHDL

Παρακάτω γίνεται η περιγραφή του ημιαθροιστή σε VHDL κώδικα.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY half_add IS
PORT (x, y : IN  BIT;
      s, cout: OUT BIT);
END half_add;

ARCHITECTURE LogicFunc OF half_add IS
BEGIN
      s <= x XOR y ;
      cout <= x AND y;
END LogicFunc;
```

Σχήμα 3.2 Περιγραφή ημιαθροιστή με VHDL

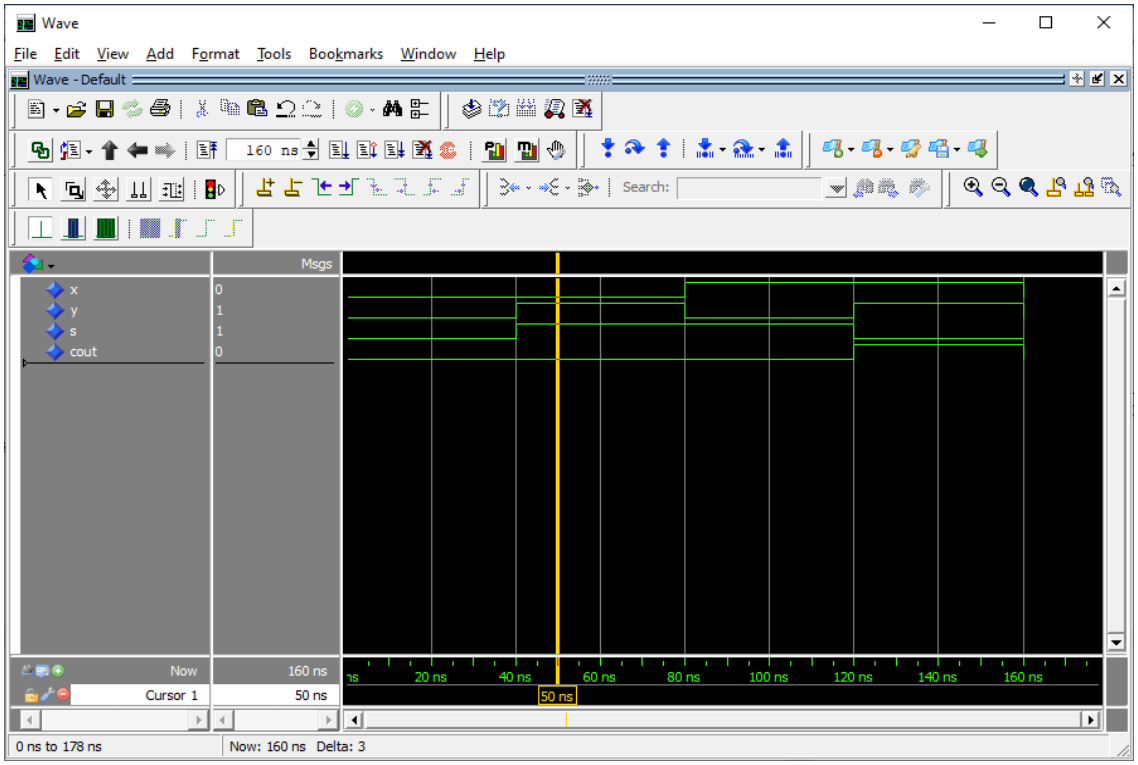
3.2.1.2 Προσομοίωση του ημιαθροιστή με χρήση VHDL

Ακολουθεί η προσομοίωση του ημιαθροιστή με την δημιουργία ενός Testbench στο οποίο εκχωρούνται όλες οι πιθανές τιμές ανά 40 ns (νανοδευτερόλεπτα). Επιπλέον, παρατηρείται το αποτέλεσμα της προσομοίωσης σε γράφημα Waveform.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY half_addtb IS
END half_addtb;
ARCHITECTURE behavior OF half_addtb IS
```

```
SIGNAL x, y, s, cout : BIT;

COMPONENT half_add
PORT (x, y : IN BIT;
     s, cout: OUT BIT);
END COMPONENT;

 BEGIN

m1: half_add PORT MAP(x=>x, y=>y, s=>s, cout=>cout);

PROCESS
BEGIN
x<='0';y<='0'; WAIT FOR 40 ns;
x<='0';y<='1'; WAIT FOR 40 ns;
x<='1';y<='0'; WAIT FOR 40 ns;
x<='1';y<='1'; WAIT FOR 40 ns;
 END PROCESS;
 END behavior;
```

Σχήμα 3.3 Testbench ημιαθροιστή με VHDL



Σχήμα 3.4 Προσομοίωση του ημιαθροιστή με VHDL

3.2.1.3 Περιγραφή ημιαθροιστή με Verilog

Παρακάτω ακολουθεί η περιγραφή του ημιαθροιστή με Verilog.

```
module half_adder( a, b, sum, carry); //setting wire inputs and reg output
    input a, b;
    output sum, carry;
        assign sum  = a^b; //assignment to outputs using logical expressions
        assign carry = a&b;
    endmodule
```

Σχήμα 3.5 Περιγραφή ημιαθροιστή με Verilog

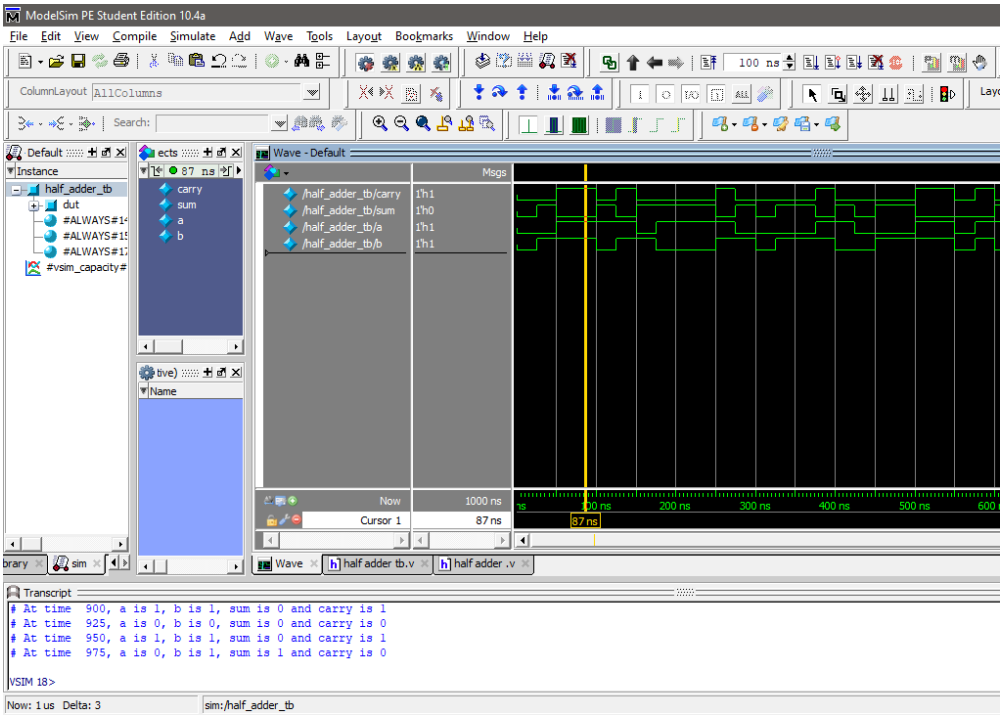
3.2.1.4 Προσομοίωση του ημιαθροιστή με χρήση Verilog

Ο ημιαθροιστής δέχεται δύο εισόδους του ενός bit, αρχικοποιημένες στο μηδέν, οι οποίες είναι οι αριθμοί που προστίθενται. Εξάγει δύο εξόδους του ενός bit, το

άθροισμα και το κρατούμενο. Παρακάτω θα πραγματοποιηθεί η προσομοίωση του ημιαθροιστή.

```
module half_adder_tb();
    wire carry, sum;
    reg a, b;
half_adder dut (.a(a), .b(b), .carry(carry), .sum(sum)); // initialization
    initial begin
        a <= 0;
        b <= 0;
    end
    always #25 a <= $random; // every 25ns value of input is ran1ndomised
    always #25 b <= $random;
    always @(a or b) // if any input changes the message appears
$monitor ("At time %4d, a is %d, b is %d, sum is %d and carry is %d", $time, a, b, sum, carry);
endmodule
```

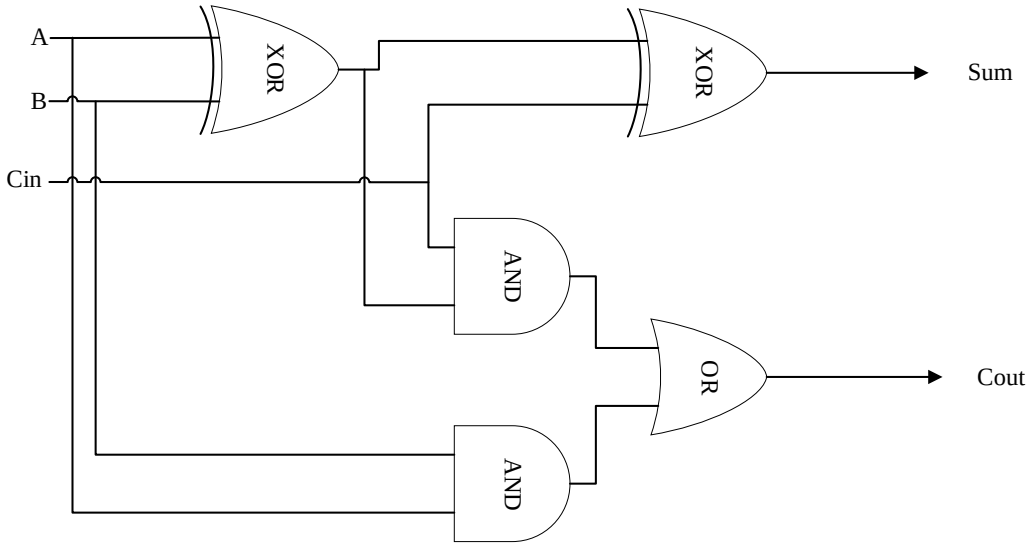
Σχήμα 3.6 Testbench ημιαθροιστή με Verilog



Σχήμα 3.7 Προσομοίωση του ημιαθροιστή με Verilog

3.2.2 Περιγραφή του πλήρη αθροιστή με χρήση VHDL και Verilog

Ένας πλήρης αθροιστής αποτελεί την λογική εξέλιξη ενός ημιαθροιστή με δυνατότητα εμφάνισης αθροίσματος και κρατουμένου μαζί. Έστω οι είσοδοι x, y, cin και έξοδοι, το άθροισμα S και το κρατούμενο cout. Η υλοποίηση τους μπορεί να γίνει με την χρήση ορισμένων λογικών παραστάσεων, για το κρατούμενο $c_{out} = x \cdot y + (x \oplus y) \cdot c_{in}$ και το άθροισμα $s = (x \oplus y) \oplus c_{in}$.



Σχήμα 3.8 Κύκλωμα πλήρους αθροιστή

3.2.2.1 Περιγραφή του πλήρους αθροιστή με χρήση VHDL

Στη συνέχεια ακολουθεί η υλοποίηση του πλήρη αθροιστή σε VHDL κώδικα.

```
LIBRARY ieee;

USE ieee.std_logic_1164.all;


ENTITY full_add IS

PORT (x, y, cin : IN  BIT;

      s, cout: OUT BIT);

END full_add;
```

```
ARCHITECTURE LogicFunc OF full_add IS
BEGIN
    s <= (x XOR y) XOR cin ;
    cout <= (x AND y) OR ((x XOR y) AND cin);
END LogicFunc;
```

Σχήμα 3.9 Περιγραφή πλήρους αθροιστή με VHDL

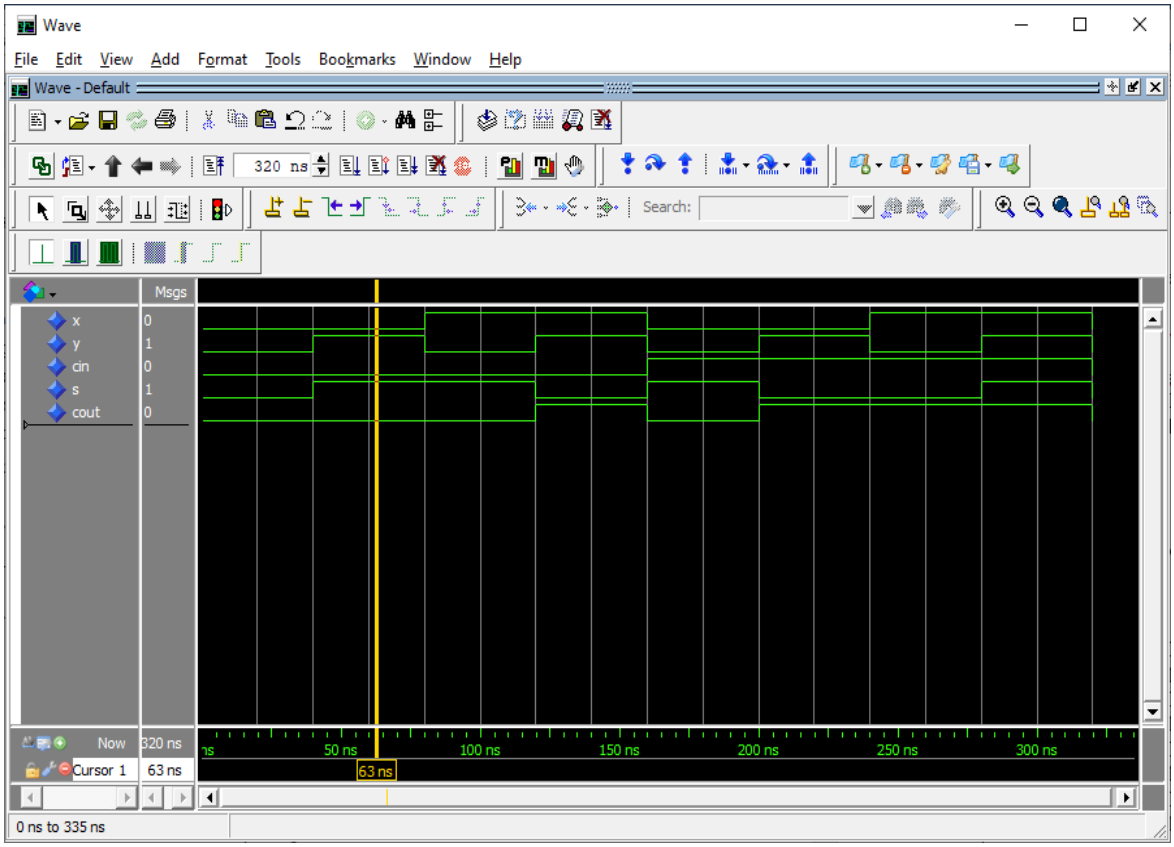
3.2.2.2 Προσομοίωση του πλήρους αθροιστή με χρήση VHDL

Παρακάτω αποτυπώνεται η προσομοίωση του πλήρη αθροιστή με ένα Testbench με όλες τις πιθανές τιμές εισόδου που μπορεί να δεχτεί ανά 40 ns (νανοδευτερόλεπτα). Επίσης, το αποτέλεσμα της προσομοίωσης με ένα Waveform.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;


ENTITY full_addtb IS
END full_addtb;


ARCHITECTURE behavior OF full_addtb IS


SIGNAL x, y, s, cin, cout : BIT;


COMPONENT full_add
PORT (x, y, cin : IN BIT;
      s, cout: OUT BIT);
END COMPONENT;


 BEGIN
```

```
m1: full_add PORT MAP(x=>x,y=>y,cin=>cin,s=>s,cout=>cout);


PROCESS
BEGIN
 cin<='0';x<='0';y<='0'; WAIT FOR 40 ns;
 cin<='0';x<='0';y<='1'; WAIT FOR 40 ns;
 cin<='0';x<='1';y<='0'; WAIT FOR 40 ns;
 cin<='0';x<='1';y<='1'; WAIT FOR 40 ns;
 cin<='1';x<='0';y<='0'; WAIT FOR 40 ns;
 cin<='1';x<='0';y<='1'; WAIT FOR 40 ns;
 cin<='1';x<='1';y<='0'; WAIT FOR 40 ns;
 cin<='1';x<='1';y<='1'; WAIT FOR 40 ns;
 END PROCESS;
END behavior;
```

Σχήμα 3.10 Testbench πλήρους αθροιστή με Verilog



Σχήμα 3.11 Προσομοίωση του πλήρους αθροιστή με VHDL

3.2.2.3 Περιγραφή του πλήρη αθροιστή με χρήση Verilog

Με βάση τα παραπάνω πραγματοποιείται η παρακάτω υλοποίηση με Verilog.

```
module full_adder( x, y, cin, S, cout); //setting wire inputs and reg output
        input wire x, y, cin;
        output reg S, cout;
        always @(x or y or cin) //for any change in any input the following happens
begin
                S = x ^ y ^ cin; //assignment to outputs using logical expressions
                cout = x&y | ((x^y) & cin);
        end
endmodule
```

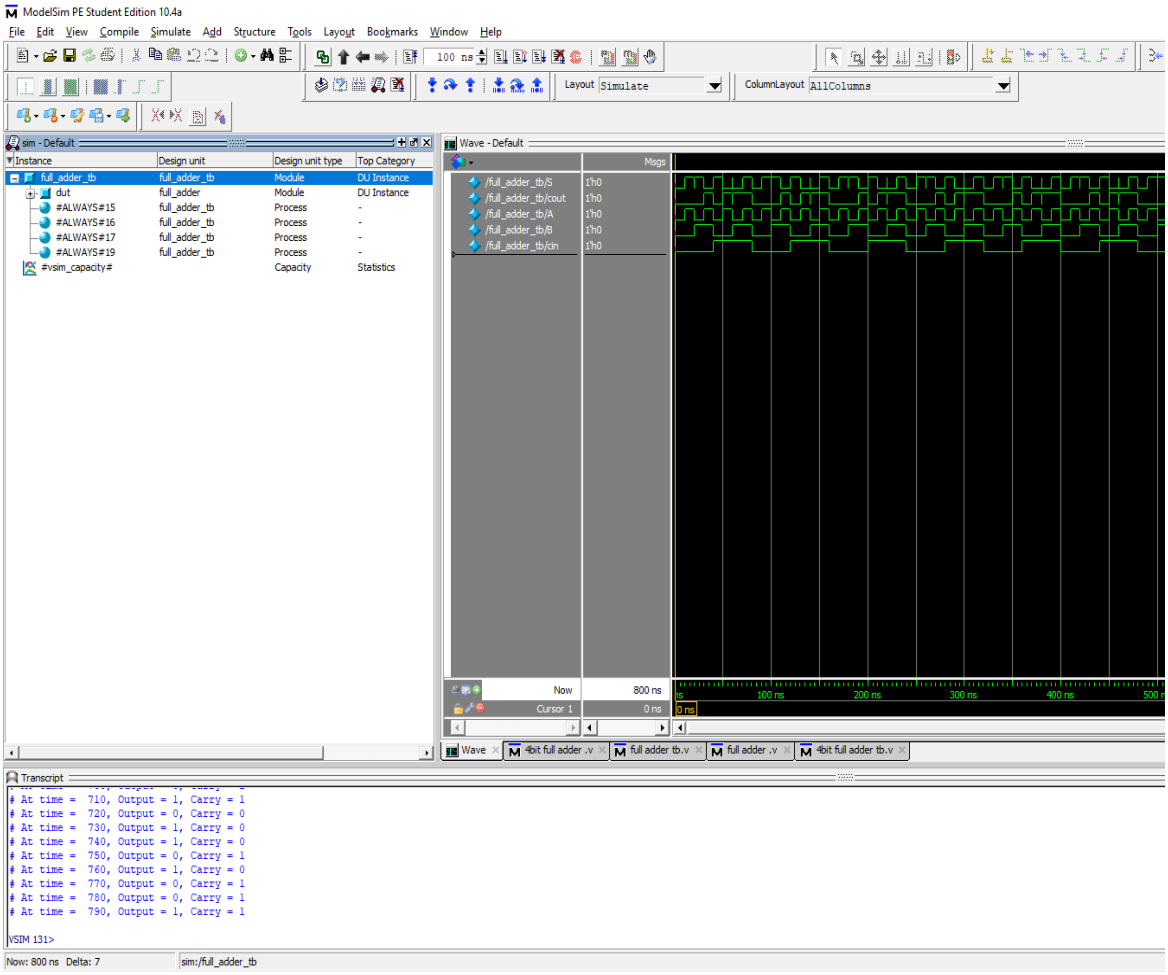
Σχήμα 3.12 Περιγραφή πλήρους αθροιστή με Verilog

3.2.2.4 Προσομοίωση του πλήρους αθροιστή με χρήση Verilog

Ο πλήρης αθροιστής δέχεται τρεις εισόδους του ενός bit, αρχικοποιημένες στο μηδέν, οι οποίες είναι οι αριθμοί που προστίθενται και ένα κρατούμενο. Εξάγει δύο εξόδους του ενός bit, το συνολικό άθροισμα και ένα κρατούμενο. Παρακάτω θα πραγματοποιηθεί η προσομοίωση του πλήρους αθροιστή.

```
module full_adder_tb(); //setting inputs and output
        wire S, cout;
        reg A, B, cin;
full_adder dut(.A(A), .B(B), .S(S), .cin(cin), .cout(cout)); // instantiation by port name.


        initial begin // initialization
                cin <= 0;
                A <= 0;
                B <= 0;
        end
                always #10 A = ~A; // every set time in ns the value of inputs changes
                always #20 B <= ~B;
                always #40 cin <= ~cin;


        always@(A or B or cin)// if any input changes the message appears
$monitor("At time = %4d, Output = %d, Carry = %d", $time, S, cout); // message for extra info
endmodule
```

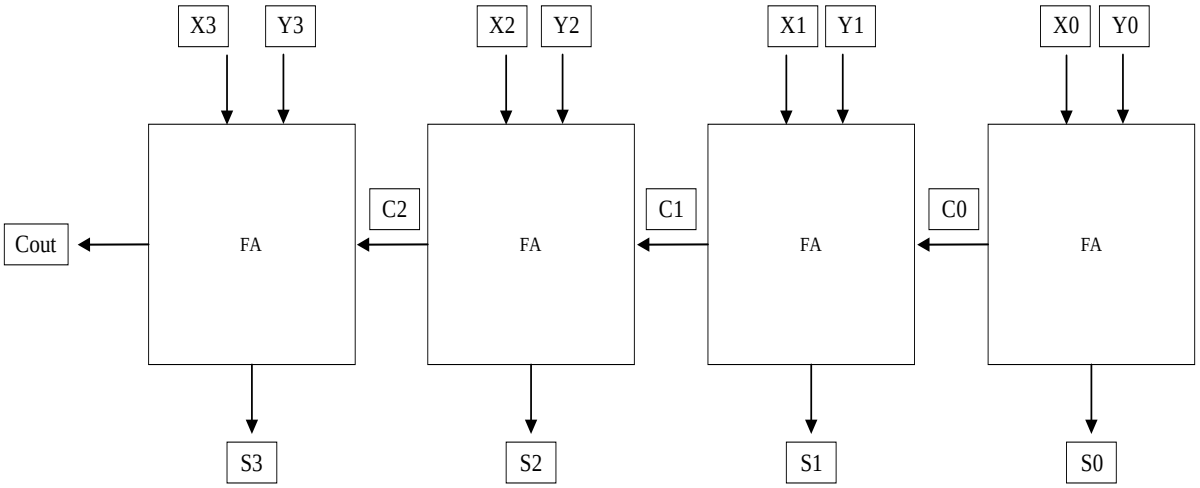
Σχήμα 3.13 Testbench πλήρους αθροιστή με Verilog



Σχήμα 3.14 Προσομοίωση του πλήρους αθροιστή με Verilog

3.2.3 Περιγραφή αθροιστή με διάδοση κρατουμένου των 4bit με χρήση συνεχόμενης αντιστοίχισης με τις VHDL και Verilog

Ένας αθροιστής με διάδοση κρατουμένου πολλαπλών bit, σχεδιάζεται συνδέοντας παράλληλα τον αντίστοιχο αριθμό από πλήρεις αθροιστές ώστε το κρατούμενο εισόδου του ενός να είναι το κρατούμενο εξόδου του επόμενου. Στο σχήμα 3.16 παρουσιάζεται η σχεδίαση ενός αθροιστή 4bit με κρατούμενο εισόδου χρησιμοποιώντας 4 πλήρεις αθροιστές. Ο αθροιστής αυτός πραγματοποιεί την πρόσθεση των αριθμών 4bit $X=x_3x_2x_1x_0$ και $Y=y_3y_2y_1y_0$ και ένα κρατούμενο εισόδου c_{in} και παράγει το άθροισμα $S=s_3s_2s_1s_0$, και ένα κρατούμενο εξόδου c_{out}. Στο παρακάτω σχήμα εκτελείται η πράξη $c_{out}2^4+S=X+Y+c_{in}$ και με c_0, c_1, c_2 τα ενδιάμεσα κρατούμενα.



Σχήμα 3.15 Σχεδίαση αθροιστή 4bit με την χρήση πλήρεις αθροιστών.

3.2.3.1 Περιγραφή αθροιστή με διάδοση κρατουμένου των 4bit με χρήση συνεχόμενης αντιστοίχισης με VHDL

Ακολουθεί η περιγραφή του αθροιστή με διάδοση κρατουμένου των 4bit με χρήση συνεχόμενης αντιστοίχισης σε VHDL κώδικα.

```
LIBRARY ieee;

USE ieee.std_logic_1164.all;


ENTITY rc_adder IS

PORT ( cin          : IN STD_LOGIC;

       x0, x1, x2, x3 : IN STD_LOGIC;

       y0, y1, y2, y3 : IN STD_LOGIC;

       s0, s1, s2, s3, cout: OUT STD_LOGIC

     );

 END rc_adder;


ARCHITECTURE Behavioral OF rc_adder IS

SIGNAL c0, c1, c2, c3 : STD_LOGIC := '0';

  BEGIN
```

```
 s0 <= (x0 XOR y0) XOR cin;
 c0 <= (x0 AND y0) OR ((x0 XOR y0) AND cin);
 s1 <= x1 XOR y1 XOR c0;
 c1 <= (x1 AND y1) OR ((x1 XOR y1) AND c0);
 s2 <= x2 XOR y2 XOR c1;
 c2 <= (x2 AND y2) OR ((x2 XOR y2) AND c1);
 s3 <= x3 XOR y3 XOR c2;
 c3 <= (c3 AND y3) OR ((x3 XOR y3) AND c2);
 cout <= c3;
END Behavioral;
```

Σχήμα 3.16 Περιγραφή αθροιστή με διάδοση κρατουμένου 4bit με χρήση συνεχόμενης αντιστοίχισης VHDL.

3.2.3.2 Προσομοίωση αθροιστή με διάδοση κρατουμένου 4bit με VHDL

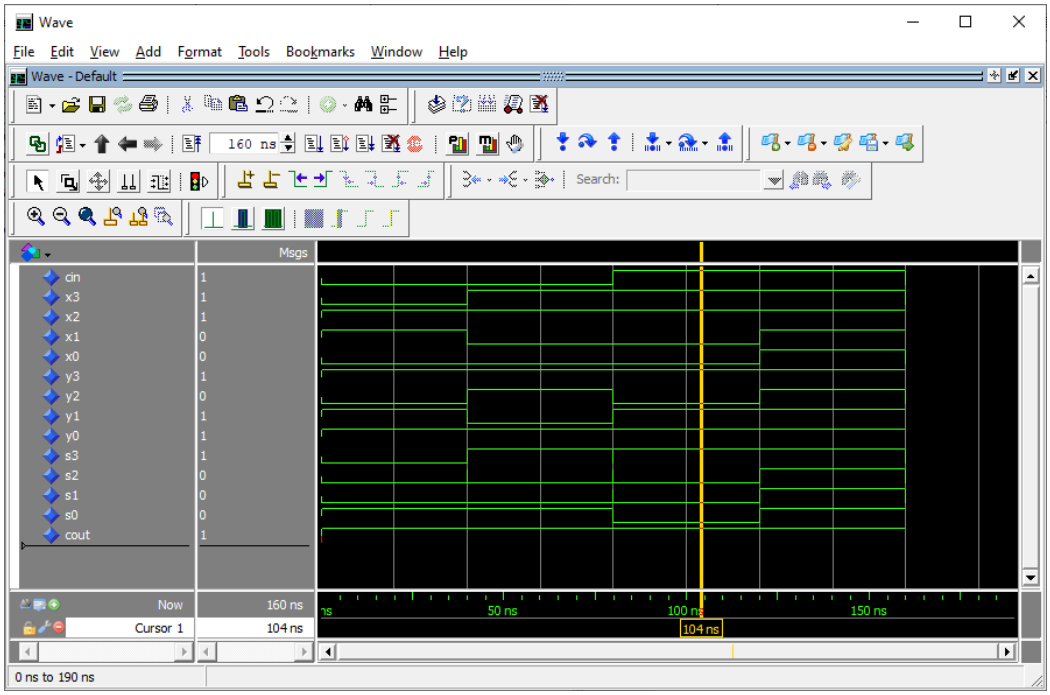
Στη συνέχεια παρατίθεται ο κώδικας της προσομοίωσης του αθροιστή με διάδοση κρατουμένου σε VHDL. Γίνεται άθροιση για τους αριθμούς 0110 και 1011 (κρατούμενο εισόδου 0), 1100 και 1101 (κρατούμενο εισόδου 0), 1100 και 1011 (κρατούμενο εισόδου 1) και τέλος 1111 και 1111 (κρατούμενο εισόδου 1). Κάθε πράξη εκτελείται ανά 40 ns (νανοδευτερόλεπτα).

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY rc_addertb IS
END rc_addertb;

ARCHITECTURE behavior OF rc_addertb IS

SIGNAL cin, cout : STD_LOGIC;
```

```
SIGNAL x3, x2, x1, x0: STD_LOGIC;
SIGNAL y3, y2, y1, y0: STD_LOGIC;
SIGNAL s3, s2, s1, s0: STD_LOGIC;

COMPONENT rc_adder
PORT (cin          : IN STD_LOGIC;
     x3, x2, x1, x0 : IN STD_LOGIC;
     y3, y2, y1, y0 : IN STD_LOGIC;
     s3, s2, s1, s0 : OUT STD_LOGIC;
     cout          : OUT STD_LOGIC);
END COMPONENT;

 BEGIN

m1: rc_adder PORT MAP(cin=>cin,x3=>x3,x2=>x2,x1=>x1,x0=>x0,
y3=>y3,y2=>y2,y1=>y1,y0=>y0,s3=>s3,s2=>s2,s1=>s1,s0=>s0,cout=>cout);

PROCESS
BEGIN

 cin<='0';x3<='0';x2<='1';x1<='1';x0<='0';y3<='1';y2<='0';y1<='1';y0<='1'; WAIT
FOR 40 ns;
 cin<='0';x3<='1';x2<='1';x1<='0';x0<='0';y3<='1';y2<='1';y1<='0';y0<='1'; WAIT
FOR 40 ns;
 cin<='1';x3<='1';x2<='1';x1<='0';x0<='0';y3<='1';y2<='0';y1<='1';y0<='1'; WAIT
FOR 40 ns;
 cin<='1';x3<='1';x2<='1';x1<='1';x0<='1';y3<='1';y2<='1';y1<='1';y0<='1'; WAIT
FOR 40 ns;
```

```
END PROCESS;

END behavior;
```

Σχήμα 3.17 Testbench αθροιστή με διάδοση κρατουμένου 4bit



Σχήμα 3.18 Προσομοίωση αθροιστή με διάδοση κρατουμένου 4bit

3.2.3.3 Περιγραφή αθροιστή με διάδοση κρατουμένου των 4bit με χρήση συνεχόμενης αντιστοίχισης με Verilog

Παρακάτω γίνεται η υλοποίηση του κυκλώματος με κώδικα Verilog. Μετά την υλοποίηση αυτή ακολουθεί και η προσομοίωση μαζί με τον κώδικα ελέγχου.

```
module ripple_carry_adder (output s0, s1, s2, s3, cout, input x0, x1, x2, x3, y0, y1 , y2 , y3, cin);

wire c0, c1, c2,c3;

        assign s0 = (x0 ^ y0)^cin;

        assign c0 = ((x0 & y0)|(x0 ^ y0))&cin;

        assign s1 =  x1 ^ y1^c0;
```

```
        assign c1 = (x1 & y1)|((x1 ^ y1)&c0);

        assign s2 =  x2 ^ y2^c1;

        assign c2 = (x2 & y2)|((x2 ^ y2)&c1);

        assign s3 =  x3 ^ y3^c2;

        assign c3 = (x3 & y3)|((x3 ^ y3)&c1);

        assign cout = c3;

    endmodule
```

Σχήμα 3.19 Περιγραφή αθροιστή με διάδοση κρατουμένου 4bit με χρήση συνεχόμενης αντιστοίχισης Verilog

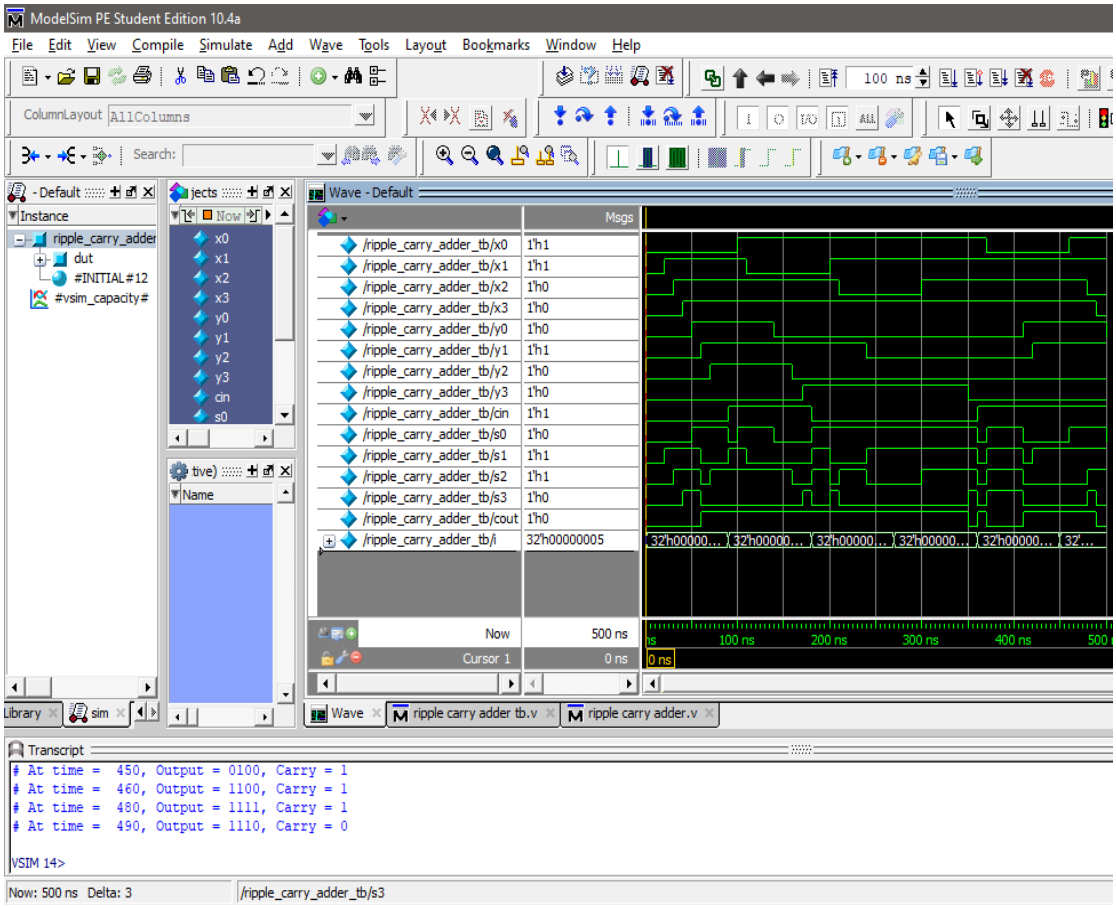
3.2.3.4 Προσομοίωση αθροιστή με διάδοση κρατουμένου 4bit με Verilog

Ο αθροιστής με διάδοση κρατουμένου δέχεται 2 εισόδους των τεσσάρων bit και μια είσοδο του ενός bit ως κρατούμενο που εισέρχεται στο κύκλωμα. Έχει δύο εξόδους, μια των τεσσάρων bit ως άθροισμα και άλλη μια ως κρατούμενο. Στον κώδικα όμως φαίνονται πολλές μικρότερες μεταβλητές από αυτές που περιγράφονται. Το παραπάνω συμβαίνει επειδή υλοποιείται ξεχωριστά κάθε αθροιστής που αποτελεί μέρος του κυκλώματος και όχι κάποια γενική υλοποίηση του κυκλώματος. Κάθε είσοδος αρχικοποιείται στο μηδέν και εισάγονται τυχαίες τιμές ανά διαστήματα μερικών νανοδευτερόλεπτων.

```
    module ripple_carry_adder_tb; //setting inputs and outputs

        reg x0, x1, x2, x3;

        reg y0, y1 , y2 , y3;

        reg cin;

        wire s0,s1,s2,s3;

        wire cout;

        integer i; //used as counter

ripple_carry_adder dut(.x0(x0), .x1(x1), .x2(x2),  .x3(x3), .y0(y0), .y1(y1), .y2(y2), .y3(y3),
.s0(s0), .s1(s1), .s2(s2), .s3(s3), .cin(cin), .cout(cout));

        initial begin // initialization
```

```
                x0 <= 0;
                x1 <= 0;
                x2 <= 0;
                x3 <= 0;
                y0 <= 0;
                y1 <= 0;
                y2 <= 0;
                y3 <= 0;
                cin <= 0;
        for (i = 0; i < 100; i = i + 1) begin
        #10 x0 <= $random; // every set time in ns the value of inputs is randomized
        #10 x1 <= $random;
        #10 x2 <= $random;
        #10 x3 <= $random;
        #10 y0 <= $random;
        #10 y1 <= $random;
        #10 y2 <= $random;
        #10 y3 <= $random;
        #10 cin <= $random;
$monitor("At time = %4d, Output = %d%d%d%d, Carry = %d", $time, s0, s1, s2, s3, cout);
                end
        end
   endmodule
```

Σχήμα 3.20 Testbench αθροιστή με διάδοση κρατουμένου 4bit



Σχήμα 3.21 Προσομοίωση αθροιστή με διάδοση κρατουμένου 4bit

3.2.4 Δομική περιγραφή αθροιστή των 4bit με διάδοση κρατουμένου με VHDL και Verilog

Χρησιμοποιείται η περιγραφή του πλήρη αθροιστή του σχήματος Χ ως υποκύκλωμα και έπειτα δημιουργείται ένα κυρίως κύκλωμα στο οποίο χρησιμοποιούμε τέσσερις φορές τον πλήρη αθροιστή. Επίσης γίνεται χρήση τριών σημάτων για τα κρατούμενα ως έξοδοι μαζί με τα αντίστοιχα αθροίσματα. Η περιγραφές δίνονται παρακάτω στο σχήματα 3.22 και 3.24.

3.2.4.1 Δομική περιγραφή αθροιστή των 4bit με διάδοση κρατουμένου με VHDL

Παρακάτω περιγράφεται ο κώδικας του αθροιστή των 4bit με διάδοση κρατουμένου σε VHDL κώδικα.

```
LIBRARY ieee;

LIBRARY work;

USE ieee.std_logic_1164.all;
```

```
USE work.full_add_package.all;


ENTITY adder_4 IS
        PORT (cin          : IN  STD_LOGIC;
          x3, x2, x1, x0 : IN  STD_LOGIC;
          y3, y2, y1, y0 : IN  STD_LOGIC;
          s3, s2, s1, s0 : OUT STD_LOGIC;
          cout           : OUT STD_LOGIC);
END adder_4;


ARCHITECTURE structured OF adder_4 IS
        SIGNAL c3, c2, c1, c0 : STD_LOGIC;
BEGIN
  FA0: full_add PORT MAP (x0, y0, cin, c0, s0);
  FA1: full_add PORT MAP (x1, y1, c0, c1, s1);
  FA2: full_add PORT MAP (x2, y2, c1, c2, s2);
  FA3: full_add PORT MAP (x3, y3, c2, c3, s3);
  cout<=c3;
END structured;
```

Σχήμα 3.22 Δομική περιγραφή αθροιστή με διάδοση κρατουμένου 4bit με VHDL

Η δημιουργία σημάτων πολλαπλών bit έχει αναλυθεί νωρίτερα στο σχήμα 2.16. Ακολουθεί η περιγραφή του πλήρους αθροιστή 4bit με χρήση σημάτων X,Y,S 4bit για είσοδο και έξοδο.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.full_add_package.all;
```

```
ENTITY adder_4B IS
        PORT (cin    : IN   STD_LOGIC;
          X, Y   : IN   STD_LOGIC_VECTOR(3 DOWNTO 0);
          S      : OUT  STD_LOGIC_VECTOR(3 DOWNTO 0) ;
          cout   : OUT  STD_LOGIC);
END adder_4B;

ARCHITECTURE structured OF adder_4B IS
        SIGNAL C : STD_LOGIC_VECTOR (3 DOWNTO 0);
BEGIN
        FA0: full_add PORT MAP (X(0), Y(0), Cin, C(0), S(0));
        FA1: full_add PORT MAP (X(1), Y(1), C(0), C(1), S(1));
      FA2: full_add PORT MAP (X(2), Y(2), C(1), C(2), S(2));
      FA3: full_add PORT MAP (X(3), Y(3), C(2), C(3), S(3));
      cout <= C(3);
END structured;
```

Σχήμα 3.23 Δομική περιγραφή αθροιστή με διάδοση κρατουμένου 4bit με VHDL

3.2.4.2 Δομική περιγραφή αθροιστή των 4bit με διάδοση κρατουμένου με Verilog

Ακολουθεί η σχεδίαση του αθροιστή 4bit με χρήση πλήρων αθροιστών με χρήση της Verilog.

```
module fulladder_4bit_using_full_adder (output reg s0, s1, s2, s3, cout,  input  x0,  x1,
x2, x3, y0, y1 , y2 , y3, cin, clk); //setting inputs and outputs
        full_adder f1 (s0, cout0, x0, y0, cin);
        full_adder f2 (s1, cout1, x1, y1, cout0);
        full_adder f3 (s2, cout2, x2, y2, cout1);
```

```
        full_adder f4 (s3, cout, x3, y3, cout2);
  endmodule
  module full_adder( A, B, cin, S, cout); //setting wire inputs and reg output
        input wire A, B, cin;
        output reg S, cout;
        always @(A or B or cin) //for any change in any input the following happens
        begin
              S = A ^ B ^ cin; //assignment to outputs using logical expressions
              cout = A&B | ((A^B) & cin);
        end
  endmodule
```

Σχήμα 3.24 Δομική περιγραφή του αθροιστή 4bit με χρήση πλήρους αθροιστών

Η δημιουργία σημάτων πολλαπλών bit έχει αναλυθεί νωρίτερα στο σχήμα 2.18. Ακολουθεί η περιγραφή του πλήρους αθροιστή 4bit με χρήση σημάτων X,Y,S 4bit για είσοδο και έξοδο.

```
  module full_adder_4b(s, cout, x, y, cin); //setting inputs and outputs
        output reg [3:0] s;
output  reg cout;
  input [3:0]  x, y;
        input  cin;
    full_adder f1 (s[0], cout0, x[0], y[0], cin); //assigning values to full adders
    full_adder f2 (s[1], cout1, x[1], y[1], cout0);
    full_adder f3 (s[2], cout2, x[2], y[2], cout1);
    full_adder f4 (s[3], cout, x[3], y[3], cout2);
  endmodule
```

```
module full_adder( A, B, cin, S, cout); //setting wire inputs and reg output
        input wire A, B, cin;
        output reg S, cout;
always @(A or B or cin) //for any change in any input the following happens
         begin
                S = A ^ B ^ cin; //assignment to outputs using logical expressions
                cout = A&B | ((A^B) & cin);
        end
  endmodule
```

Σχήμα 3.25 Τρόπος σχεδίασης προσθετή με χρήση σημάτων πολλαπλών bit

3.2.5 Περιγραφή αθροιστή 4bit με χρήση της generate με VHDL και Verilog

3.2.5.1 Περιγραφή αθροιστή 4bit με χρήση της generate με VHDL

Ο παρακάτω κώδικας περιγράφει την δημιουργία ενός αθροιστή 4bit με τη χρήση της εντολής GENERATE, η οποία επιτρέπει την δημιουργία ταυτόσημων συνιστωσών (COMPONENTS). Αυτό έχει ως αποτέλεσμα την δημιουργία τεσσάρων πλήρεις αθροιστών εκτελώντας 4bit πλήρης πρόσθεση.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;


ENTITY adder4FG IS
   PORT (  Cin : IN  STD_LOGIC ;
        X, Y : IN  STD_LOGIC_VECTOR(3 DOWNTO 0) ;
          S : OUT STD_LOGIC_VECTOR(3 DOWNTO 0) ;
        Cout : OUT STD_LOGIC ) ;
END adder4FG ;
```

```
ARCHITECTURE Structured OF adder4FG IS

SIGNAL C : STD_LOGIC_VECTOR(0 TO 4) ;


COMPONENT full_add IS

        PORT(x, y, cin : in STD_LOGIC;

                cout, s: out STD_LOGIC);

END COMPONENT;


BEGIN

  C(0) <= cin ;

  G1: FOR i IN 0 TO 3 GENERATE

  stages: full_add PORT MAP (C(i), X(i), Y(i), S(i), C(i+1)) ;

  END GENERATE;

  Cout <= C(4);

END Structured;
```

Σχήμα 3.26 Περιγραφή αθροιστή 4bit με χρήση της generate με VHDL

3.2.5.2 Περιγραφή αθροιστή 4bit με χρήση της generate με Verilog

Στην Verilog πέρα από την υλοποίηση ενός διαφορετικού module είναι δυνατόν να πραγματοποιηθεί η αντίστοιχη λειτουργία με την εντολή generate. Η εντολή generate λειτουργεί σαν βρόγχος επανάληψης μέσα στον οποίο μπορούμε να έχουμε πολλές επαναλήψεις της ίδιας υλοποίησης. Ορίζεται μεταβλητή genvar με σκοπό την χρήση της ως παράμετρο στις εν λόγω επαναλήψεις. Μέσα στον κώδικα τίθεται η παράμετρος n=4 όπου, μπορεί να μεταβληθεί όποτε υπάρχει επιθυμία για να αυξομείωσης στον αριθμό των bit του αθροιστή. Ακολουθεί κώδικας με την περιγραφή πλήρους αθροιστή 4bit με χρήση της εντολής generate.

```
module full_adder_generate(cin, x, y, s, cout);// inputs and outputs
parameter n=4; //setting n as parameter for multibit inputs and outputs
input cin;
input [n-1:0] x, y;
output [n-1:0] s;
output cout;
wire [n:0] c; // carry
genvar k; // generate 4 full adders with k as variable
assign c[0]=cin;
assign cout=c[n];
generate // generate 4 full adders
for(k=0; k<n; k=k+1) begin: fa
        wire w1,w2,w3;
                xor (w1, x[k], y[k]);
                xor (s[k], w1, cin);
                and (w2, w1, cin);
                and (w3, x[k], y[k]);
                or (c[k+1], w2, w3);
        end
        endgenerate
endmodule
```

Σχήμα 3.27 Περιγραφή αθροιστή 4bit με χρήση εντολής generate με Verilog

3.3 Περιγραφή προσθετή 16bit με χρήση αριθμητικής πρόσθεσης με VHDL και Verilog

Η χρήση ιεραρχικών περιγραφών είναι δύσχρηστη αναλογικά με το μέγεθος τους. Για αυτό παρακάτω χρησιμοποιείται η εντολή αριθμητικής αντιστοίχισης, κοινώς μια απλή πρόσθεση. Σε συνέχεια της περιγραφής με VHDL και Verilog θα ακολουθήσουν προσομοιώσεις και στις δύο γλώσσες.

3.3.1 Περιγραφή προσθετή 16bit με χρήση αριθμητικής πρόσθεσης με VHDL

Παρακάτω παρατηρείται ο κώδικας του 16bit προσθετή σε VHDL.

```
LIBRARY ieee ;

USE ieee.numeric_bit_unsigned.all;


ENTITY adder_16 IS

  PORT ( X, Y : IN  BIT_VECTOR(15 DOWNTO 0) ;

         S : OUT BIT_VECTOR(15 DOWNTO 0) );

END adder_16;


ARCHITECTURE Behavioral OF adder_16 IS

BEGIN

  S<=X+Y;

END Behavioral;
```

Σχήμα 3.28 Περιγραφή προσθετή 16bit με χρήση απλής πρόσθεσης με VHDL.

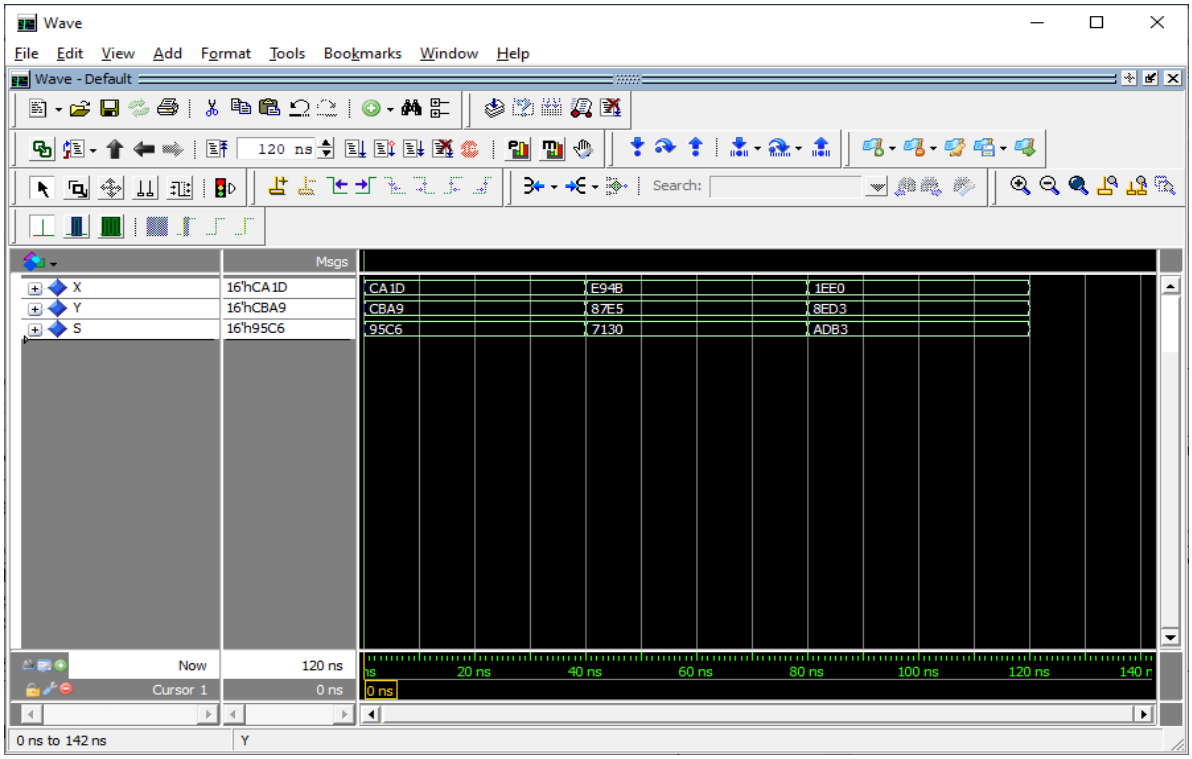
3.3.2 Προσομοίωση προσθετή 16bit με χρήση αριθμητικής πρόσθεσης με VHDL

Στη συνέχεια, αποτυπώνεται η προσομοίωση του παραπάνω κώδικα με τη χρήση ενός Testbench. Η πρόσθεση γίνεται μεταξύ των δεκαεξαδικών αριθμών CA1D και CBA9, E948 και 87E5 καθώς και 1EE0 και 8ED3 ανά 40 ns. Τέλος, τα αποτελέσματα των πράξεων αναπαρίστανται μέσω Waveform διαγράμματος.

```
LIBRARY IEEE;


ENTITY adder_16tb IS

END adder_16tb;
```

```
ARCHITECTURE behavior OF adder_16tb IS


SIGNAL X, Y, S : BIT_VECTOR (15 DOWNTO 0);


COMPONENT adder_16
PORT ( X, Y : IN  BIT_VECTOR (15 DOWNTO 0);
       S : OUT BIT_VECTOR (15 DOWNTO 0));
END COMPONENT;


 BEGIN


m1: adder_16 PORT MAP(X=>X,Y=>Y,S=>S);


PROCESS
BEGIN


 X<="1100101000011101";Y<="1100101110101001";WAIT FOR 40 ns;
 X<="1110100101001011";Y<="1000011111100101";WAIT FOR 40 ns;
 X<="0001111011100000";Y<="1000111011010011";WAIT FOR 40 ns;


 END PROCESS;
END behavior;
```

Σχήμα 3.29 Testbench προσθετή 16bit με χρήση απλής πρόσθεσης VHDL.



Σχήμα 3.30 Προσομοίωση προσθετή 16bit με χρήση απλής πρόσθεσης VHDL.

3.3.3 Περιγραφή προσθετή 16bit με χρήση αριθμητικής πρόσθεσης με Verilog

Ακολουθεί η περιγραφή του προσθετή 16bit με χρήση αριθμητικής πρόσθεσης σε Verilog κώδικα.

```
module adder_16bit (output [15:0] sum, input [15:0] x, y); //setting input and output

  assign sum = x + y;

endmodule
```

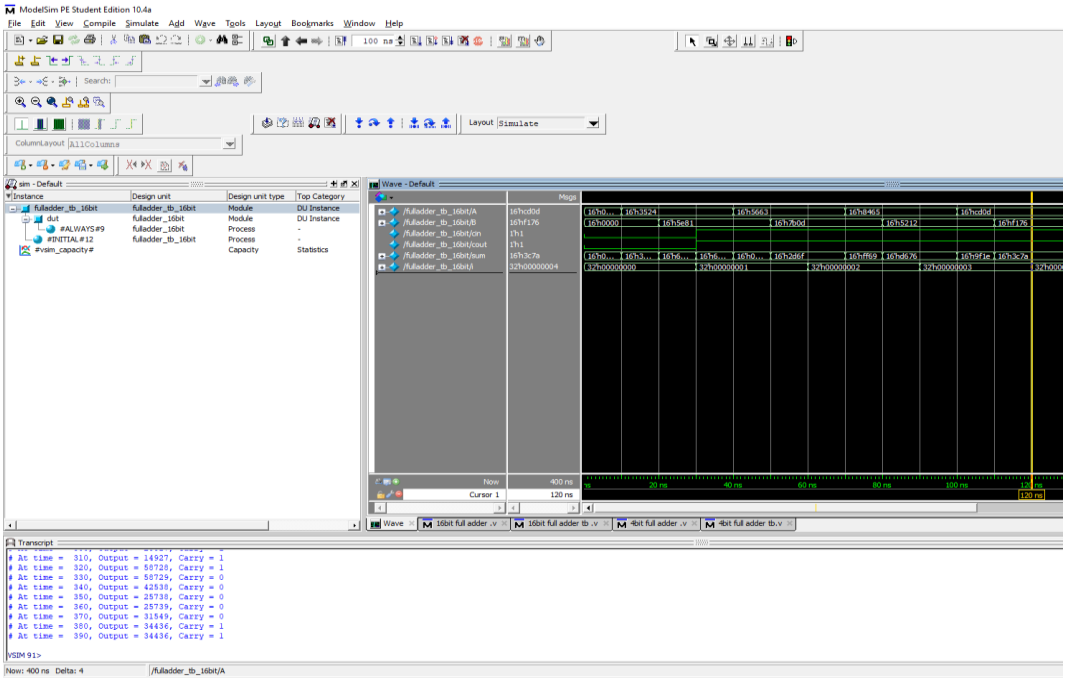
Σχήμα 3.31 Περιγραφή προσθετή 16bit με χρήση απλής πρόσθεσης με Verilog.

3.3.4 Προσομοίωση προσθετή 16bit με χρήση αριθμητικής πρόσθεσης με Verilog

Ο προσθετής των δεκαέξι bit έχει δύο εισόδους των δεκαέξι bit, που αθροίζονται και δύο εξόδους η μια εκ των οποίων έχει μήκος δεκαέξι bit και λειτουργεί ως το αποτέλεσμα της πράξης ενώ η άλλη λειτουργεί ως το κρατούμενο και έχει μήκος ένα bit. Οι είσοδοι αρχικοποιούνται στο μηδέν και δέχονται τυχαίες τιμές ανά δέκα νανοδευτερόλεπτα. Ακολουθούν το testbench και η προσομοίωση.

```
module adder_16bit_tb; //setting inputs and output
        reg [15:0] x, y;
        reg cin;
        wire cout;
        wire [15:0] sum;
adder_16bit dut(.x(x), .y(y), .cin(cin), .cout(cout), .sum(sum)); // instantiation by port name.
        initial begin // initialization
                x <= 0;
                y <= 0;
                cin <= 0;
        end
        #10 x <= $random; // every set time in ns the value of inputs is randomized
        #10 y <= $random;
        #10 cin <= $random;
        always @ (x or y or cin) begin
$monitor ("At time = %4d, Output = %d, Carry = %d", $time, sum, cout);
        end
endmodule
```

Σχήμα 3.32 Testbench προσθετή 16bit με χρήση απλής πρόσθεσης Verilog.



Σχήμα 3.33 Προσομοίωση προσθετή 16bit με χρήση απλής πρόσθεσης Verilog.

3.4 Σχεδίαση πολλαπλασιαστών με χρήση των VHDL και Verilog

Οι σύγχρονες απαιτήσεις για υψηλής ταχύτητας τρισδιάστατων γραφικών υπολογιστών και μονάδων κινητών υποδιαστολών, έχουν οδηγήσει στην ενσωμάτωση κυκλωμάτων πολλαπλασιαστών σχεδόν σε κάθε ψηφιακό επεξεργαστή σημάτων. Ωστόσο, υπάρχουν πολλές διαφορετικές αρχιτεκτονικές για την υλοποίηση πολλαπλασιαστών όπως Array και Wallace tree οι οποίες διαφέρουν κυρίως ως προς την ταχύτητα. Οι αρχιτεκτονικές Array, Carry Save, Wallace Tree και Booth θα αναπτυχθούν στη συνέχεια, ξεκινώντας με τον πολλαπλασιαστή Array. Για τις αρχιτεκτονικές θα υπάρχουν δύο κυκλώματα ένα στα τέσσερα bit και ένα στα οκτώ bit. Οι αρχιτεκτονικές Array, Carry Save, Wallace Tree θα υλοποιηθούν με χρήση μοντελοποίησης δομής και θα έχουν ως βάση ημιαθροιστές και πλήρεις αθροιστές. Ο πολλαπλασιασμός από την φύση του όντας, εν ολίγοις μια σειρά από προσθέσεις δεν απαιτεί κάτι περαιτέρω. Θα ήταν απαραίτητη και η χρήση λογικών πυλών AND στις υλοποιήσεις αλλά, επειδή ο κώδικας θα ήταν εξαιρετικά δυσανάγνωστος και μεγάλος, γίνεται χρήση του λογικού «και» (& στον κώδικα) αντί αυτών. Η αρχιτεκτονική Booth, επειδή λειτουργεί και για προσημασμένους αριθμούς η περιγραφή της θα πραγματοποιηθεί με μοντελοποίηση συμπεριφοράς καθώς υλοποιείται ως αλγόριθμος. Παρακάτω θα πραγματοποιηθεί περιγραφή πολλαπλασιαστών με χρήση της Verilog.

3.4.1 Περιγραφή πολλαπλασιαστή 4x4 με χρήση αριθμητικού πολλαπλασιασμού με VHDL και Verilog

Το παρακάτω κύκλωμα κάνει χρήση απλού πολλαπλασιασμού για να υλοποιήσει έναν πολλαπλασιαστή με μήκος οκτώ bit χρησιμοποιώντας τρόπο ίδιο όπως την παραπάνω περιγραφή του πρόσθετη μήκους δεκαέξι bit (σχήμα 3.28). Σε συνέχεια της περιγραφής με VHDL και Verilog θα ακολουθήσουν προσομοιώσεις και στις δύο γλώσσες.

3.4.1.1 Περιγραφή πολλαπλασιαστή 4x4 με χρήση αριθμητικού πολλαπλασιασμού με VHDL

Ακολουθεί η περιγραφή του πολλαπλασιαστή 4x4 με χρήση αριθμητικού πολλαπλασιασμού σε VHDL κώδικα.

```
LIBRARY IEEE;

USE ieee.numeric_bit_unsigned.all;


ENTITY mul4by4 IS

  PORT

  (

    x, y: IN BIT_vector(3 DOWNTO 0);

    z: OUT BIT_vector(7 DOWNTO 0)

  );

END ENTITY mul4by4;

ARCHITECTURE Behavioral OF mul4by4 IS

BEGIN


  z <= x * y;


END ARCHITECTURE Behavioral;
```

Σχήμα 3.34 Περιγραφή πολλαπλασιαστή 4x4 με χρήση αριθμητικού πολλαπλασιασμού VHDL.

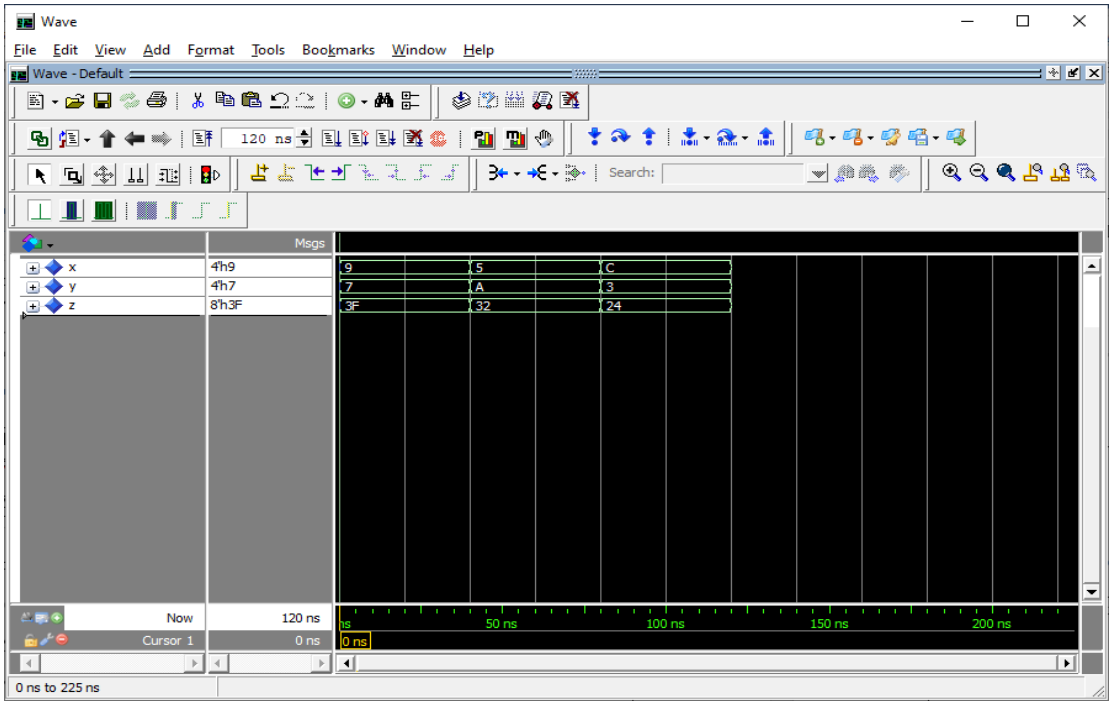
3.4.1.2 Προσομοίωση πολλαπλασιαστή 4x4με χρήση αριθμητικού πολλαπλασιασμού με την VHDL

Η προσομοίωση του αριθμητικού κυκλώματος γίνεται με την χρήση Testbench. Επιπλέον, ακολουθεί το Waveform με τα αποτελέσματα σύμφωνα με τις εισόδους που έχουν αποδοθεί από το Testbench ανά 40ns.

```
LIBRARY IEEE;


ENTITY mul4by4tb IS

END mul4by4tb;


ARCHITECTURE behavior OF mul4by4tb IS


SIGNAL x, y : BIT_VECTOR(3 DOWNTO 0);

SIGNAL z : BIT_VECTOR(7 DOWNTO 0);


COMPONENT mul4by4

PORT ( x, y : IN  BIT_VECTOR(3 DOWNTO 0);

        z : OUT BIT_VECTOR(7 DOWNTO 0));

END COMPONENT;


 BEGIN


m1: mul4by4 PORT MAP(x=>x,y=>y,z=>z);


PROCESS

BEGIN
```

```
x<="1001";y<="0111";WAIT FOR 40 ns;

x<="0101";y<="1010";WAIT FOR 40 ns;

x<="1100";y<="0011";WAIT FOR 40 ns;


END PROCESS;

END behavior;
```

Σχήμα 3.35 Testbench πολλαπλασιαστή 4x4 με χρήση αριθμητικού πολλαπλασιασμού VHDL.



Σχήμα 3.36 Προσομοίωση πολλαπλασιαστή 4x4 με χρήση αριθμητικού πολλαπλασιασμού VHDL.

3.4.1.3 Περιγραφή πολλαπλασιαστή 4x4 με χρήση αριθμητικού πολλαπλασιασμού με Verilog

Ακολουθεί η περιγραφή του πολλαπλασιαστή 4x4 με χρήση αριθμητικού πολλαπλασιασμού με χρήση κώδικα Verilog.

```
module multiplier_4x4 (output [7:0] out, input [3:0] a,[3:0] b);

        assign out = a * b;
```

```
endmodule
```

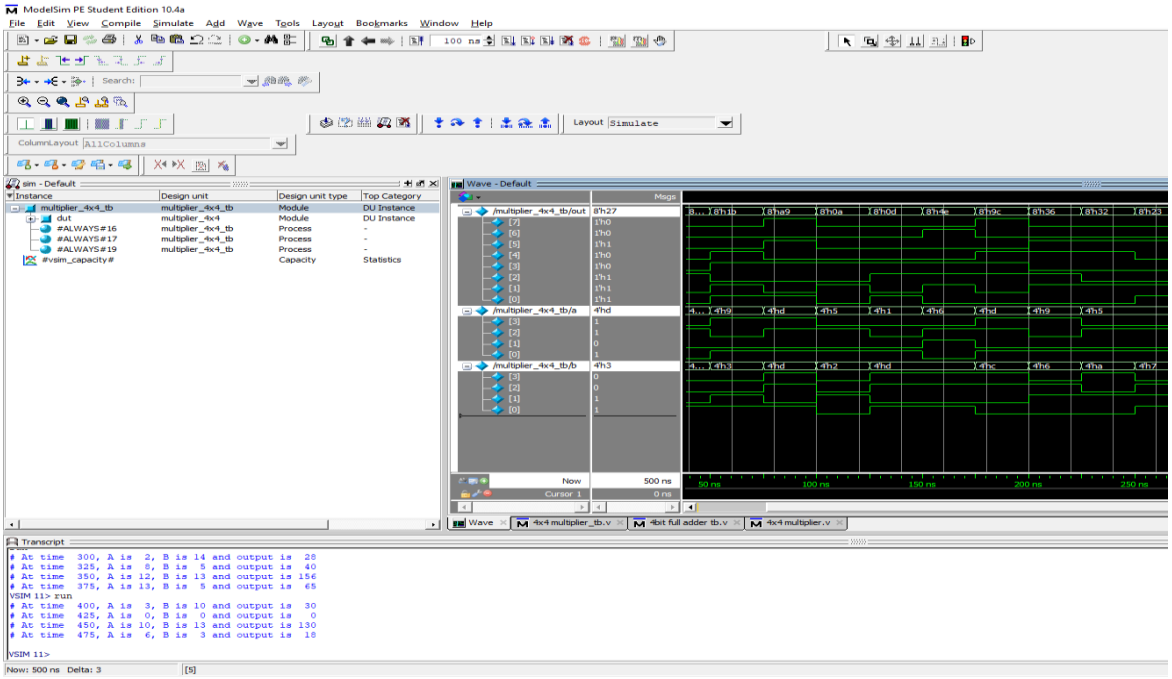
Σχήμα 3.37 Περιγραφή πολλαπλασιαστή 4x4 με χρήση αριθμητικού πολλαπλασιασμού Verilog.

3.4.1.4 Προσομοίωση πολλαπλασιαστή 4x4με χρήση αριθμητικού πολλαπλασιασμού με την Verilog

Θα ακολουθήσουν ο κώδικας ελέγχου και η προσομοίωση του πολλαπλασιαστή 4x4. Έχει δύο εισόδους μήκους τεσσάρων bit έκαστη και μια έξοδο των οκτώ bit. Οι είσοδοι αρχικοποιούνται στο μηδέν και δέχονται τυχαίες τιμές κάθε εικοσιπέντε νανοδευτερόλεπτα.

```
module multiplier_4x4_tb(); //setting inputs and output
        wire [7:0] out;
        reg [3:0] a;
        reg [3:0] b;
multiplier_4x4 dut (.a(a), .b(b), .out(out)); // initialization
        initial begin
                a <= 0; b <= 0;
end
        always #25 a <= $random; // every 25ns value of input is randomized
        always #25 b <= $random;
        always @(a or b) // if any input changes the message appears
$monitor ("At time %4d, A is %d, B is %d and output is %d", $time, a, b, out);
endmodule
```

Σχήμα 3.38 Testbench πολλαπλασιαστή 4x4 με χρήση αριθμητικού πολλαπλασιασμού Verilog.



Σχήμα 3.39 Προσομοίωση πολλαπλασιαστή 4x4 με χρήση αριθμητικού πολλαπλασιασμού Verilog.

3.5 Περιγραφή πολλαπλασιαστών αρχιτεκτονικής array με VHDL και Verilog

Ο πολλαπλασιαστής αρχιτεκτονικής array είναι η πιο βασική αρχιτεκτονική η οποία μιμείται τον κάθετο πολλαπλασιασμό. Κάθετος πολλαπλασιασμός είναι η διαδικασία με την οποία, ο πολλαπλασιαστής πολλαπλασιάζει τον πολλαπλασιαστέο ανά ψηφίο. Η εκάστοτε σειρά από ψηφία, τοποθετείται σε μια στήλη με κάθε επόμενη σειρά μετακινημένη κατά ένα ψηφίο αριστερά. Για να εξαχθεί το αποτέλεσμα προστίθενται όλες οι σειρές που προέκυψαν και εξάγονται τα κρατούμενα.

```
   154
 X 255
-------
   770
  770
 308
-------
 39270
```

```
    10011010
  X 11111111
-------------
    10011010
   10011010
  10011010
 10011010
10011010
10011010
10011010
10011010
-------------
1001100101100110
```

Σχήμα 3.40 Πολλαπλασιασμοί σε δεκαδικό και δυαδικό σύστημα αντίστοιχα

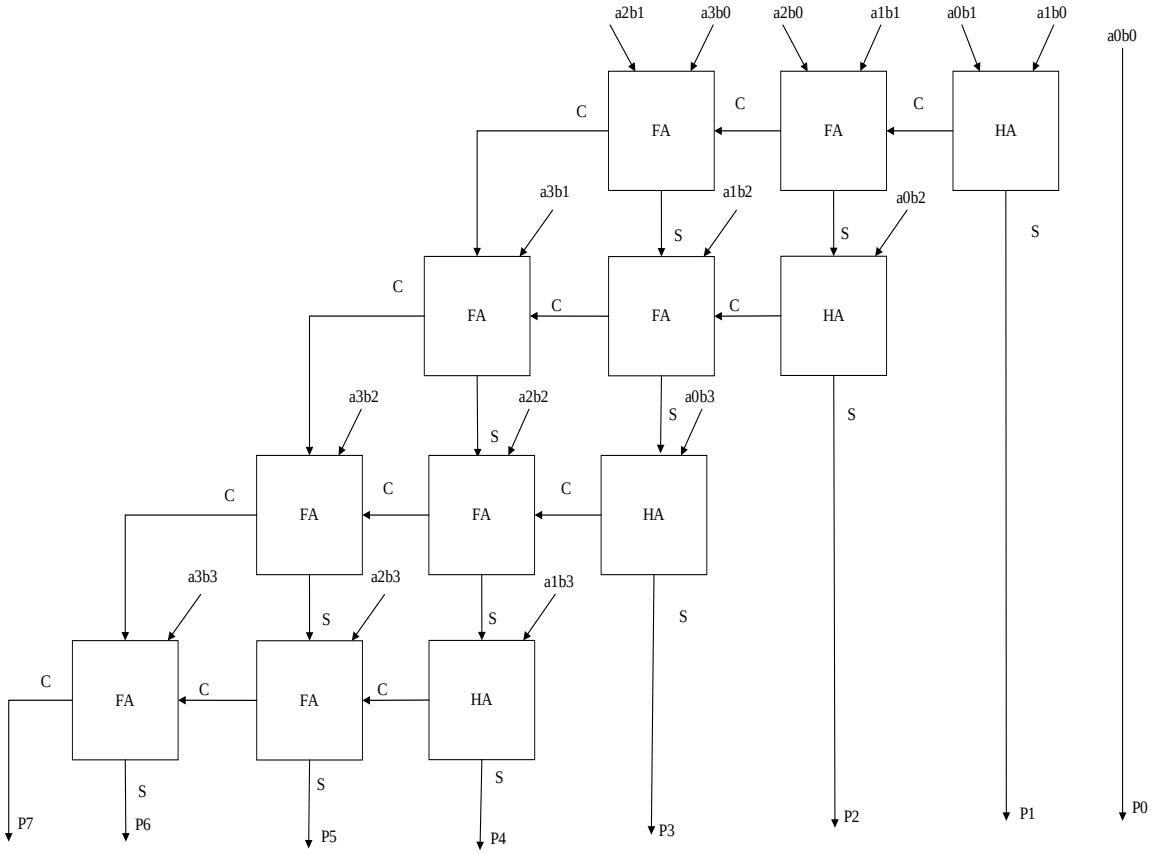
Ο Array πολλαπλασιαστής χρησιμοποιεί έναν δεδομένο αριθμό από πλήρεις και ήμι αθροιστές για να βγάλει το γινόμενο των αριθμών που δίνονται. Οι αθροιστές είναι σε διάταξη ανά σειρά όπου, ξεκινώντας από την πάνω σειρά, κάθε επόμενη βρίσκεται

κατά μια θέση αριστερότερα της προηγούμενης, ακριβώς όπως στον κάθετο πολλαπλασιασμό. Στην συγκεκριμένη αρχιτεκτονική, τα κρατούμενα προστίθενται κατά μήκος των αθροιστών και τα αθροίσματα κάθετα μέχρι να εξαχθεί το τελικό αποτέλεσμα.

```
                    a3   a2   a1   a0
                    b3   b2   b1   b0
  ------------------------------------
                  b0a3 b0a2 b0a1 b0a0
             b1a3 b1a2 b1a1 b1a0
        b2a3 b2a2 b2a1 b2a0
   b3a3 b3a2 b3a1 b3a0
  ------------------------------------
    p7   p6   p5   p4   p3   p2   p1   p0
```

Σχήμα 3.41 Παράδειγμα πολλαπλασιασμού σε αρχιτεκτονική Array

Δομικά η αρχιτεκτονική αυτή είναι η πιο εύκολη ως προς τον σχεδιασμό και την υλοποίηση, αλλά παράλληλα είναι και η πιο χρονοβόρα όσον αφορά τις πράξεις. Η περιγραφή του πολλαπλασιαστή array 4x4 γίνεται με χρήση τεσσάρων ημιαθροιστών και οκτώ πλήρων αθροιστών. Οι πύλες δεν έχουν κάποια στοίχιση μεταξύ τους.



Σχήμα 3.42 Δομή array πολλαπλασιαστή 4x4

3.5.1 Περιγραφή array πολλαπλασιαστή 4x4 με την VHDL

Ακολουθεί η περιγραφή του πολλαπλασιαστή Array 4x4.

```
LIBRARY ieee;

LIBRARY work;

USE ieee.std_logic_1164.all;

USE work.full_add_package.all;

USE work.half_add_package.all;


ENTITY array4X4 IS

        PORT (                  a3, a2, a1, a0 : IN  STD_LOGIC;

                        b3, b2, b1, b0 : IN  STD_LOGIC;

          p7,p6,p5,p4,p3, p2, p1, p0 : OUT STD_LOGIC);

END array4X4;


ARCHITECTURE structured OF array4X4 IS

        SIGNAL a1b0,a0b1,c1 : STD_LOGIC;

        SIGNAL a2b0,a1b1,a0b2,c21,c20,fp2 : STD_LOGIC;

        SIGNAL a3b0,a2b1,a1b2,a0b3,c32,c30,c31,fp30,fp31 : STD_LOGIC;

        SIGNAL a3b1,a2b2,a1b3,c42,c40,c41,fp40,hp4 : STD_LOGIC;

        SIGNAL a3b2,a2b3,c50,c51,fp50 : STD_LOGIC;

        SIGNAL a3b3 : STD_LOGIC;

        SIGNAL cout: STD_LOGIC;


BEGIN

--P0:

  p0<=a0 AND b0;
```

```
--P1:
  a1b0<=a1 AND b0;
  a0b1<=a0 AND b1;
  HA1:half_add PORT MAP (a1b0, a0b1, c1, p1);

--P2:
  a2b0<=a2 AND b0;
  a1b1<=a1 AND b1;
  FA2: full_add PORT MAP (a1b1, a2b0, c1, c20, fp2);
  a0b2<=a0 AND b2;
  HA2: half_add PORT MAP (fp2, a0b2, c21, p2);



--P3:
  a3b0<=a3 AND b0;
  a2b1<=a2 AND b1;
  FA30: full_add PORT MAP (a3b0, a2b1, c20, c30, fp30);
  a1b2<=a1 AND b2;
  FA31: full_add PORT MAP (fp30, a1b2, c21, c31, fp31);
  a0b3<=a0 AND b3;
  HA3: half_add PORT MAP (fp31, a0b3, c32, p3);

--P4:
  a3b1<=a3 AND b1;
  a2b2<=a2 AND b2;
  HA4:  half_add PORT MAP (a3b1, c30, c40, hp4);
```

```
  FA40: full_add PORT MAP (hp4, a2b2, c31, c41, fp40);
  a1b3<=a1 AND b3;
  FA41: full_add PORT MAP (fp40, a1b3, c32, c42, p4);

--P5:
  a3b2<=a3 AND b2;
  a2b3<=a2 AND b3;
  FA50: full_add PORT MAP (c40, a3b2, c41, c50, fp50);
  FA51: full_add PORT MAP (fp50, a2b3, c42, c51, p5);

--P6:
  a3b3<=a3 AND b3;
  FA6: full_add PORT MAP (a3b3, c50, c51, cout, p6);

--P7:
  p7<=cout;
END structured;
```

Σχήμα 3.43 Περιγραφή του πολλαπλασιαστή array 4x4 με χρήση VHDL.

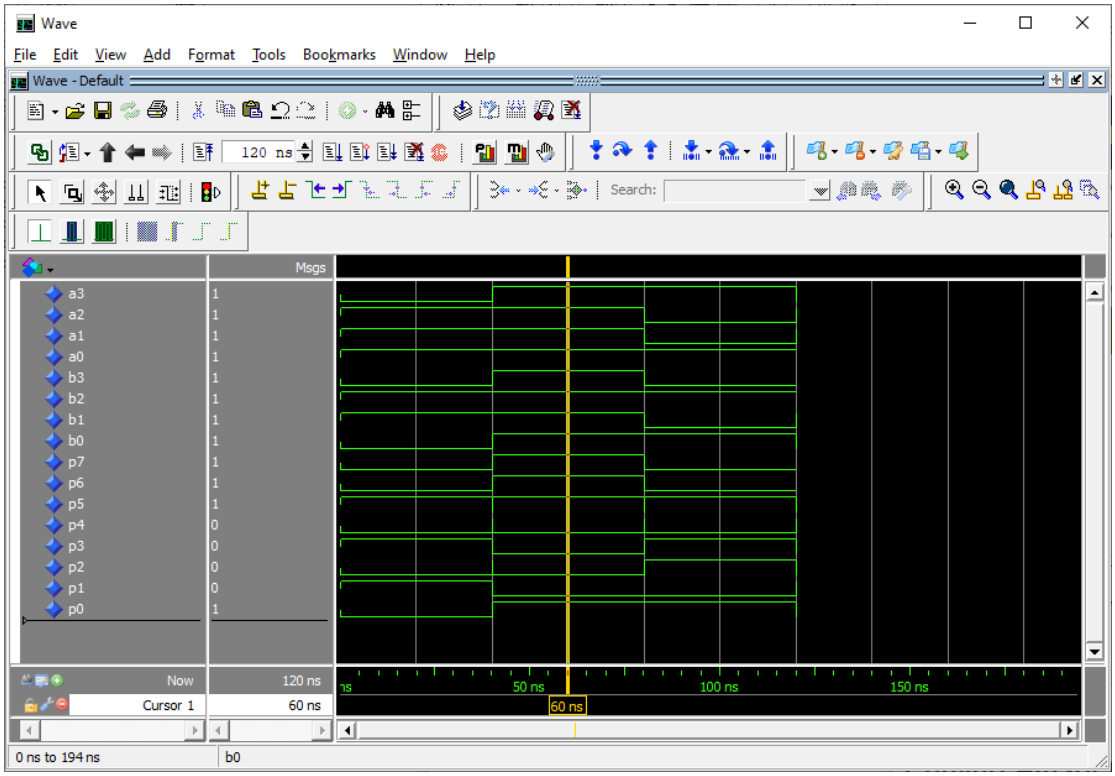
3.5.2 Προσομοίωση πολλαπλασιαστή array 4x4 με την VHDL

Με το παρακάτω Testbench γίνεται η προσομοίωση του πολλαπλασιαστή array 4x4 για τους αριθμούς 0111 και 0110, 1110 και 1111 καθώς και για 1001 και 0101. Η χρονική απόσταση μεταξύ των πράξεων είναι 40 ns. Τέλος, παρατηρούνται τα αποτελέσματα σε διάγραμμα Waveform.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY array4X4tb IS
END array4X4tb;
```

```
ARCHITECTURE behavior OF array4X4tb IS

SIGNAL a3, a2, a1, a0: STD_LOGIC;
SIGNAL b3, b2, b1, b0: STD_LOGIC;
SIGNAL p7,p6,p5,p4,p3, p2, p1, p0: STD_LOGIC;

COMPONENT array4X4
PORT (a3, a2, a1, a0: IN STD_LOGIC;
     b3, b2, b1, b0: IN STD_LOGIC;
     p7,p6,p5,p4,p3, p2, p1, p0: OUT STD_LOGIC);
END COMPONENT;
 BEGIN
  m1: array4X4 PORT MAP(a3=>a3,a2=>a2,a1=>a1,a0=>a0,
                        b3=>b3,b2=>b2,b1=>b1,b0=>b0,
p7=>p7,p6=>p6,p5=>p5,p4=>p4,p3=>p3,p2=>p2,p1=>p1,p0=>p0);
PROCESS
BEGIN
   a3<='0';a2<='1';a1<='1';a0<='1';b3<='0';b2<='1';b1<='1';b0<='0'; WAIT FOR 40 ns;
 a3<='1';a2<='1';a1<='1';a0<='1';b3<='1';b2<='1';b1<='1';b0<='1'; WAIT FOR 40 ns;
 a3<='1';a2<='0';a1<='0';a0<='1';b3<='0';b2<='1';b1<='0';b0<='1'; WAIT FOR 40 ns;
 END PROCESS;
END behavior;
```

Σχήμα 3.44 Testbench του πολλαπλασιαστή array 4x4 VHDL



Σχήμα 3.45 Προσομοίωση πολλαπλασιαστή array 4x4 VHDL.

3.5.3 Περιγραφή array πολλαπλασιαστή 4x4 με την Verilog

Ακολουθεί η περιγραφή του πολλαπλασιαστή αρχιτεκτονικής array 4x4 με Verilog.

```
module HA(output sum, carry, input a, b);
        assign sum = a^b;
        assign carry = (a&b);
endmodule
module FA(output sum, carry, input a, b, cin);
        assign sum =(a^b^cin);
        assign carry = ((a&b)|(a&cin)|(b&cin));
endmodule
module array_4x4(output [7:0] product, input [3:0] in1,in2,input clock);
        wire x1,x2,x3,x4,x5,x6,x7,x8,x9,x10,x11,x12,x13,x14,x15,x16,x17;
assign product[0]= (in1[0]&in2[0]);
HA HA1(product[1],x1,(in1[1]&in2[0]),(in1[0]&in2[1]));
```

```
 FA FA1(x2,x3,(in1[1]&in2[1]),(in1[0]&in2[2]),x1);

FA FA2(x4,x5,(in1[1]&in2[2]),(in1[0]&in2[3]),x3);

HA HA2(x6,x7,(in1[1]&in2[3]),x5);

HA HA3(product[2],x15,x2,(in1[2]&in2[0]));

FA FA5(x14,x16,x4,(in1[2]&in2[1]),x15);

FA FA4(x13,x17,x6,(in1[2]&in2[2]),x16);

FA FA3(x9,x8,x7,(in1[2]&in2[3]),x17);

HA HA4(product[3],x12,x14,(in1[3]&in2[0]));

FA FA8(product[4],x11,x13,(in1[3]&in2[1]),x12);

FA FA7(product[5],x10,x9,(in1[3]&in2[2]),x11);

FA FA6(product[6],product[7],x8,(in1[3]&in2[3]),x10);

endmodule
```

Σχήμα 3.46 Περιγραφή του πολλαπλασιαστή array 4x4 με χρήση Verilog.

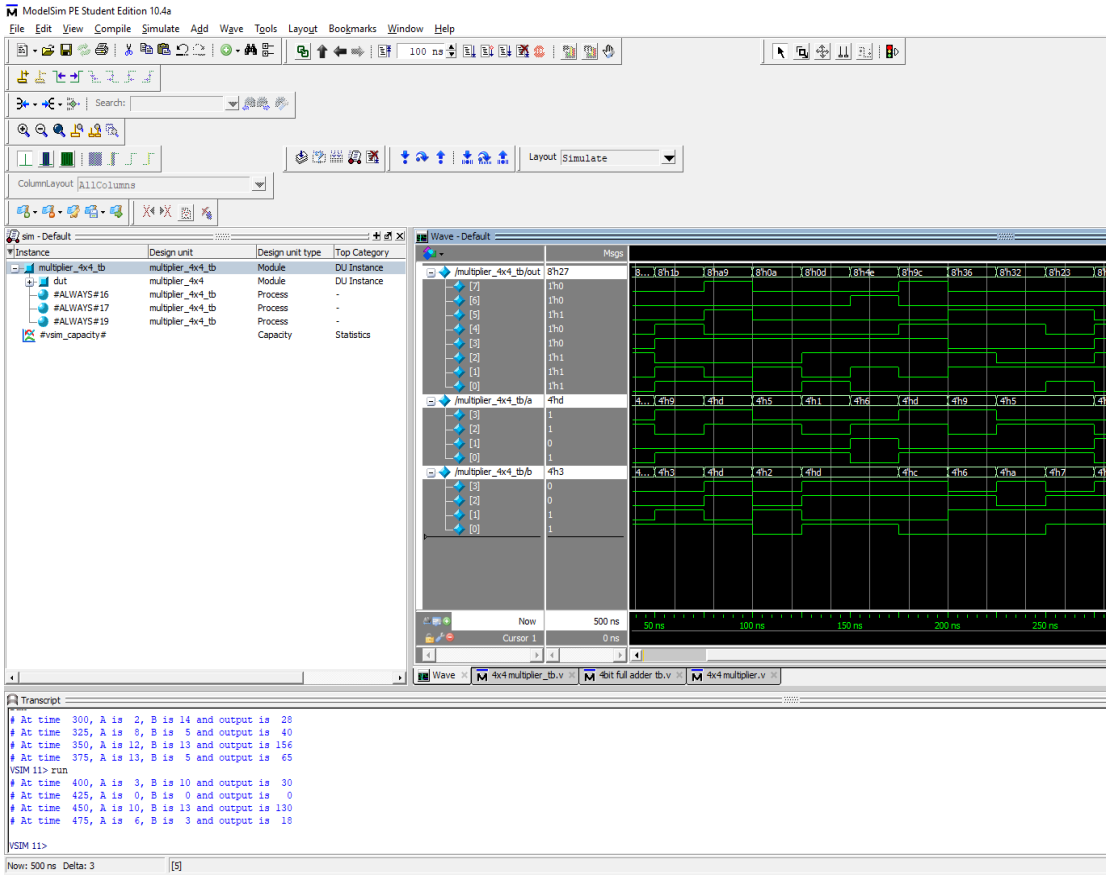
3.5.4 Προσομοίωση πολλαπλασιαστή array 4x4 με την Verilog

Ο πολλαπλασιαστής array 4x4 δέχεται δύο εισόδους των τεσσάρων bit, μια για πολλαπλασιαστή και μια για πολλαπλασιαστέο, και μια έξοδο των οκτώ bit για το γινόμενο που παράγεται. Οι είσοδοι αρχικοποιούνται με τιμή μηδέν και δέχονται αλλαγές κάθε εικοσιπέντε νανοδευτερόλεπτα. Η δομή του φαίνεται στο σχήμα 3.42. Ακολουθούν ο κώδικας ελέγχου και η προσομοίωση.

```
 module array_4x4_tb(); //setting inputs and output

        wire [7:0] product;

        reg [3:0] in1;

        reg [3:0] in2;

        reg clock;

array_4x4 dut (.in1(in1), .in2(in2), .product(product), .clock(clock)); // initialization

        initial begin
```

```
            in1 <= 0;

            in2 <= 0;

            clock <= 0;

        end

        always #100 clock <= ~clock;

always #25 in1 <= $random; // every 25ns value of input is randomized

        always #25 in2 <= $random;

        always @(in1 or in2) // if any input changes the message appears

$monitor ("At time %4d, in1 is %d, in2 is %d and output is %d", $time, in1, in2, product); // message for extra info

endmodule
```

Σχήμα 3.47 Testbench του πολλαπλασιαστή array 4x4



Σχήμα 3.48 Προσομοίωση πολλαπλασιαστή array 4x4

3.6 Περιγραφή array πολλαπλασιαστή 8x8 με VHDL και Verilog

Παρακάτω ακολουθεί το σχηματικό και η υλοποίηση του πολλαπλασιαστή array οκτώ bit, με χρήση οκτώ ημιαθροιστών και σαράντα οκτώ πλήρων αθροιστών, στοιχισμένα ανά σειρά. Η υλοποίηση πραγματοποιείται με τον ίδιο τρόπο όπως ο 4x4 αλλά μπορεί να διαχειριστεί πολύ μεγαλύτερο πλήθος τιμών.

3.6.1 Περιγραφή array πολλαπλασιαστή 8x8 με VHDL

Η περιγραφή του Array πολλαπλασιαστή 8x8 γίνεται με τον παρακάτω VHDL κώδικα.

```
LIBRARY ieee;

LIBRARY work;

USE ieee.std_logic_1164.all;

USE work.full_add_package.all;

USE work.half_add_package.all;


ENTITY array8X8 IS

        PORT (                                  a7, a6, a5, a4, a3, a2, a1, a0 :
IN  STD_LOGIC;

                                        b7, b6, b5, b4, b3, b2, b1, b0 : IN
STD_LOGIC;

          p15, p14, p13, p12, p11, p10, p9, p8, p7, p6, p5, p4, p3, p2, p1, p0 : OUT
STD_LOGIC);

END array8X8;


ARCHITECTURE structured OF array8X8 IS

        SIGNAL a1b0,a0b1,c1 : STD_LOGIC;

      SIGNAL a2b0,a1b1,a0b2,c2,c20,fp2 : STD_LOGIC;

      SIGNAL a3b0,a2b1,a1b2,a0b3,c3,c30,c31,fp31,fp30 : STD_LOGIC;

      SIGNAL a4b0,a3b1,a2b2,a1b3,a0b4,c4,c40,c41,c42,fp40,fp41,fp42 :
```

```
STD_LOGIC;

        SIGNAL
a5b0,a4b1,a3b2,a2b3,a1b4,a0b5,c5,c50,c51,c52,c53,fp50,fp51,fp52,fp53 :
STD_LOGIC;

        SIGNAL
a6b0,a5b1,a4b2,a3b3,a2b4,a1b5,a0b6,c6,c60,c61,c62,c63,c64,fp60,fp61,fp62,fp63,fp
64 : STD_LOGIC;

        SIGNAL
a7b0,a6b1,a5b2,a4b3,a3b4,a2b5,a1b6,a0b7,c7,c70,c71,c72,c73,c74,c75,fp70,fp71,fp7
2,fp73,fp74,fp75 : STD_LOGIC;

        SIGNAL
a7b1,a6b2,a5b3,a4b4,a3b5,a2b6,a1b7,c8,c80,c81,c82,c83,c84,c85,fp80,fp81,fp82,fp8
3,fp84,fp85 : STD_LOGIC;

        SIGNAL
a7b2,a6b3,a5b4,a4b5,a3b6,a2b7,c90,c91,c92,c93,c94,c95,fp90,fp91,fp92,fp93,fp94,fp
95 : STD_LOGIC;

        SIGNAL a7b3,a6b4,a5b5,a4b6,a3b7,
c100,c101,c102,c103,c104,fp100,fp101,fp102,fp103 : STD_LOGIC;

        SIGNAL a7b4,a6b5,a5b6,a4b7, c110,c111,c112,c113,fp110,fp111,fp112 :
STD_LOGIC;

        SIGNAL a7b5,a6b6,a5b7,c120,c121,c122,fp120,fp121 : STD_LOGIC;

        SIGNAL a7b6,a6b7,c130,c131,fp130 : STD_LOGIC;

        SIGNAL a7b7,c140 : STD_LOGIC;


BEGIN

--P0:

  p0<=a0 AND b0;


--P1:

  a1b0<=a1 AND b0;

  a0b1<=a0 AND b1;
```

```
  HA1:half_add PORT MAP (a1b0, a0b1, c1, p1);


--P2:
  a2b0<=a2 AND b0;
  a1b1<=a1 AND b1;
  FA2: full_add PORT MAP (a2b0, a1b1, c1, c20, fp2);
  a0b2<=a0 AND b2;
  HA2: half_add PORT MAP (a0b2, fp2, c2, p2);


--P3:
  a3b0<=a3 AND b0;
  a2b1<=a2 AND b1;
  FA30: full_add PORT MAP (a3b0, a2b1, c20, c30, fp30);
  a1b2<=a1 AND b2;
  FA31: full_add PORT MAP (fp30, a1b2, c20, c31, fp31);
  a0b3<=a0 AND b3;
  HA3: half_add PORT MAP (a0b3, fp31, c3, p3);


--P4:
  a4b0<=a4 AND b0;
  a3b1<=a3 AND b1;
  FA40: full_add PORT MAP (a4b0, a3b1, c30, c40, fp40);
  a2b2<=a2 AND b2;
  FA41: full_add PORT MAP (fp40, a2b2, c31,c41, fp41);
  a1b3<=a1 AND b3;
  FA42: full_add PORT MAP (fp41, a1b3, c3,c42, fp42);
```

```
  a0b4<=a0 AND b4;
  HA4:  half_add PORT MAP (fp42, a0b4, c4, p4);

--P5:
  a5b0<=a5 AND b0;
  a4b1<=a4 AND b1;
  FA50: full_add PORT MAP (a5b0, a4b1, c40, c50, fp50);
  a3b2<=a3 AND b2;
  FA51: full_add PORT MAP (fp50, a3b2, c41, c51, fp51);
  a2b3<=a2 AND b3;
  FA52: full_add PORT MAP (fp51, a2b3, c42, c52, fp52);
  a1b4<=a1 AND b4;
  FA53: full_add PORT MAP (fp52, a1b4, c4, c53, fp53);
  a0b5<=a0 AND b5;
  HA5:  half_add PORT MAP (fp53, a0b5, c5, p5);
--P6:
  a6b0<=a6 AND b0;
  a5b1<=a5 AND b1;
  FA60: full_add PORT MAP (a6b0, a5b1, c50, c60, fp60);
  a4b2<=a4 AND b2;
  FA61: full_add PORT MAP (fp60, a4b2, c51, c61, fp61);
  a3b3<=a3 AND b3;
  FA62: full_add PORT MAP (fp61, a3b3, c52, c62, fp62);
  a2b4<=a2 AND b4;
  FA63: full_add PORT MAP (fp62, a2b4, c53, c63, fp63);
  a1b5<=a1 AND b5;
```

```
  FA64: full_add PORT MAP (fp63, a1b5, c5, c64, fp64);
  a0b6<=a0 AND b6;
  HA6:  half_add PORT MAP (fp64, a0b6, c6, p6);

--P7:
  a7b0<=a7 AND b0;
  a6b1<=a6 AND b1;
  FA70: full_add PORT MAP (a7b0, a6b1, c60, c70, fp70);
  a5b2<=a5 AND b2;
  FA71: full_add PORT MAP (fp70, a5b2, c61, c71, fp71);
  a4b3<=a4 AND b3;
  FA72: full_add PORT MAP (fp71, a4b3, c62, c72, fp72);
  a3b4<=a3 AND b4;
  FA73: full_add PORT MAP (fp72, a3b4, c63, c73, fp73);
  a2b5<=a2 AND b5;
  FA74: full_add PORT MAP (fp73, a2b5, c64, c74, fp74);
  a1b6<=a1 AND b6;
  FA75: full_add PORT MAP (fp74, a1b6, c6, c75, fp75);
  a0b7<=a0 AND b7;
  HA7:  half_add PORT MAP (fp75, a0b7, c7, p7);

--P8:
  a7b1<=a7 AND b1;
  FA85: full_add PORT MAP (a7b1, c71, c70, c80, fp80);
  a6b2<=a6 AND b2;
  FA80: full_add PORT MAP (fp80, a6b2, c72, c81, fp81);
```

```
  a5b3<=a5 AND b3;
  FA81: full_add PORT MAP (fp81, a5b3, c73, c82, fp82);
  a4b4<=a4 AND b4;
  FA82: full_add PORT MAP (fp82, a4b4, c74, c83, fp83);
  a3b5<=a3 AND b5;
  FA83: full_add PORT MAP (fp83, a3b5, c75, c84, fp84);
  a2b6<=a2 AND b6;
  FA84: full_add PORT MAP (fp84, a2b6, c7, c85, fp85);
  a1b7<=a1 AND b7;
  H8: half_add PORT MAP (a1b7, fp85, c8, p8);



--P9:
  a7b2<=a7 AND b2;
  FA90: full_add PORT MAP (c80, a7b2, c81, c90, fp90);
  a6b3<=a6 AND b3;
  FA91: full_add PORT MAP (fp90, a6b3, c82, c91, fp91);
  a5b4<=a5 AND b4;
  FA92: full_add PORT MAP (fp91, a5b4, c83, c92, fp92);
  a4b5<=a4 AND b5;
  FA93: full_add PORT MAP (fp92, a4b5, c84, c93, fp93);
  a3b6<=a3 AND b6;
  FA94: full_add PORT MAP (fp93, a3b6, c85, c94, fp94);
  a2b7<=a2 AND b7;
  FA95: full_add PORT MAP (fp94, a2b7, c8, c95, p9);
```

```
--P10:
  a7b3<=a7 AND b3;
  FA100: full_add PORT MAP (c90, a7b3, c91, c100, fp100);
  a6b4<=a6 AND b4;
  FA7101: full_add PORT MAP (fp100, a6b4, c92, c101, fp101);
  a5b5<=a5 AND b5;
  FA102: full_add PORT MAP (fp101, a5b5, c93, c102, fp102);
  a4b6<=a4 AND b6;
  FA103: full_add PORT MAP (fp102, a4b6, c94, c103, fp103);
  a3b7<=a3 AND b7;
  FA104: full_add PORT MAP (fp103, a3b7, c95, c104, p10);

--P11:
  a7b4<=a7 AND b4;
  FA110: full_add PORT MAP (c100, a7b4, c101, c110, fp110);
  a6b5<=a6 AND b5;
  FA111: full_add PORT MAP (fp110, a6b5, c102, c111, fp111);
  a5b6<=a5 AND b6;
  FA112: full_add PORT MAP (fp111, a5b6, c103, c112, fp112);
  a4b7<=a4 AND b7;
  FA113: full_add PORT MAP (fp112, a4b7, c104, c113, p11);

--P12:
  a7b5<=a7 AND b5;
  FA120: full_add PORT MAP (c110, a7b5, c111, c120, fp120);
  a6b6<=a6 AND b6;
```

```
  FA121: full_add PORT MAP (fp120, a6b6, c112, c121, fp121);
  a5b7<=a5 AND b7;
  FA122: full_add PORT MAP (fp121, a5b7, c113, c122, p12);

--P13:
  a7b6<=a7 AND b6;
  FA130: full_add PORT MAP (c120, a7b6, c121, c130, fp130);
  a6b7<=a6 AND b7;
  FA131: full_add PORT MAP (fp130, a6b7, c122, c131, p13);

--P14:
  a7b7<=a7 AND b7;
  FA140: full_add PORT MAP (c130, a7b7, c131, c140, p14);

--p15
  p15<=c140;

END structured;
```

Σχήμα 3.49 Περιγραφή του πολλαπλασιαστή array 8bit με χρήση VHDL.

3.6.2 Προσομοίωση πολλαπλασιαστή array 8x8 VHDL

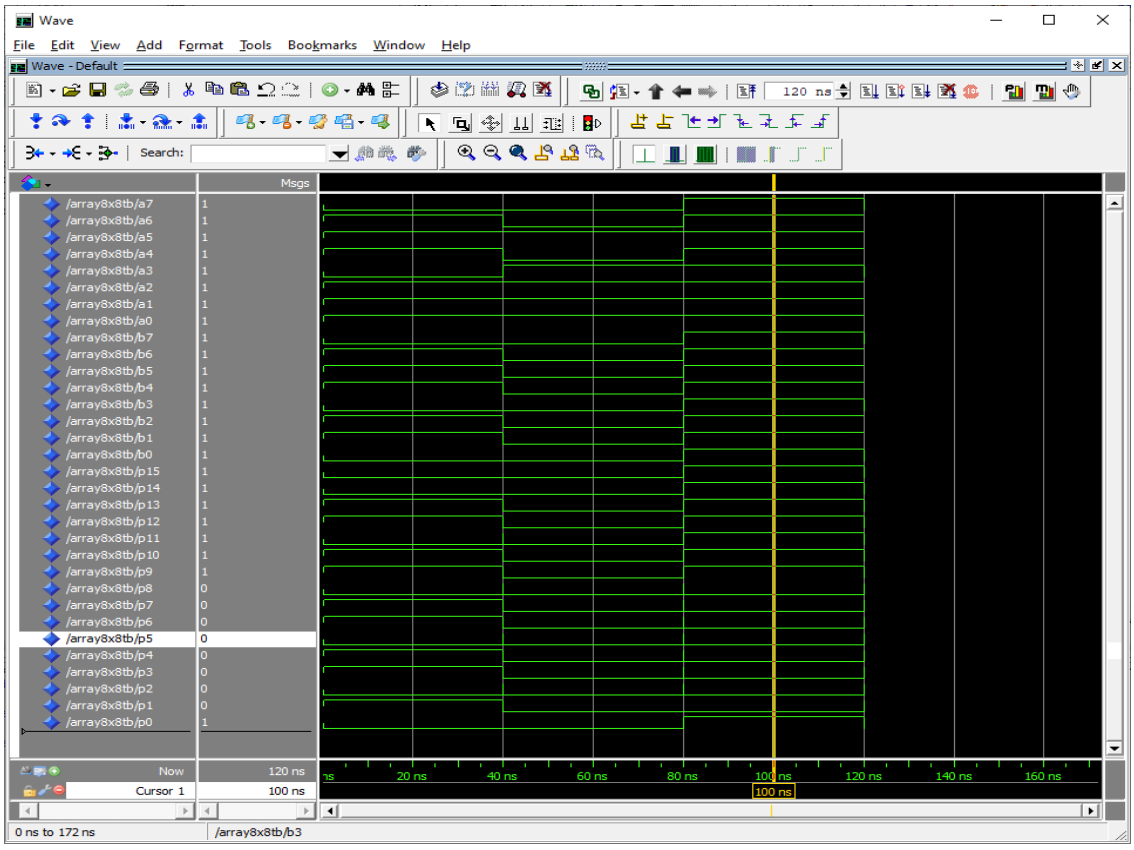
Η ακόλουθη προσομοίωση επιτυγχάνεται με Testbench για τους αριθμούς 01110111 και 01110110, 00101111 και 00000000 καθώς και 11111111 και 11111111. Τα αποτελέσματα της προσομοίωσης παρατηρούνται στο ακόλουθο διάγραμμα Waveform.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
```

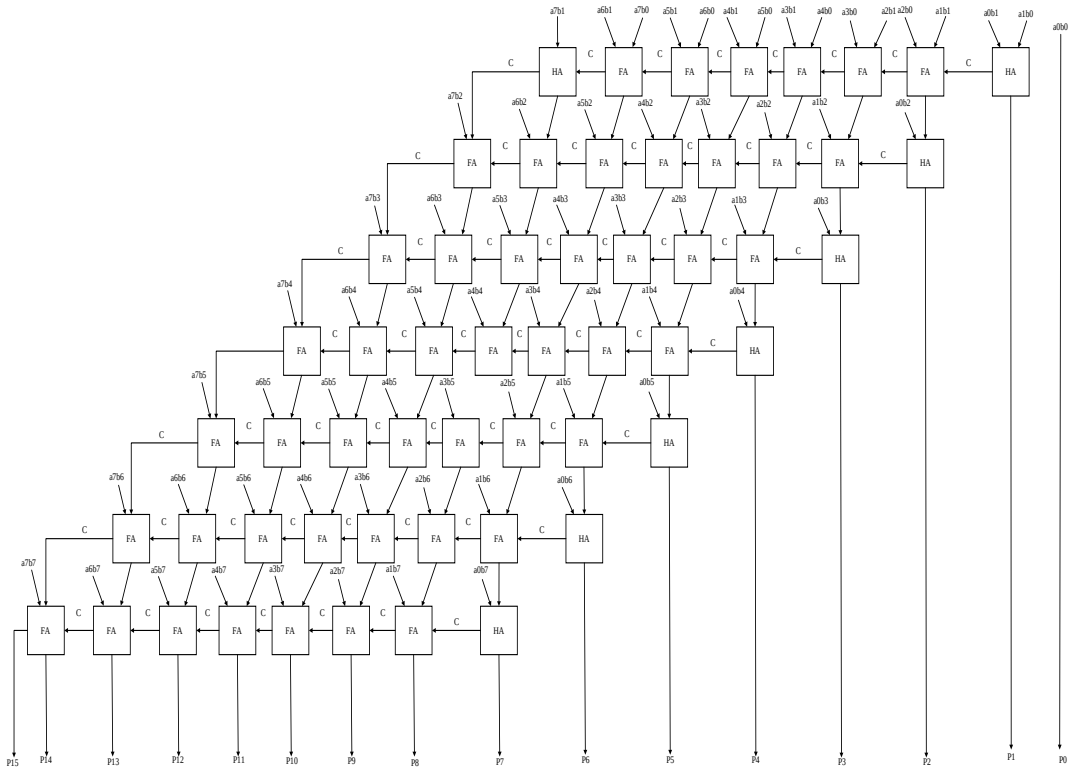
```
ENTITY array8X8tb IS

END array8X8tb;


ARCHITECTURE behavior OF array8X8tb IS


SIGNAL  a7, a6, a5, a4, a3, a2, a1, a0: STD_LOGIC;

SIGNAL  b7, b6, b5, b4, b3, b2, b1, b0: STD_LOGIC;

SIGNAL  p15, p14, p13, p12, p11, p10, p9, p8, p7, p6, p5, p4, p3, p2, p1, p0:
STD_LOGIC;


COMPONENT array8X8

PORT (a7, a6, a5, a4, a3, a2, a1, a0: IN STD_LOGIC;

     b7, b6, b5, b4, b3, b2, b1, b0: IN STD_LOGIC;

     p15, p14, p13, p12, p11, p10, p9, p8, p7, p6, p5, p4, p3, p2, p1, p0: OUT
STD_LOGIC);

END COMPONENT;


 BEGIN


m1: array8X8 PORT
MAP(a7=>a7,a6=>a6,a5=>a5,a4=>a4,a3=>a3,a2=>a2,a1=>a1,a0=>a0,

                        b7=>b7,b6=>b6,b5=>b5,b4=>b4,b3=>b3,b2=>b2,b1=>b1,b0=>b0,

p15=>p15,p14=>p14,p13=>p13,p12=>p12,p11=>p11,p10=>p10,p9=>p9,p8=>p8,p7=
>p7,p6=>p6,p5=>p5,p4=>p4,p3=>p3,p2=>p2,p1=>p1,p0=>p0);


PROCESS

BEGIN
```

```
a7<='0';a6<='1';a5<='1';a4<='1';a3<='0';a2<='1';a1<='1';a0<='1';b7<='0';b6<='1';b5<='
1';b4<='1';b3<='0';b2<='1';b1<='1';b0<='0'; WAIT FOR 40 ns;

a7<='0';a6<='0';a5<='1';a4<='0';a3<='1';a2<='1';a1<='1';a0<='1';b7<='0';b6<='0';b5<='
0';b4<='0';b3<='0';b2<='0';b1<='0';b0<='0'; WAIT FOR 40 ns;

a7<='1';a6<='1';a5<='1';a4<='1';a3<='1';a2<='1';a1<='1';a0<='1';b7<='1';b6<='1';b5<='
1';b4<='1';b3<='1';b2<='1';b1<='1';b0<='1'; WAIT FOR 40 ns;


 END PROCESS;

END behavior;
```

Σχήμα 3.50 Testbench πολλαπλασιαστή array 8x8 VHDL.



Σχήμα 3.51 Προσομοίωση πολλαπλασιαστή array 8x8 VHDL.



Σχήμα 3.52 Δομή πολλαπλασιαστή array 8x8

3.6.3 Περιγραφή array πολλαπλασιαστή 8x8 με Verilog

Ακολουθεί η περιγραφή του πολλαπλασιαστή αρχιτεκτονικής array 8x8 με Verilog.

```
module HA(output sum, carry, input a, b);
        assign sum = a^b;
        assign carry = (a&b);
endmodule
module FA(output sum, carry, input a, b, cin);
        assign sum =(a^b^cin);
        assign carry = ((a&b)|(a&cin)|(b&cin));
endmodule
module multiplier_8x8_struct(output [15:0] out, input [7:0] A, B, input clk);
        assign out[0]= (A[0]&B[0]); //p0
wire [64:0] s, c;
```

```
  //line 1
  HA HA11(product[1],c[0],(A[0]&B[1]),(A[1]&B[0])); //p1
  FA FA11(s[1],c[1],(A[1]&B[1]),(A[2]&B[0]),c[0]);
  FA FA12(s[2],c[2],(A[2]&B[1]),(A[3]&B[0]),c[1]);
  FA FA13(s[3],c[3],(A[3]&B[1]),(A[4]&B[0]),c[2]);
FA FA14(s[4],c[4],(A[4]&B[1]),(A[5]&B[0]),c[3]);
FA FA15(s[5],c[5],(A[5]&B[1]),(A[6]&B[0]),c[4]);
FA FA16(s[6],c[6],(A[6]&B[1]),(A[7]&B[0]),c[5]);
HA HA12(s[7],c[7],c[6],(A[7]&B[1]));
//line 2
HA HA21(product[2],c[8],(A[0]&B[2]),s[1]); //p2
FA FA21(s[9],c[9],(A[1]&B[2]),s[2],c[8]);
FA FA22(s[10],c[10],(A[2]&B[2]),s[3],c[9]);
FA FA23(s[11],c[11],(A[3]&B[2]),s[4],c[10]);
FA FA24(s[12],c[12],(A[4]&B[2]),s[5],c[11]);
FA FA25(s[13],c[13],(A[5]&B[2]),s[6],c[12]);
FA FA26(s[14],c[14],(A[6]&B[2]),s[7],c[13]);
FA FA27(s[15],c[15],(A[7]&B[2]),c[7],c[14]);
//line 3
HA HA31(product[3],c[16],(A[0]&B[3]),s[9]); //p3
FA FA31(s[17],c[17],(A[1]&B[3]),s[10],c[16]);
FA FA32(s[18],c[18],(A[2]&B[3]),s[11],c[17]);
FA FA33(s[19],c[19],(A[3]&B[3]),s[12],c[18]);
FA FA34(s[20],c[20],(A[4]&B[3]),s[13],c[19]);
FA FA35(s[21],c[21],(A[5]&B[3]),s[14],c[20]);
FA FA36(s[22],c[22],(A[6]&B[3]),s[15],c[21]);
```

```
FA FA37(s[23],c[23],(A[7]&B[3]),c[15],c[22]);
//line 4
HA HA41(product[4],c[24],(A[0]&B[4]),s[17]); //p4
FA FA41(s[25],c[25],(A[1]&B[4]),s[18],c[24]);
FA FA42(s[26],c[26],(A[2]&B[4]),s[19],c[25]);
FA FA43(s[27],c[27],(A[3]&B[4]),s[20],c[26]);
FA FA44(s[28],c[28],(A[4]&B[4]),s[21],c[27]);
FA FA45(s[29],c[29],(A[5]&B[4]),s[22],c[28]);
FA FA46(s[30],c[30],(A[6]&B[4]),s[23],c[29]);
FA FA47(s[31],c[31],(A[7]&B[4]),c[23],c[30]);
//line 5
HA HA51(product[5],c[32],(A[0]&B[5]),s[25]); //p5
FA FA51(s[33],c[33],(A[1]&B[5]),s[26],c[32]);
FA FA52(s[34],c[34],(A[2]&B[5]),s[27],c[33]);
FA FA53(s[35],c[35],(A[3]&B[5]),s[28],c[34]);
FA FA54(s[36],c[36],(A[4]&B[5]),s[29],c[35]);
FA FA55(s[37],c[37],(A[5]&B[5]),s[30],c[36]);
FA FA56(s[38],c[38],(A[6]&B[5]),s[31],c[37]);
FA FA57(s[39],c[39],(A[7]&B[5]),c[31],c[38]);
//line 6
HA HA61(product[6],c[40],(A[0]&B[6]),s[33]); //p6
FA FA61(s[41],c[41],(A[1]&B[6]),s[34],c[40]);
FA FA62(s[42],c[42],(A[2]&B[6]),s[35],c[41]);
FA FA63(s[43],c[43],(A[3]&B[6]),s[36],c[42]);
FA FA64(s[44],c[44],(A[4]&B[6]),s[37],c[43]);
FA FA65(s[45],c[45],(A[5]&B[6]),s[38],c[44]);
```

```
FA FA66(s[46],c[46],(A[6]&B[6]),s[39],c[45]);

FA FA67(s[47],c[47],(A[7]&B[6]),c[39],c[46]);

//line 7

HA HA71(product[7],c[48],(A[0]&B[7]),s[41]); //p7

FA FA71(product[8],c[49],(A[1]&B[7]),s[42],c[48]);//p8

FA FA72(product[9],c[50],(A[2]&B[7]),s[43],c[49]);//p9

FA FA73(product[10],c[51],(A[3]&B[7]),s[44],c[50]);//p10

FA FA74(product[11],c[52],(A[4]&B[7]),s[45],c[51]);//p11

FA FA75(product[12],c[53],(A[5]&B[7]),s[46],c[52]);//p12

FA FA76(product[13],c[54],(A[6]&B[7]),s[47],c[53]);//p13

FA FA77(product[14],product[15],(A[7]&B[7]),c[47],c[54]); //p14,15

endmodule
```

Σχήμα 3.53 Περιγραφή του πολλαπλασιαστή array 8bit με χρήση Verilog

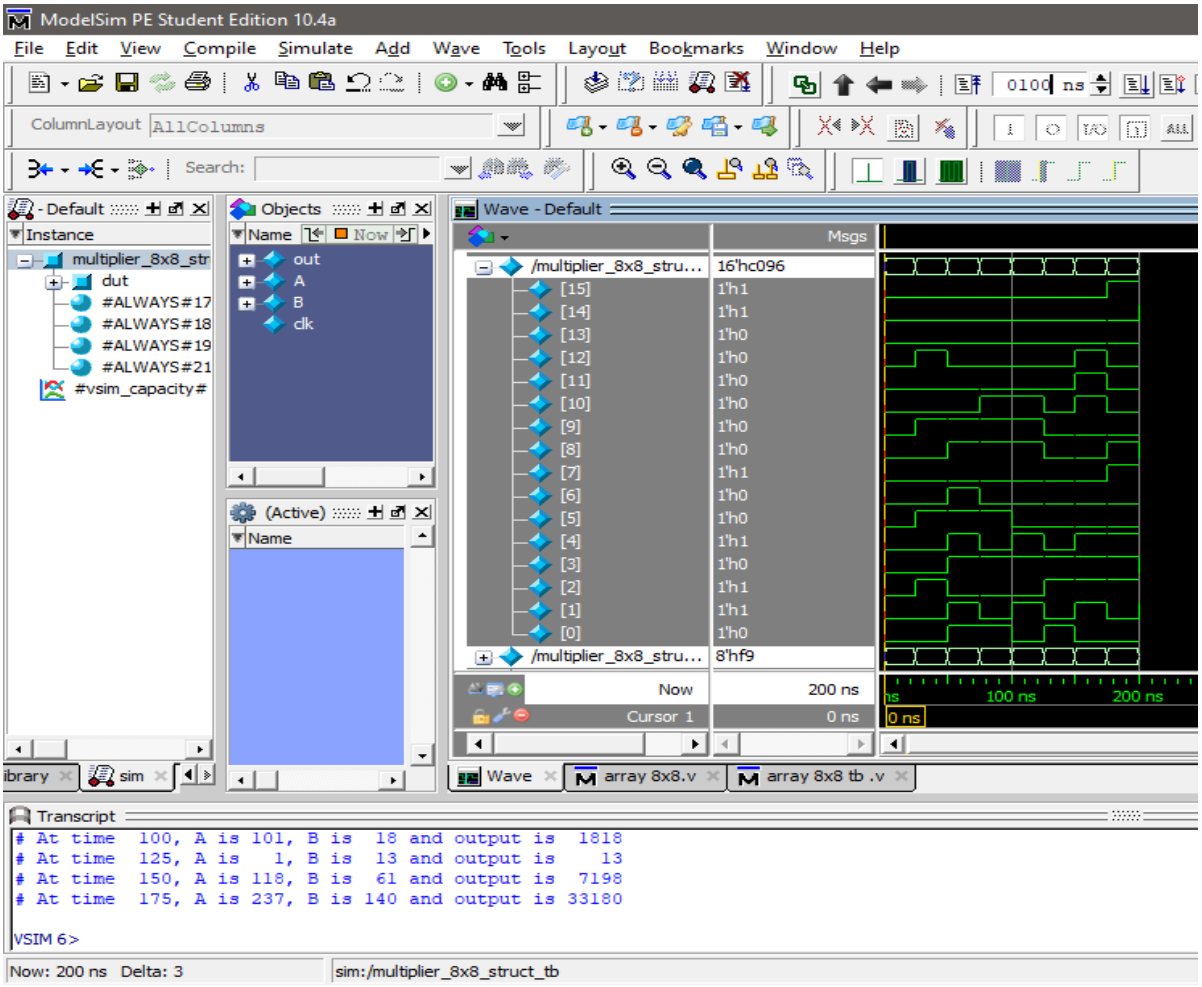
3.6.4 Προσομοίωση πολλαπλασιαστή array 8x8 Verilog

Ο πολλαπλασιαστής array 8x8 έχει την ίδια ακριβώς λειτουργικότητα με τον πολλαπλασιαστή array 4x4 αλλά είναι δομημένος για εισόδους των οκτώ bit. Οι δύο είσοδοι αρχικοποιούνται με μηδέν και αλλάζουν τυχαίες τιμές ανά εικοσιπέντε νανοδευτερόλεπτα. Η δομή του φαίνεται στο σχήμα 3.52. Ακολουθούν το testbench και η προσομοίωση.

```
module multiplier_8x8_struct_tb(); //setting inputs and output

        wire [15:0] out;

        reg [7:0] A;

        reg [7:0] B;

        reg clk;

multiplier_8x8_struct dut (.A(A), .B(B), .out(out), .clk(clk)); // initialization

        initial begin
```

```
            A <= 0;

            B <= 0;

            clk <= 0;

        end

        always #100 clk <= ~clk;

        always #25 A <= $random; // every 25ns value of input is randomized

        always #25 B <= $random;

        always @(A or B) // if any input changes the message appears

$monitor ("At time %4d, A is %d, B is %d and output is %d", $time, A, B, out);
endmodule
```

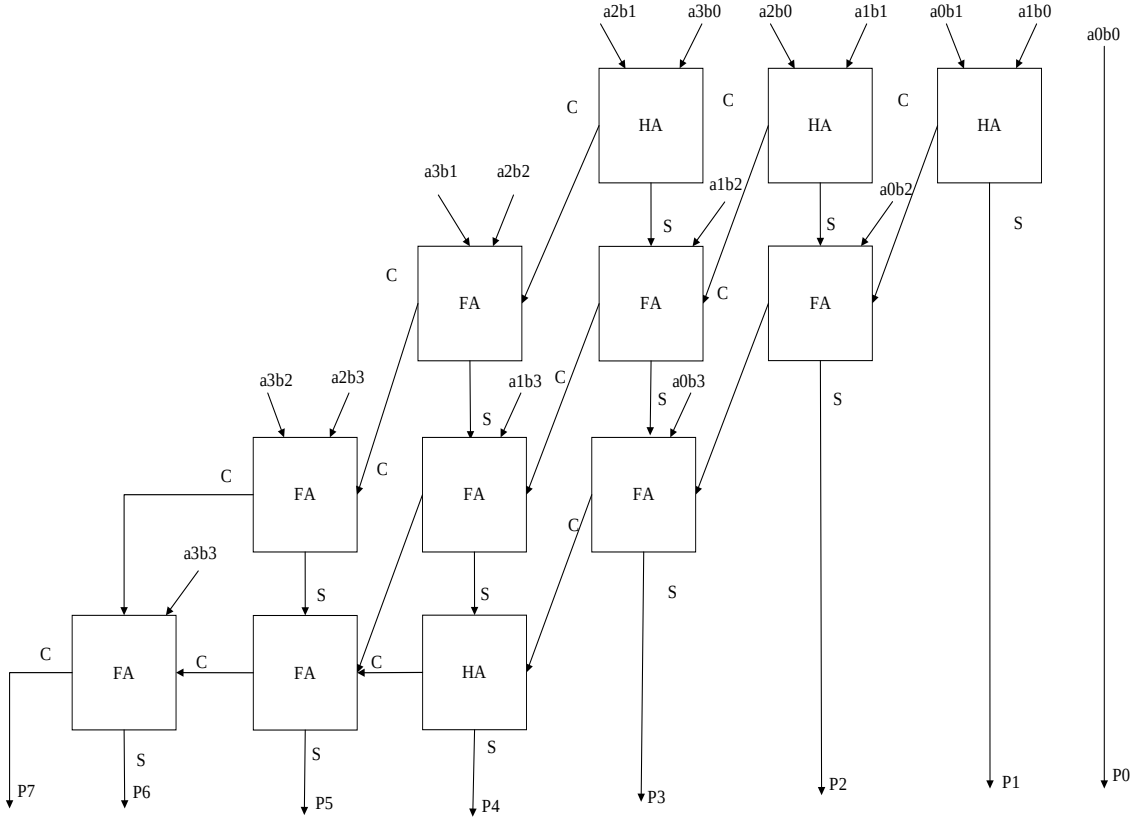
Σχήμα 3.54 Testbench πολλαπλασιαστή array 8x8



Σχήμα 3.55 Προσομοίωση πολλαπλασιαστή array 8x8

3.7 Περιγραφή πολλαπλασιαστών αρχιτεκτονικής carry save με VHDL και Verilog

Οι πολλαπλασιαστές carry save, δηλαδή διατήρησης κρατουμένου έχουν την ίδια δομή με τους array αλλά με μια βασική διαφορά. Οι carry save αντί για να δίνουν το κρατούμενο τους, το διαδίδουν παρακάτω στο κύκλωμα του πολλαπλασιαστή. Το γεγονός αυτό τους κάνει να είναι ταχύτεροι από τον τυπικό array. Χρησιμοποιείται ο ίδιος αριθμός αθροιστών όπως και στον array, δηλαδή τέσσερις ημιαθροιστές και οκτώ πλήρεις αθροιστές.



Σχήμα 3.56 Δομή πολλαπλασιαστή carry save 4x4

3.7.1 Περιγραφή carry save πολλαπλασιαστή 4x4 με την VHDL

Παρακάτω ακολουθεί η περιγραφή του Carry Save πολλαπλασιαστή 4x4 με τη χρήση της VHDL γλώσσας.

```
LIBRARY ieee;

LIBRARY work;

USE ieee.std_logic_1164.all;

USE work.full_add_package.all;
```

```
USE work.half_add_package.all;


ENTITY carrysafe4X4 IS
        PORT (                  a3, a2, a1, a0 : IN  STD_LOGIC;
                        b3, b2, b1, b0 : IN  STD_LOGIC;
          p7,p6,p5,p4,p3, p2, p1, p0 : OUT STD_LOGIC);
END carrysafe4X4;


ARCHITECTURE structured OF carrysafe4X4 IS
        SIGNAL a1b0,a0b1,c1 : STD_LOGIC;
        SIGNAL a2b0,a1b1,a0b2,c2,c20,hp2 : STD_LOGIC;
        SIGNAL a3b0,a2b1,a1b2,a0b3,c3,c30,c31,hp3,fp3 : STD_LOGIC;
        SIGNAL a3b1,a2b2,a1b3,c4,c40,c41,fp40,fp41 : STD_LOGIC;
        SIGNAL a3b2,a2b3,c50,c51,fp50 : STD_LOGIC;
        SIGNAL a3b3 : STD_LOGIC;
        SIGNAL cout: STD_LOGIC;


BEGIN
--P0:
  p0<=a0 AND b0;


--P1:
  a1b0<=a1 AND b0;
  a0b1<=a0 AND b1;
  HA1:half_add PORT MAP (a1b0, a0b1, c1, p1);
```

```
--P2:
  a2b0<=a2 AND b0;
  a1b1<=a1 AND b1;
  HA2: half_add PORT MAP (a2b0, a1b1, c2, hp2);
  a0b2<=a0 AND b2;
  FA2: full_add PORT MAP (hp2, a0b2, c1, c20, p2);

--P3:
  a3b0<=a3 AND b0;
  a2b1<=a2 AND b1;
  HA3: half_add PORT MAP (a3b0, a2b1, c3, hp3);
  a1b2<=a1 AND b2;
  FA30: full_add PORT MAP (hp3, a1b2, c2, c30, fp3);
  a0b3<=a0 AND b3;
  FA31: full_add PORT MAP (fp3, a0b3, c20, c31, p3);

--P4:
  a3b1<=a3 AND b1;
  a2b2<=a2 AND b2;
  FA40: full_add PORT MAP (a3b1, a2b2, c3, c40, fp40);
  a1b3<=a1 AND b3;
  FA41: full_add PORT MAP (fp40, a1b3, c30,c41, fp41);
  HA4:  half_add PORT MAP (fp41, c31, c4, p4);

--P5:
  a3b2<=a3 AND b2;
```

```
  a2b3<=a2 AND b3;
  FA50: full_add PORT MAP (a3b2, a2b3, c40, c50, fp50);
  FA51: full_add PORT MAP (fp50, c4, c41, c51, p5);

--P6:
  a3b3<=a3 AND b3;
  FA6: full_add PORT MAP (a3b3, c50, c51, cout, p6);

--P7:
  p7<=cout;

END structured;
```

Σχήμα 3.57 Περιγραφή του πολλαπλασιαστή carry save 4x4 με χρήση VHDL.

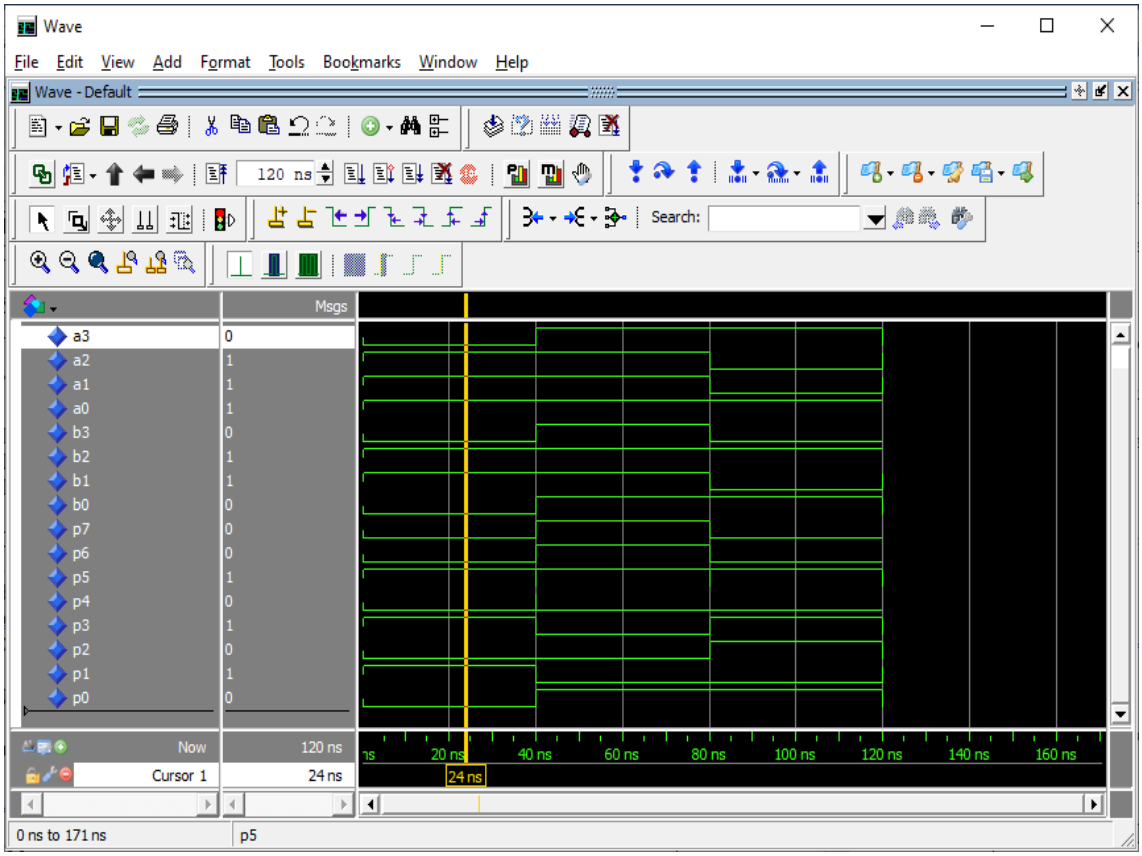
3.7.2 Προσομοίωση πολλαπλασιαστή carry save 4x4 με την VHDL

Στη συνέχεια ακολουθεί η προσομοίωση του παραπάνω αριθμητικού κυκλώματος χρησιμοποιώντας Testbench. Κάθε πράξη πραγματοποιείται σε χρονική απόσταση 40 ns. Επιπλέον, παρατηρείται το διάγραμμα Waveform με τα αποτελέσματα της προσομοίωσης

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY carrysafe4X4tb IS
END carrysafe4X4tb;

ARCHITECTURE behavior OF carrysafe4X4tb IS
```

```
SIGNAL a3, a2, a1, a0: STD_LOGIC;
SIGNAL b3, b2, b1, b0: STD_LOGIC;
SIGNAL p7,p6,p5,p4,p3, p2, p1, p0: STD_LOGIC;

COMPONENT carrysafe4X4
PORT (a3, a2, a1, a0: IN STD_LOGIC;
     b3, b2, b1, b0: IN STD_LOGIC;
     p7,p6,p5,p4,p3, p2, p1, p0: OUT STD_LOGIC);
END COMPONENT;

 BEGIN

m1: carrysafe4X4 PORT MAP(a3=>a3,a2=>a2,a1=>a1,a0=>a0,
                          b3=>b3,b2=>b2,b1=>b1,b0=>b0,
p7=>p7,p6=>p6,p5=>p5,p4=>p4,p3=>p3,p2=>p2,p1=>p1,p0=>p0);

PROCESS
BEGIN

 a3<='0';a2<='1';a1<='1';a0<='1';b3<='0';b2<='1';b1<='1';b0<='0'; WAIT FOR 40 ns;
 a3<='1';a2<='1';a1<='1';a0<='1';b3<='1';b2<='1';b1<='1';b0<='1'; WAIT FOR 40 ns;
 a3<='1';a2<='0';a1<='0';a0<='1';b3<='0';b2<='1';b1<='0';b0<='1'; WAIT FOR 40 ns;

 END PROCESS;
END behavior;
```

Σχήμα 3.58 Testbench πολλαπλασιαστή carry save 4x4 VHDL.



Σχήμα 3.59 Προσομοίωση πολλαπλασιαστή carry save 4x4 VHDL.

3.7.3 Περιγραφή carry save πολλαπλασιαστή 4x4 με την Verilog

Ακολουθεί η περιγραφή του πολλαπλασιαστή αρχιτεκτονικής array 8x8 με Verilog.

```
module HA(output sum, carry, input a, b);
    assign sum = a^b;
    assign carry = (a&b);
endmodule

module FA(output sum, carry, input a, b, cin);
    assign sum =(a^b^cin);
    assign carry = ((a&b)|(a&cin)|(b&cin));
```

```
  endmodule


  module carry_save_4x4(output [7:0] product, input [3:0] in1,in2,input clock);
        wire [10:0] s, c;
  assign product[0]= (in1[0]&in2[0]);
//row 1
HA HA1(product[1],c[0],(in1[1]&in2[0]),(in1[0]&in2[1])); //p1
//row 2
HA HA2(s[1],c[1],(in1[2]&in2[0]),(in1[1]&in2[1]));
FA FA1(product[2],c[2],c[0],(in1[0]&in2[2]),s[1]);//p2
//row 3
HA HA3(s[3],c[3],(in1[3]&in2[0]),(in1[2]&in2[1]));
FA FA2(s[4],c[4],s[3],(in1[1]&in2[2]),c[1]);
FA FA3(product[3],c[5],c[2],(in1[0]&in2[3]),s[4]);//p3
// row 4
FA FA4(s[6],c[6],(in1[3]&in2[1]),(in1[2]&in2[2]),c[3]);
FA FA5(s[7],c[7],s[6],(in1[1]&in2[3]),c[4]);
HA HA4(product[4],c[8],c[5],s[7]);//p4
//row 5
FA FA6(s[9],c[9],(in1[3]&in2[2]),(in1[2]&in2[3]),c[6]);
FA FA7(product[5],c[10],c[8],c[7],s[9]);//p5
//row 6
FA FA8(product[6],product[7],(in1[3]&in2[3]),c[10],c[9]);//p6,7
endmodule
```

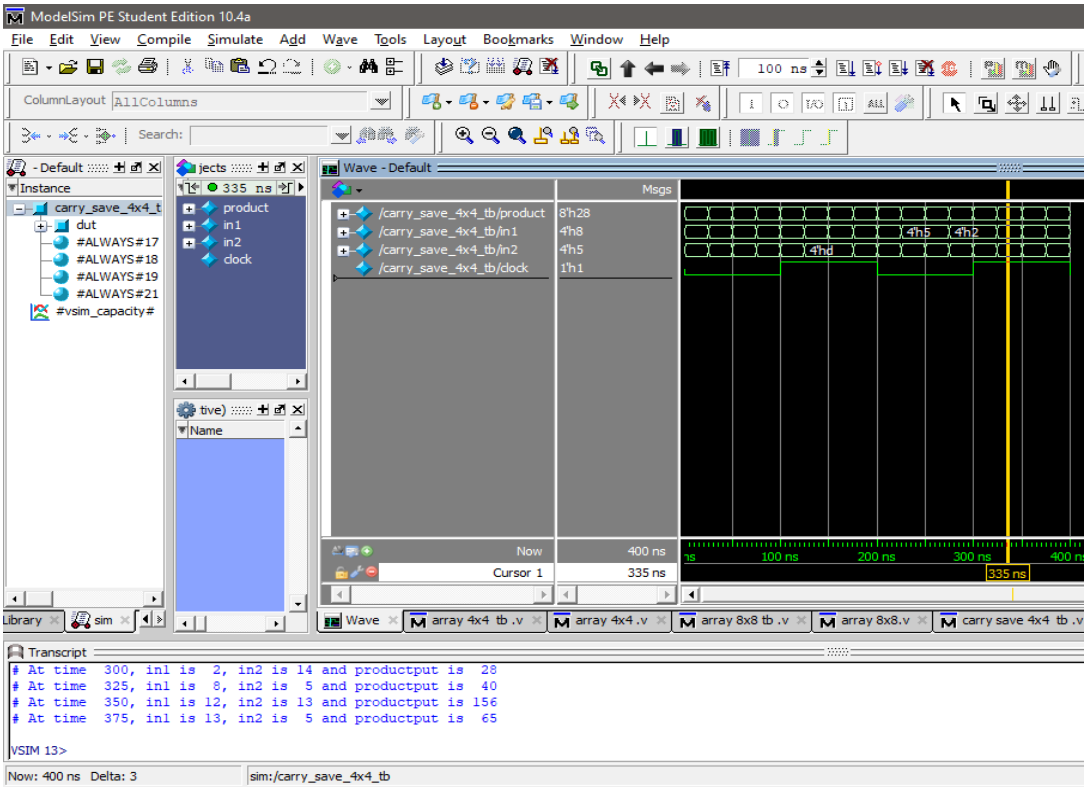
Σχήμα 3.60 Περιγραφή του πολλαπλασιαστή carry save 4x4 με χρήση Verilog

3.7.4 Προσομοίωση πολλαπλασιαστή carry save 4x4 με την Verilog

Ο πολλαπλασιαστής carry save δεν αλλάζει δομικά σε σχέση με τον array αλλά αλλάζει ελαφρώς η συνδεσμολογία. Οι είσοδοι και έξοδος παραμένουν ίδιες, δηλαδή ο πολλαπλασιαστής, ο πολλαπλασιαστέος και το γινόμενο. Όπως και στην αρχιτεκτονική array 4x4 οι είσοδοι είναι τέσσαρα bit και η έξοδος οκτώ. Η αρχικοποίηση γίνεται στο μηδέν και τυχαίες αλλαγές γίνονται στις εισόδους ανά εικοσιπέντε νανοδευτερόλεπτα. Η δομή του φαίνεται στο σχήμα 3.56. Ακολουθούν το testbench και η προσομοίωση.

```
module carry_save_4x4_tb(); //setting inputs and output
    wire [7:0] product;
    reg [3:0] in1;
    reg [3:0] in2;
    reg clock;

carry_save_4x4 dut (.in1(in1), .in2(in2), .product(product), .clock(clock));
    initial begin
        in1 <= 0;
        in2 <= 0;
        clock <= 0;
    end
    always #100 clock <= ~clock;
    always #25 in1 <= $random; // every 25ns value of input is ran1ndomised
    always #25 in2 <= $random;
    always @(in1 or in2) // if any input changes the message appears
$monitor ("At time %4d, in1 is %d, in2 is %d and product is %d", $time, in1, in2, product); // message for extra info
endmodule
```

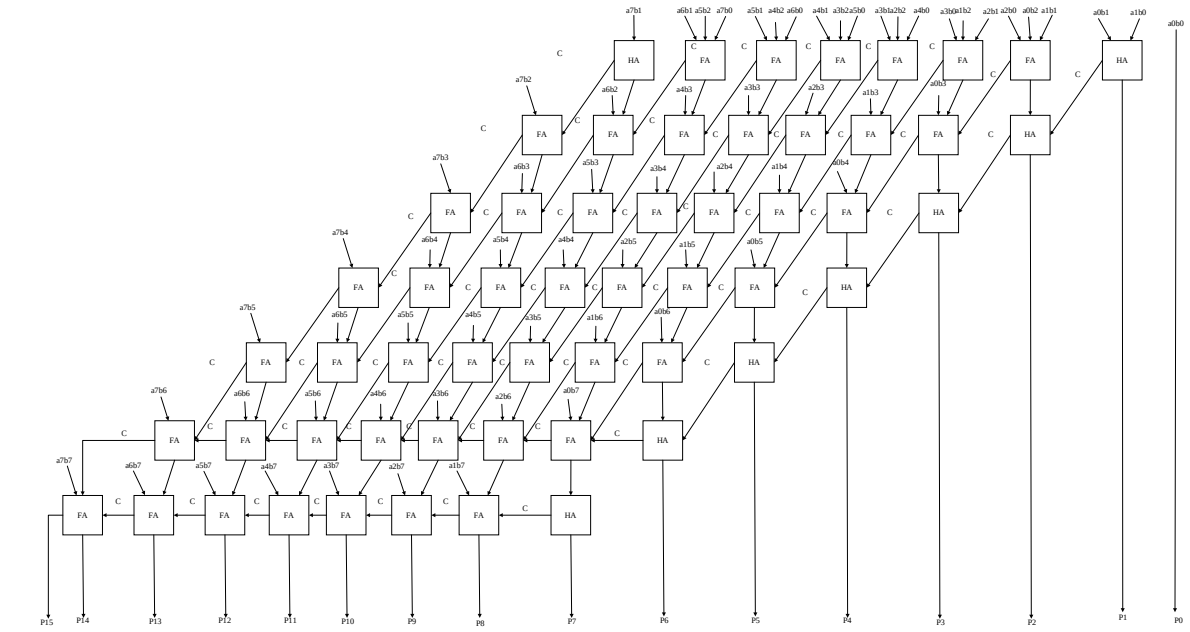
Σχήμα 3.61 Testbench πολλαπλασιαστή carry save 4x4



Σχήμα 3.62 Προσομοίωση πολλαπλασιαστή carry save 4x4

3.8 Περιγραφή carry save πολλαπλασιαστή 8x8 με VHDL και Verilog

Ο πολλαπλασιαστής αρχιτεκτονικής carry save 8x8 εφαρμόζει την ίδια αρχιτεκτονική με τον 4x4 αλλά για πολύ μεγαλύτερο πλήθος τιμών. Χρησιμοποιούνται οκτώ ημιαθροιστές και σαράντα οκτώ πλήρεις αθροιστές. Ακολουθούν το κύκλωμα και η υλοποίηση σε Verilog.



Σχήμα 3.63 Δομή πολλαπλασιαστή carry save 8x8

3.8.1 Περιγραφή carry save πολλαπλασιαστή 8x8 με VHDL

Παρακάτω αποτυπώνεται η περιγραφή Carry Save πολλαπλασιαστή 8x8 σε VHDL κώδικα.

```
LIBRARY ieee;

LIBRARY work;

USE ieee.std_logic_1164.all;

USE work.full_add_package.all;

USE work.half_add_package.all;



ENTITY carrysave8X8 IS

        PORT (                                          a7, a6, a5, a4, a3, a2, a1, a0 :
IN  STD_LOGIC;

                                        b7, b6, b5, b4, b3, b2, b1, b0 : IN
STD_LOGIC;

          p15, p14, p13, p12, p11, p10, p9, p8, p7, p6, p5, p4, p3, p2, p1, p0 : OUT
STD_LOGIC);

END carrysave8X8;



ARCHITECTURE structured OF carrysave8X8 IS

        SIGNAL a1b0,a0b1,c1 : STD_LOGIC;

      SIGNAL a2b0,a1b1,a0b2,c2,c20,fp2 : STD_LOGIC;

      SIGNAL a3b0,a2b1,a1b2,a0b3,c3,c30,c31,fp31,fp30 : STD_LOGIC;

      SIGNAL a4b0,a3b1,a2b2,a1b3,a0b4,c4,c40,c41,c42,fp40,fp41,fp42 :
STD_LOGIC;

        SIGNAL
a5b0,a4b1,a3b2,a2b3,a1b4,a0b5,c5,c50,c51,c52,c53,fp50,fp51,fp52,fp53 :
STD_LOGIC;

        SIGNAL
a6b0,a5b1,a4b2,a3b3,a2b4,a1b5,a0b6,c6,c60,c61,c62,c63,c64,fp60,fp61,fp62,fp63,fp
64 : STD_LOGIC;

        SIGNAL
```

```
a7b0,a6b1,a5b2,a4b3,a3b4,a2b5,a1b6,a0b7,c7,c70,c71,c72,c73,c74,c75,fp70,fp71,fp7
2,fp73,fp74,fp75 : STD_LOGIC;

        SIGNAL
a7b1,a6b2,a5b3,a4b4,a3b5,a2b6,a1b7,c8,c80,c81,c82,c83,c84,c85,fp80,fp81,fp82,fp8
3,fp84,hp8 : STD_LOGIC;

        SIGNAL
a7b2,a6b3,a5b4,a4b5,a3b6,a2b7,c90,c91,c92,c93,c94,c95,fp90,fp91,fp92,fp93,fp94 :
STD_LOGIC;

        SIGNAL a7b3,a6b4,a5b5,a4b6,a3b7,
c100,c101,c102,c103,c104,fp100,fp101,fp102,fp103 : STD_LOGIC;

        SIGNAL a7b4,a6b5,a5b6,a4b7, c110,c111,c112,c113,fp110,fp111,fp112 :
STD_LOGIC;

        SIGNAL a7b5,a6b6,a5b7,c120,c121,c122,fp120,fp121 : STD_LOGIC;

        SIGNAL a7b6,a6b7,c130,c131,fp130 : STD_LOGIC;

        SIGNAL a7b7,c140 : STD_LOGIC;



BEGIN

--P0:

  p0<=a0 AND b0;


--P1:

  a1b0<=a1 AND b0;

  a0b1<=a0 AND b1;

  HA1:half_add PORT MAP (a1b0, a0b1, c1, p1);


--P2:

  a2b0<=a2 AND b0;

  a1b1<=a1 AND b1;
```

```
  a0b2<=a0 AND b2;
  FA2: full_add PORT MAP (a2b0, a1b1, a0b2, c20, fp2);
  HA2: half_add PORT MAP (c1, fp2, c2, p2);

--P3:
  a3b0<=a3 AND b0;
  a2b1<=a2 AND b1;
  a1b2<=a1 AND b2;
  FA30: full_add PORT MAP (a3b0, a2b1, a1b2, c30, fp30);
  a0b3<=a0 AND b3;
  FA31: full_add PORT MAP (fp30, a0b3, c20, c31, fp31);
  HA3: half_add PORT MAP (c2, fp31, c3, p3);

--P4:
  a4b0<=a4 AND b0;
  a3b1<=a3 AND b1;
  a2b2<=a2 AND b2;
  FA40: full_add PORT MAP (a4b0, a3b1, a2b2, c40, fp40);
  a1b3<=a1 AND b3;
  FA41: full_add PORT MAP (fp40, a1b3, c30,c41, fp41);
  a0b4<=a0 AND b4;
  FA42: full_add PORT MAP (fp41, a0b4, c31,c42, fp42);
  HA4:  half_add PORT MAP (fp42, c3, c4, p4);

--P5:
  a5b0<=a5 AND b0;
```

```
  a4b1<=a4 AND b1;
  a3b2<=a3 AND b2;
  FA50: full_add PORT MAP (a5b0, a4b1, a3b2, c50, fp50);
  a2b3<=a2 AND b3;
  FA51: full_add PORT MAP (fp50, a2b3, c40, c51, fp51);
  a1b4<=a1 AND b4;
  FA52: full_add PORT MAP (fp51, a1b4, c41, c52, fp52);
  a0b5<=a0 AND b5;
  FA53: full_add PORT MAP (fp52, a0b5, c42, c53, fp53);
  HA5:  half_add PORT MAP (fp53, c4, c5, p5);

--P6:
  a6b0<=a6 AND b0;
  a5b1<=a5 AND b1;
  a4b2<=a4 AND b2;
  FA60: full_add PORT MAP (a6b0, a5b1, a4b2, c60, fp60);
  a3b3<=a3 AND b3;
  FA61: full_add PORT MAP (fp60, a3b3, c50, c61, fp61);
  a2b4<=a2 AND b4;
  FA62: full_add PORT MAP (fp61, a2b4, c51, c62, fp62);
  a1b5<=a1 AND b5;
  FA63: full_add PORT MAP (fp62, a1b5, c52, c63, fp63);
  a0b6<=a0 AND b6;
  FA64: full_add PORT MAP (fp63, a0b6, c53, c64, fp64);
  HA6:  half_add PORT MAP (fp64, c5, c6, p6);
```

```
--P7:
  a7b0<=a7 AND b0;
  a6b1<=a6 AND b1;
  a5b2<=a5 AND b2;
  FA70: full_add PORT MAP (a7b0, a6b1, a5b2, c70, fp70);
  a4b3<=a4 AND b3;
  FA71: full_add PORT MAP (fp70, a4b3, c60, c71, fp71);
  a3b4<=a3 AND b4;
  FA72: full_add PORT MAP (fp71, a3b4, c61, c72, fp72);
  a2b5<=a2 AND b5;
  FA73: full_add PORT MAP (fp72, a2b5, c62, c73, fp73);
  a1b6<=a1 AND b6;
  FA74: full_add PORT MAP (fp73, a1b6, c63, c74, fp74);
  a0b7<=a0 AND b7;
  FA75: full_add PORT MAP (fp74, a0b7, c64, c75, fp75);
  HA7:  half_add PORT MAP (fp75, c6, c7, p7);

--P8:
  a7b1<=a7 AND b1;
  a6b2<=a6 AND b2;
  H8: half_add PORT MAP (a7b1, a6b2, c8, hp8);
  a5b3<=a5 AND b3;
  FA85: full_add PORT MAP (a5b3, hp8, c70, c80, fp80);
  a4b4<=a4 AND b4;
  FA80: full_add PORT MAP (fp80, a4b4, c71, c81, fp81);
  a3b5<=a3 AND b5;
```

```
  FA81: full_add PORT MAP (fp81, a3b5, c72, c82, fp82);
  a2b6<=a2 AND b6;
  FA82: full_add PORT MAP (fp82, a2b6, c73, c83, fp83);
  a1b7<=a1 AND b7;
  FA83: full_add PORT MAP (fp83, a1b7, c74, c84, fp84);
  FA84: full_add PORT MAP (fp84, c75, c7, c85, p8);



--P9:
  a7b2<=a7 AND b2;
  a6b3<=a6 AND b3;
  FA90: full_add PORT MAP (a6b3, a7b2, c8, c90, fp90);
  a5b4<=a5 AND b4;
  FA91: full_add PORT MAP (fp90, a5b4, c80, c91, fp91);
  a4b5<=a4 AND b5;
  FA92: full_add PORT MAP (fp91, a4b5, c81, c92, fp92);
  a3b6<=a3 AND b6;
  FA93: full_add PORT MAP (fp92, a3b6, c82, c93, fp93);
  a2b7<=a2 AND b7;
  FA94: full_add PORT MAP (fp93, a2b7, c83, c94, fp94);
  FA95: full_add PORT MAP (fp94, c84, c85, c95, p9);


--P10:
  a7b3<=a7 AND b3;
  a6b4<=a6 AND b4;
  FA100: full_add PORT MAP (a6b4, a7b3, c90, c100, fp100);
```

```
  a5b5<=a5 AND b5;
  FA7101: full_add PORT MAP (fp100, a5b5, c91, c101, fp101);
  a4b6<=a4 AND b6;
  FA102: full_add PORT MAP (fp101, a4b6, c92, c102, fp102);
  a3b7<=a3 AND b7;
  FA103: full_add PORT MAP (fp102, a3b7, c93, c103, fp103);
  FA104: full_add PORT MAP (fp103, c94, c95, c104, p10);

--P11:
  a7b4<=a7 AND b4;
  a6b5<=a6 AND b5;
  FA110: full_add PORT MAP (a6b5, a7b4, c100, c110, fp110);
  a5b6<=a5 AND b6;
  FA111: full_add PORT MAP (fp110, a5b6, c101, c111, fp111);
  a4b7<=a4 AND b7;
  FA112: full_add PORT MAP (fp111, a4b7, c102, c112, fp112);
  FA113: full_add PORT MAP (c104, fp112, c103, c113, p11);

--P12:
  a7b5<=a7 AND b5;
  a6b6<=a6 AND b6;
  FA120: full_add PORT MAP (a6b6, a7b5, c110, c120, fp120);
  a5b7<=a5 AND b7;
  FA121: full_add PORT MAP (fp120, a5b7, c111, c121, fp121);
  FA122: full_add PORT MAP (fp121, c112, c113, c122, p12);
```

```
--P13:
  a7b6<=a7 AND b6;
  a6b7<=a6 AND b7;
  FA130: full_add PORT MAP (a6b7, a7b6, c120, c130, fp130);
  FA131: full_add PORT MAP (fp130, c121, c122, c131, p13);

--P14:
  a7b7<=a7 AND b7;
  FA140: full_add PORT MAP (c130, a7b7, c131, c140, p14);

--p15
  p15<=c140;
END structured;
```

Σχήμα 3.64 Περιγραφή του πολλαπλασιαστή carry save 8x8 με χρήση VHDL.

3.8.2 Προσομοίωση πολλαπλασιαστή carry save 8x8 με την VHDL

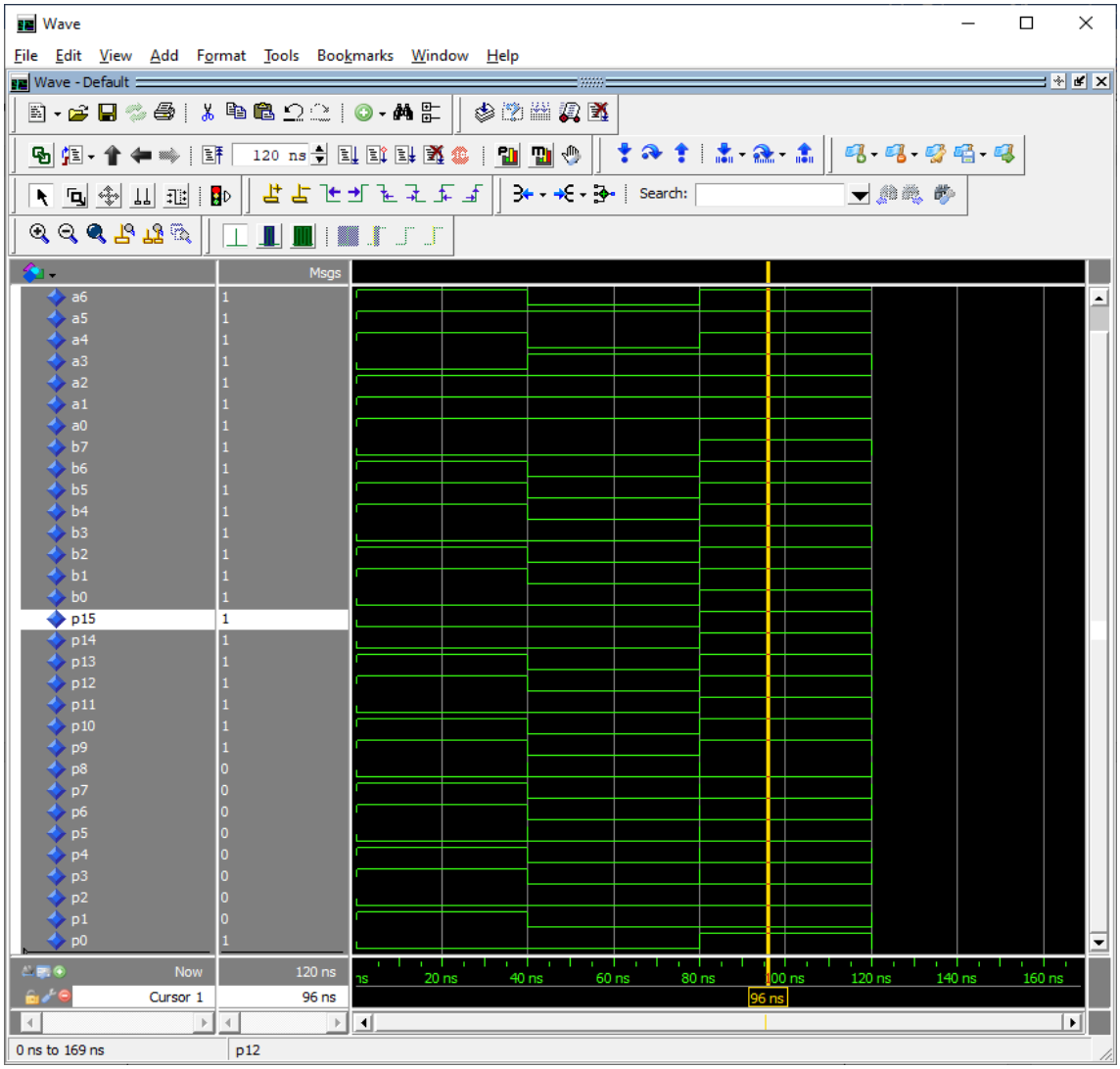
Παρατίθεται στη συνέχεια ο κώδικας της προσομοίωσης του Carry Save 8x8 με κάθε πράξη να εκτελείται ανά 40ns. Τέλος, τα αποτελέσματα της προσομοίωσης παρατηρούνται στο Waveform διάγραμμα.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY carrysave8X8tb IS
END carrysave8X8tb;

ARCHITECTURE behavior OF carrysave8X8tb IS

SIGNAL  a7, a6, a5, a4, a3, a2, a1, a0: STD_LOGIC;
```

```
SIGNAL  b7, b6, b5, b4, b3, b2, b1, b0: STD_LOGIC;

SIGNAL  p15, p14, p13, p12, p11, p10, p9, p8, p7, p6, p5, p4, p3, p2, p1, p0:
STD_LOGIC;



COMPONENT carrysave8X8

PORT (a7, a6, a5, a4, a3, a2, a1, a0: IN STD_LOGIC;

     b7, b6, b5, b4, b3, b2, b1, b0: IN STD_LOGIC;

     p15, p14, p13, p12, p11, p10, p9, p8, p7, p6, p5, p4, p3, p2, p1, p0: OUT
STD_LOGIC);

END COMPONENT;



 BEGIN



m1: carrysave8X8 PORT
MAP(a7=>a7,a6=>a6,a5=>a5,a4=>a4,a3=>a3,a2=>a2,a1=>a1,a0=>a0,

                    b7=>b7,b6=>b6,b5=>b5,b4=>b4,b3=>b3,b2=>b2,b1=>b1,b0=>b0,

p15=>p15,p14=>p14,p13=>p13,p12=>p12,p11=>p11,p10=>p10,p9=>p9,p8=>p8,p7=
>p7,p6=>p6,p5=>p5,p4=>p4,p3=>p3,p2=>p2,p1=>p1,p0=>p0);



PROCESS

BEGIN




a7<='0';a6<='1';a5<='1';a4<='1';a3<='0';a2<='1';a1<='1';a0<='1';b7<='0';b6<='1';b5<='
1';b4<='1';b3<='0';b2<='1';b1<='1';b0<='0'; WAIT FOR 40 ns;


a7<='0';a6<='0';a5<='1';a4<='0';a3<='1';a2<='1';a1<='1';a0<='1';b7<='0';b6<='0';b5<='
0';b4<='0';b3<='0';b2<='0';b1<='0';b0<='0'; WAIT FOR 40 ns;


a7<='1';a6<='1';a5<='1';a4<='1';a3<='1';a2<='1';a1<='1';a0<='1';b7<='1';b6<='1';b5<='
```

```
1';b4<='1';b3<='1';b2<='1';b1<='1';b0<='1'; WAIT FOR 40 ns;


 END PROCESS;

END behavior;
```

Σχήμα 3.65 Περιγραφή του πολλαπλασιαστή carry save 8x8 με χρήση VHDL.



Σχήμα 3.66 Προσομοίωση πολλαπλασιαστή carry save 8x8 VHDL.

3.8.3 Περιγραφή carry save πολλαπλασιαστή 8x8 με Verilog

Ακολουθούν το κύκλωμα και η υλοποίηση σε Verilog.

```
module HA(output sum, carry, input a ,b);
```

```
        assign sum = a^b;
        assign carry = (a&b);
endmodule
module FA(output sum, carry, input a, b, cin);
        assign sum =(a^b^cin);
        assign carry = ((a&b)|(a&cin)|(b&cin));
endmodule
module carry_save_8x8(output [15:0] product, input [7:0] in1,in2,input clock);
        wire [55:0] s, c;
HA HA1(s[0],c[0],(in1[1]&in2[0]),(in1[0]&in2[1])); //p1
FA FA1(s[1],c[1],(in1[2]&in2[0]),(in1[0]&in2[2]),(in1[1]&in2[1]));
HA HA2(s[2],c[2],c[0],s[1]);//p2
FA FA2(s[3],c[3],(in1[3]&in2[0]),(in1[1]&in2[2]),(in1[2]&in2[1]));
FA FA3(s[4],c[4],s[3],c[1],(in1[0]&in2[3]));
HA HA3(s[5],c[5],s[4],c[2]); //p3
FA FA4(s[6],c[6],(in1[4]&in2[0]),(in1[3]&in2[1]),(in1[2]&in2[2]));
FA FA5(s[7],c[7],s[6],c[3],(in1[1]&in2[3]));
FA FA6(s[8],c[8],s[7],c[4],(in1[0]&in2[4]));
HA HA4(s[9],c[9],s[8],c[5]);//p4
FA FA7(s[10],c[10],(in1[3]&in2[2]),(in1[4]&in2[1]),(in1[5]&in2[0]));
FA FA8(s[11],c[11],s[10],c[6],(in1[2]&in2[3]));
FA FA9(s[12],c[12],s[11],c[7],(in1[1]&in2[4]));
FA FA10(s[13],c[13],s[12],c[8],(in1[0]&in2[5]));
HA HA5(s[14],c[14],s[13],c[9]);//p5
FA FA11(s[15],c[15],(in1[6]&in2[0]),(in1[5]&in2[1]),(in1[4]&in2[2]));
FA FA12(s[16],c[16],s[15],c[10],(in1[3]&in2[3]));
```

```
FA FA13(s[17],c[17],s[16],c[11],(in1[2]&in2[4]));

FA FA14(s[18],c[18],s[17],c[12],(in1[1]&in2[5]));

FA FA15(s[19],c[19],s[18],c[13],(in1[0]&in2[6]));

HA HA6(s[20],c[20],s[19],c[14]);//p6

FA FA16(s[21],c[21],(in1[7]&in2[0]),(in1[6]&in2[1]),(in1[5]&in2[2]));

FA FA17(s[22],c[22],s[21],c[15],(in1[4]&in2[3]));

FA FA18(s[23],c[23],s[22],c[16],(in1[3]&in2[4]));

FA FA19(s[24],c[24],s[23],c[17],(in1[2]&in2[5]));

FA FA20(s[25],c[25],s[24],c[18],(in1[1]&in2[6]));

FA FA21(s[26],c[26],s[25],c[19],(in1[0]&in2[7]));

HA HA7(s[27],c[27],s[26],c[20]);//p7

HA HA8(s[28],c[28],(in1[7]&in2[1]),(in1[6]&in2[2]));

FA FA22(s[29],c[29],s[28],c[21],(in1[5]&in2[3]));

FA FA23(s[30],c[30],s[29],c[22],(in1[4]&in2[4]));

FA FA24(s[31],c[31],s[30],c[23],(in1[3]&in2[5]));

FA FA25(s[32],c[32],s[31],c[24],(in1[2]&in2[6]));

FA FA26(s[33],c[33],s[32],c[25],(in1[1]&in2[7]));

FA FA27(s[34],c[34],s[33],c[26],c[27]);//p8

FA FA28(s[35],c[35],c[28],(in1[6]&in2[3]),(in1[7]&in2[2]));

FA FA29(s[36],c[36],s[35],c[29],(in1[5]&in2[4]));

FA FA30(s[37],c[37],s[36],c[30],(in1[4]&in2[5]));

FA FA31(s[38],c[38],s[37],c[31],(in1[3]&in2[6]));

FA FA32(s[39],c[39],s[38],c[32],(in1[2]&in2[7]));

FA FA33(s[40],c[40],s[39],c[33],c[34]);//p9

FA FA34(s[41],c[41],c[35],(in1[6]&in2[4]),(in1[7]&in2[3]));

FA FA35(s[42],c[42],s[41],c[36],(in1[5]&in2[5]));
```

```
FA FA36(s[43],c[43],s[42],c[37],(in1[4]&in2[6]));
FA FA37(s[44],c[44],s[43],c[38],(in1[3]&in2[7]));
FA FA38(s[45],c[45],s[44],c[39],c[40]);//p10
FA FA39(s[46],c[46],c[41],(in1[6]&in2[5]),(in1[7]&in2[4]));
FA FA40(s[47],c[47],s[46],c[42],(in1[5]&in2[6]));
FA FA41(s[48],c[48],s[47],c[43],(in1[4]&in2[7]));
FA FA42(s[49],c[49],s[48],c[45],c[44]);//p11 La8os
FA FA43(s[50],c[50],c[46],(in1[6]&in2[6]),(in1[7]&in2[5]));
FA FA44(s[51],c[51],s[50],c[47],(in1[5]&in2[7]));
FA FA45(s[52],c[52],s[51],c[48],c[49]);//p12
FA FA46(s[53],c[53],c[50],(in1[6]&in2[7]),(in1[7]&in2[6]));
FA FA47(s[54],c[54],s[53],c[51],c[52]);//p13
FA FA48(s[55],c[55],c[54],c[53],(in1[7]&in2[7]));//p14,15
assign product[0]= (in1[0]&in2[0]); //p0
assign product[1] = s[0];
assign product[2] = s[2];
assign product[3] = s[5];
assign product[4] = s[9];
assign product[5] = s[14];
assign product[6] = s[20];
assign product[7] = s[27];
assign product[8] = s[34];
assign product[9] = s[40];
assign product[10] = s[45];
assign product[11] = s[49];
assign product[12] = s[52];
```

```
assign product[13] = s[54];

assign product[14] = s[55];

assign product[15] = c[55];

  endmodule
```

Σχήμα 3.67 Περιγραφή του πολλαπλασιαστή carry save 8x8 με χρήση Verilog

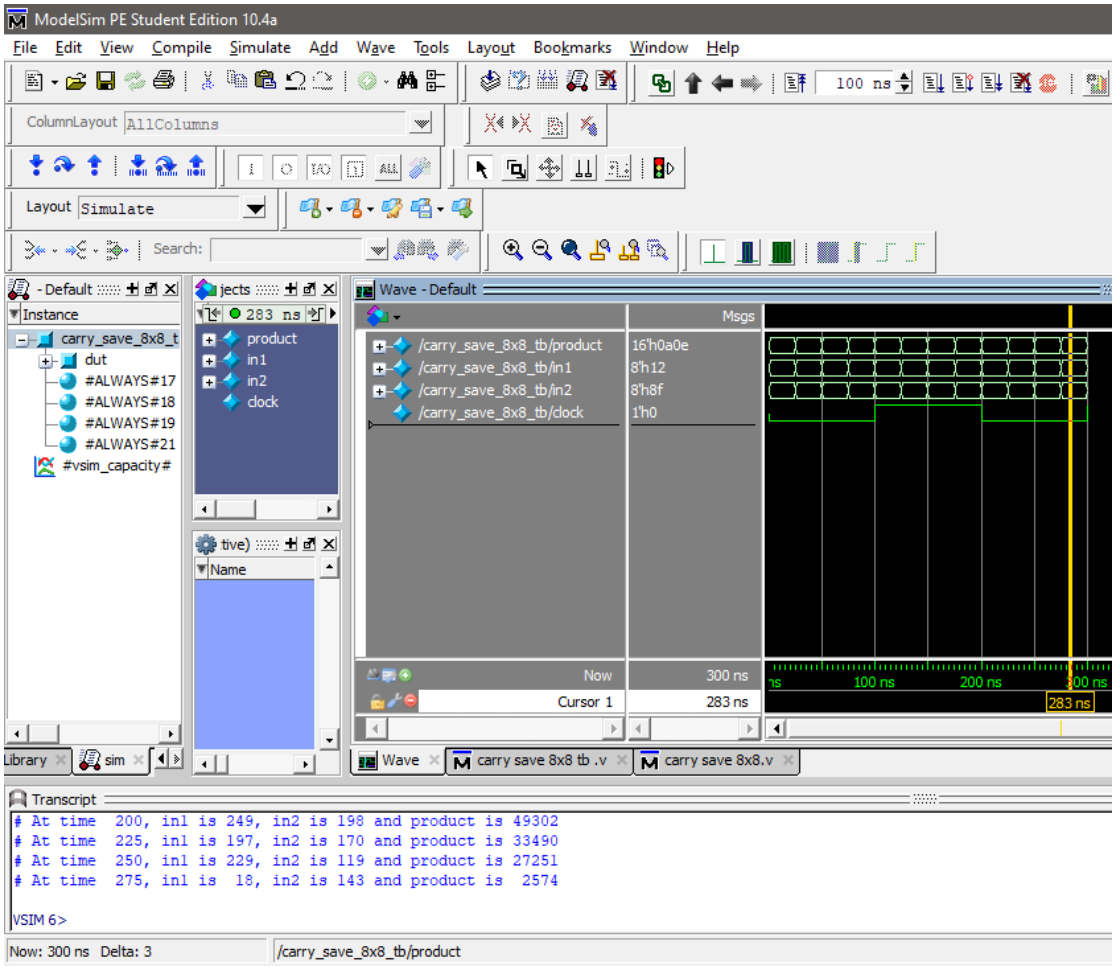
3.8.4 Προσομοίωση πολλαπλασιαστή carry save 8x8 με την Verilog

Έχει την ίδια ακριβώς λειτουργικότητα με τον πολλαπλασιαστή carry save 4x4 αλλά είναι δομημένος για εισόδους των οκτώ bit. Η αρχικοποίηση παραμένει στο μηδέν για κάθε είσοδο και οι αλλαγές στις εισόδους πραγματοποιούνται κάθε εικοσιπέντε νανοδευτερόλεπτα. Η δομή του φαίνεται στο σχήμα 3.63. Ακολουθούν το testbench και η προσομοίωση.

```
 module carry_save_8x8_tb(); //setting inputs and output

        wire [15:0] product;

        reg [7:0] in1;

        reg [7:0] in2;

        reg clock;

 carry_save_8x8 dut (.in1(in1), .in2(in2), .product(product), .clock(clock));

        initial begin

                in1 <= 0;

                in2 <= 0;

                clock <= 0;

        end

        always #100 clock <= ~clock;

        always #25 in1 <= $random; // every 25ns value of input is ran1ndomised

        always #25 in2 <= $random;

always @(in1 or in2) // if any input changes the message appears

$monitor ("At time %4d, in1 is %d, in2 is %d and product is %d", $time, in1, in2, product); // message for extra info
```

```
endmodule
```

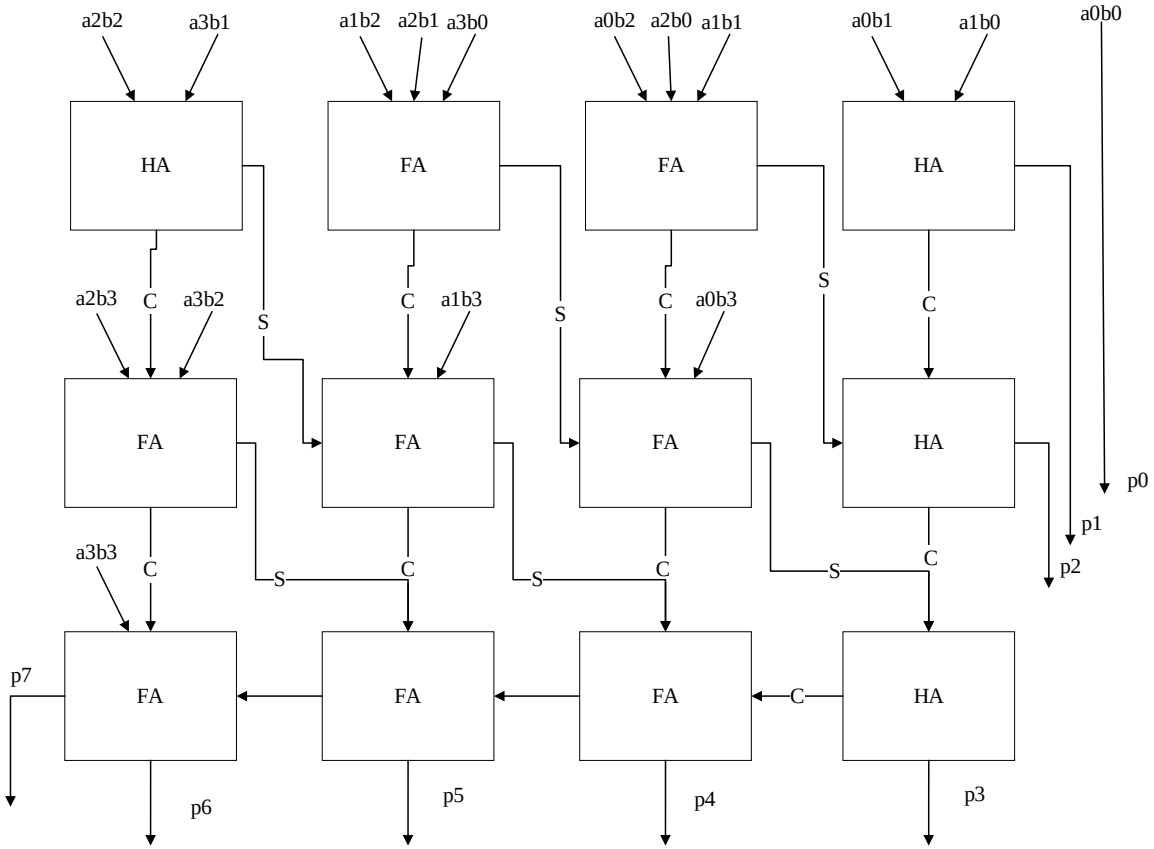
Σχήμα 3.68 Testbench πολλαπλασιαστή carry save 8x8



Σχήμα 3.69 Προσομοίωση πολλαπλασιαστή carry save 8x8

3.9 Περιγραφή πολλαπλασιαστών αρχιτεκτονικής Wallace tree με VHDL και Verilog

Οι πολλαπλασιαστές αρχιτεκτονικής Wallace tree είναι αρκετά ταχύτεροι και απαιτούν λιγότερη ενέργεια από τους array όμως απαιτούν περισσότερο χώρο και έχουν πιο περίπλοκη συνδεσμολογία. Ο Wallace tree λειτουργεί λίγο διαφορετικά από τον array. Έχει τρία στάδια. Στο πρώτο γίνεται εξαγωγή των μερικών αποτελεσμάτων. Στο δεύτερο γίνεται η προσθήκη των παραπάνω μερικών αποτελεσμάτων. Στο τρίτο και τελευταίο, αν έχει γίνει σωστά η διαδικασία γίνεται απλώς μια πρόσθεση μεταξύ όλων των τιμών που έχουν μείνει. Χρησιμοποιούνται τέσσερις ημιαθροιστές και οκτώ πλήρεις αθροιστές.



Σχήμα 3.70 Δομή πολλαπλασιαστή Wallace Tree 4x4

3.9.1 Περιγραφή Wallace tree πολλαπλασιαστή 4x4 με την VHDL

Παρακάτω περιγράφεται ο κώδικας του πολλαπλασιαστή Wallace Tree 4x4 σε VHDL.

```
LIBRARY ieee;

LIBRARY work;

USE ieee.std_logic_1164.all;

USE work.full_add_package.all;

USE work.half_add_package.all;


ENTITY Wallace4 IS

        PORT (                  a3, a2, a1, a0 : IN  STD_LOGIC;

                                b3, b2, b1, b0 : IN  STD_LOGIC;

          p7,p6,p5,p4,p3, p2, p1, p0 : OUT STD_LOGIC);
```

```
END Wallace4;


ARCHITECTURE structured OF Wallace4 IS
        SIGNAL a0b0,a1b0,a2b0,a3b0: STD_LOGIC;
        SIGNAL a0b1,a1b1,a2b1,a3b1: STD_LOGIC;
        SIGNAL a0b2,a1b2,a2b2,a3b2: STD_LOGIC;
        SIGNAL a0b3,a1b3,a2b3,a3b3: STD_LOGIC;
        SIGNAL c01,c02,c03,c04: STD_LOGIC;
        SIGNAL ap01,ap02,ap03,ap04: STD_LOGIC;
        SIGNAL c11,c12,c13,c14: STD_LOGIC;
        SIGNAL ap11,ap12,ap13,ap14: STD_LOGIC;
        SIGNAL c21,c22,c23,c24,c25: STD_LOGIC;
        SIGNAL ap21,ap22,ap23,ap24,ap25: STD_LOGIC;


BEGIN
--Stage 1


a0b0<= a0 and b0;
a1b0<= a1 and b0;
a2b0<= a2 and b0;
a3b0<= a3 and b0;


a0b1<= a0 and b1;
a1b1<= a1 and b1;
a2b1<= a2 and b1;
a3b1<= a3 and b1;
```

```
a0b2<= a0 and b2;
a1b2<= a1 and b2;
a2b2<= a2 and b2;
a3b2<= a3 and b2;

a0b3<= a0 and b3;
a1b3<= a1 and b3;
a2b3<= a2 and b3;
a3b3<= a3 and b3;

--Stage 2
--Part 1--
HA1:half_add PORT MAP(a1b0,a0b1,c01,ap01);
FA1:full_add PORT MAP(a2b0,a1b1,a0b2,c02,ap02);
FA2:full_add PORT MAP(a3b0,a2b1,a1b2,c03,ap03);
HA2:half_add PORT MAP(a3b1,a2b2,c04,ap04);

--part 2--
HA3:half_add PORT MAP(ap02,c01,c11,ap11);
FA3:full_add PORT MAP(ap03,c02,a0b3,c12,ap12);
FA4:full_add PORT MAP(ap04,c03,a1b3,c13,ap13);
FA5:full_add PORT MAP(a3b2,c04,a2b3,c14,ap14);

--part 3--
HA4:half_add PORT MAP(ap12,c11,c21,ap21);
```

```
FA6:full_add PORT MAP(ap13,c12,c21,c22,ap22);

FA7:full_add PORT MAP(ap14,c13,c22,c23,ap23);

FA8:full_add PORT MAP(a3b3,c14,c23,c24,ap24);


--Final--


p7<=c24;

p6<=ap24;

p5<=ap23;

p4<=ap22;

p3<=ap21;

p2<=ap11;

p1<=ap01;

p0<=a0b0;

END structured;
```

Σχήμα 3.71 Περιγραφή του πολλαπλασιαστή Wallace tree 8bit με χρήση VHDL.

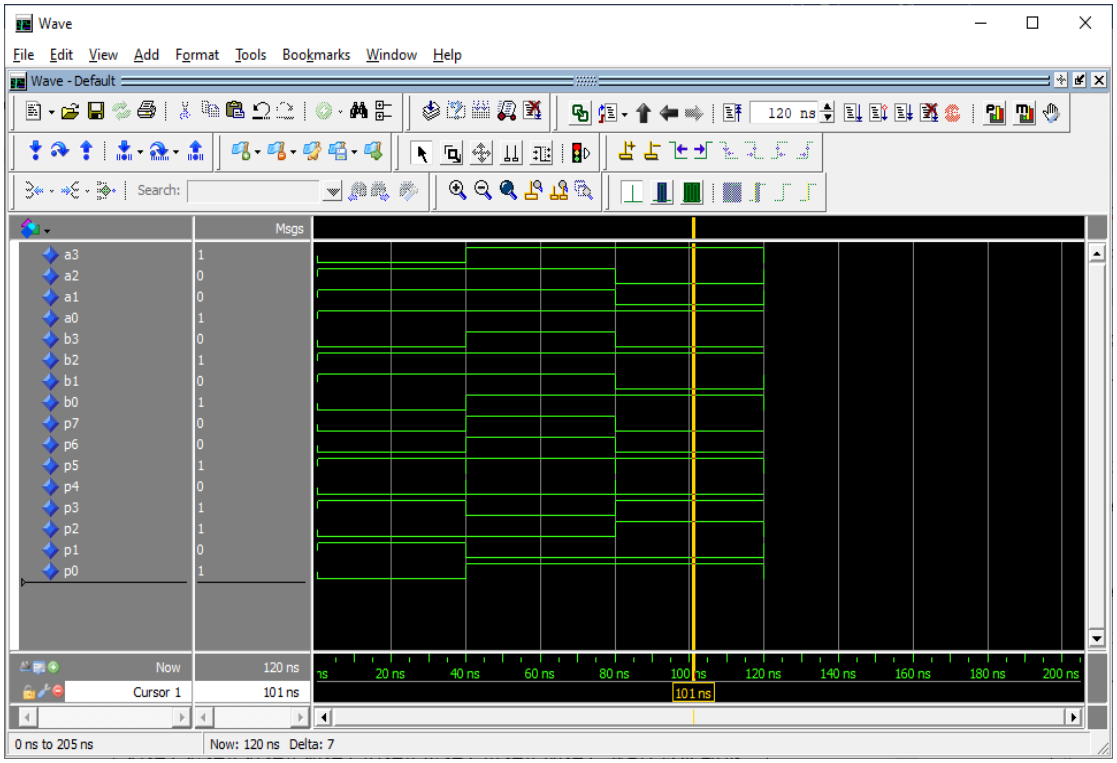
3.9.2 Προσομοίωση πολλαπλασιαστή Wallace tree 4x4 με την VHDL

Η προσομοίωση του πολλαπλασιαστή Wallace Tree 4x4 γίνεται με την δημιουργία Testbench στο οποίο οι πράξεις εκτελούνται ανά 40 ns. Επίσης, παρατηρούνται τα αποτελέσματα της προσομοίωσης σε διάγραμμα Waveform.

```
LIBRARY IEEE;

USE IEEE.STD_LOGIC_1164.ALL;


ENTITY Wallace4tb IS

END Wallace4tb;


ARCHITECTURE behavior OF Wallace4tb IS
```

```
SIGNAL a3, a2, a1, a0: STD_LOGIC;
SIGNAL b3, b2, b1, b0: STD_LOGIC;
SIGNAL p7,p6,p5,p4,p3, p2, p1, p0: STD_LOGIC;

COMPONENT Wallace4
PORT (a3, a2, a1, a0: IN STD_LOGIC;
     b3, b2, b1, b0: IN STD_LOGIC;
     p7,p6,p5,p4,p3, p2, p1, p0: OUT STD_LOGIC);
END COMPONENT;

 BEGIN

m1: Wallace4 PORT MAP(a3=>a3,a2=>a2,a1=>a1,a0=>a0,
                      b3=>b3,b2=>b2,b1=>b1,b0=>b0,
p7=>p7,p6=>p6,p5=>p5,p4=>p4,p3=>p3,p2=>p2,p1=>p1,p0=>p0);

PROCESS
BEGIN

 a3<='0';a2<='1';a1<='1';a0<='1';b3<='0';b2<='1';b1<='1';b0<='0'; WAIT FOR 40 ns;
 a3<='1';a2<='1';a1<='1';a0<='1';b3<='1';b2<='1';b1<='1';b0<='1'; WAIT FOR 40 ns;
 a3<='1';a2<='0';a1<='0';a0<='1';b3<='0';b2<='1';b1<='0';b0<='1'; WAIT FOR 40 ns;

 END PROCESS;
END behavior;
```

Σχήμα 3.72 Testbench πολλαπλασιαστή Wallace tree 4x4 VHDL.



Σχήμα 3.73 Προσομοίωση πολλαπλασιαστή Wallace tree 4x4 VHDL.

3.9.3 Περιγραφή Wallace tree πολλαπλασιαστή 4x4 με την Verilog

Ακολουθεί η υλοποίηση σε Verilog της αρχιτεκτονικής Wallace tree για τέσσερα bit.

```
module HA(output sum, carry, input a, b);
        assign sum = a^b;
        assign carry = (a&b);
endmodule
module FA(output sum, carry, input a, b, cin);
        assign sum =(a^b^cin);
        assign carry = ((a&b)|(a&cin)|(b&cin));
endmodule
module wallace_4x4(output [7:0]product, input [3:0] in1,in2,input clk);
wire [11:0] s, c;
```

```
  //stage 1
  HA HA1(s[0],c[0],(in1[1]&in2[0]),(in1[0]&in2[1])); //p1
FA FA1(s[1],c[1],(in1[2]&in2[0]),(in1[1]&in2[1]),(in1[0]&in2[2]));
FA FA2(s[2],c[2],(in1[3]&in2[0]),(in1[2]&in2[1]),(in1[1]&in2[2]));
HA HA2(s[3],c[3],(in1[3]&in2[1]),(in1[2]&in2[2]));
//stage 2
HA HA3(s[4],c[4],c[0],s[1]); //p2
FA FA3(s[5],c[5],c[1],s[2],(in1[0]&in2[3]));
FA FA4(s[6],c[6],c[2],s[3],(in1[1]&in2[3]));
FA FA5(s[7],c[7],c[3],(in1[3]&in2[2]),(in1[2]&in2[3]));
//stage 3
HA HA4(s[8],c[8],s[5],c[4]); //p3
FA FA6(s[9],c[9],s[6],c[5],c[8]); //p4
FA FA7(s[10],c[10],s[7],c[6],c[9]); //p5
FA FA8(s[11],c[11],c[10],c[7],(in1[3]&in2[3])); p//6,7


assign product[0]= (in1[0]&in2[0]); //p0
assign product[1] = s[0];
assign product[2] = s[4];
assign product[3] = s[8];
assign product[4] = s[9];
assign product[5] = s[10];
assign product[6] = s[11];
assign product[7] = c[11];
  endmodule
```

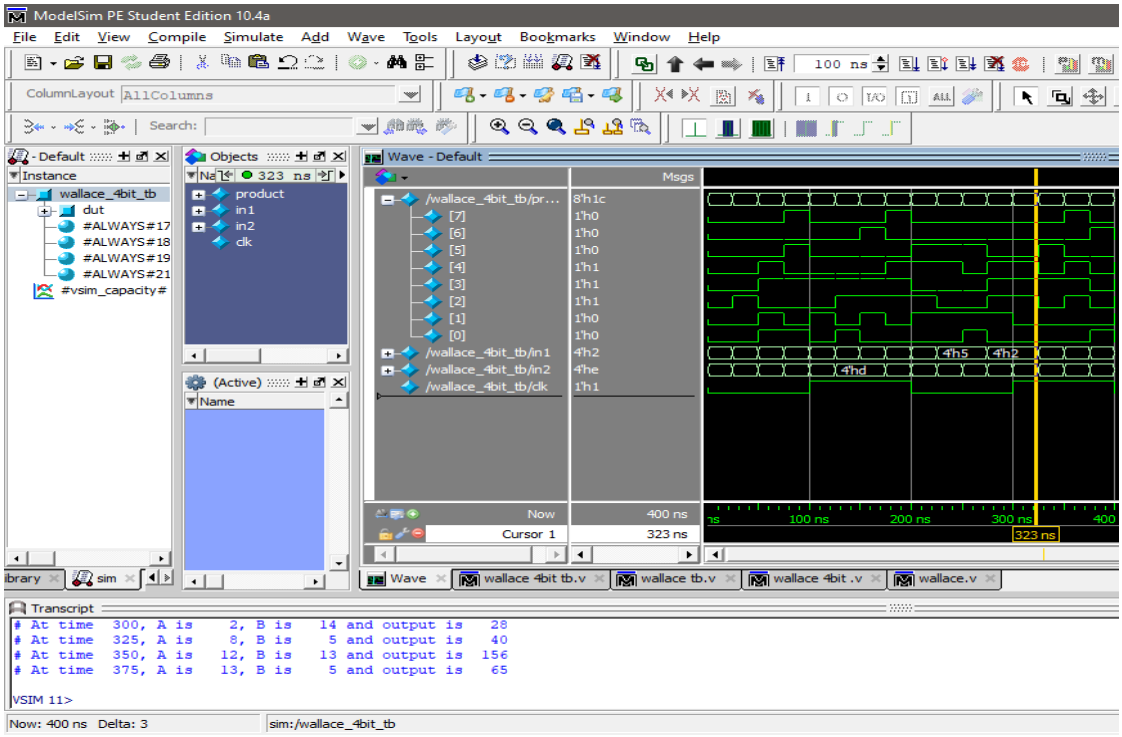
Σχήμα 3.74 Περιγραφή του πολλαπλασιαστή Wallace tree 8bit με χρήση Verilog

3.9.4 Προσομοίωση πολλαπλασιαστή Wallace tree 4x4 με την Verilog

Ο πολλαπλασιαστής Wallace tree 4x4 ενώ αλλάζει σε δομικό επίπεδο πάρα πολύ συνεχίζει να έχει τις ίδιες εισόδους και έξοδο με τις προηγούμενες δύο αρχιτεκτονικές. Δηλαδή, δύο είσοδοι για πολλαπλασιαστή και πολλαπλασιαστέο, και μια έξοδος για γινόμενο. Οι είσοδοι είναι τεσσάρων bit ενώ η έξοδος των οκτώ. Αρχικοποιούνται όλα στο μηδέν και υπάρχουν αλλαγές ανά εικοσιπέντε νανοδευτερόλεπτα. Παρακάτω είναι το testbench και η προσομοίωση.

```
module wallace_4x4_tb(); //setting inputs and output
        wire [7:0] product;
        reg [3:0] in1;
        reg [3:0] in2;
        reg clk;
wallace_4x4 dut (.in1(in1), .in2(in2), .product(product), .clk(clk)); // initialization
        initial begin
                in1 <= 0;
                in2 <= 0;
                clk <= 0;
        end
        always #100 clk <= ~clk;
        always #25 in1 <= $random; // every 25ns value of input is randomized
        always #25 in2 <= $random;
        always @(in1 or in2) // if any input changes the message appears
$monitor ("At time %4d, A is %4d, B is %4d and output is %4d", $time, in1, in2, product);
endmodule
```

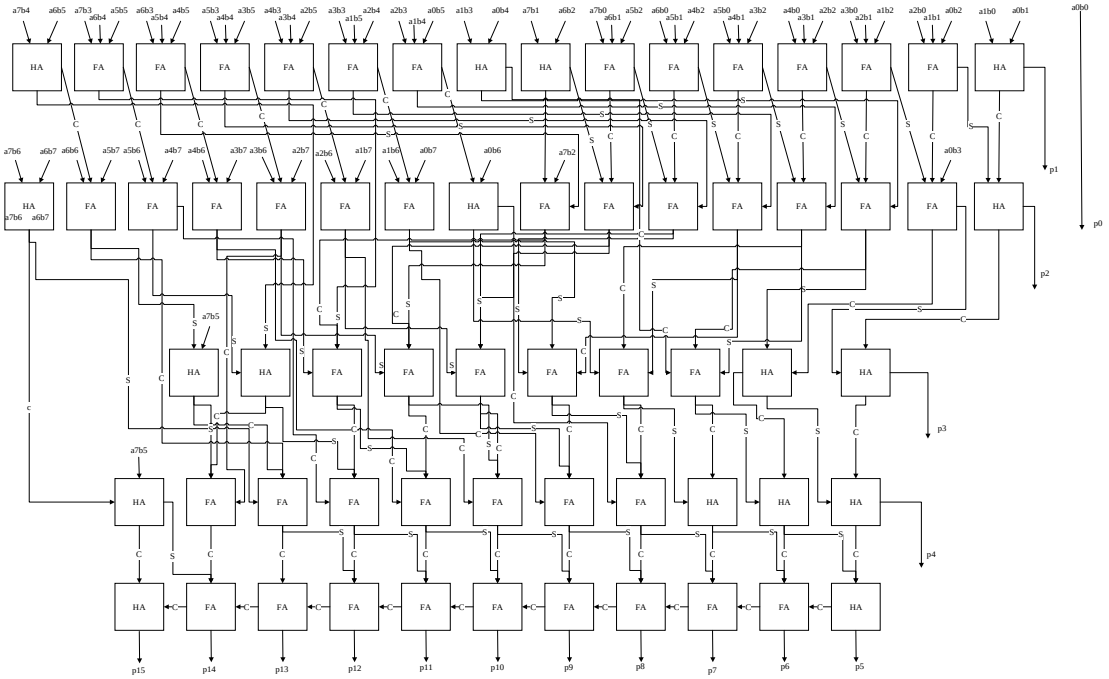
Σχήμα 3.75 Testbench πολλαπλασιαστή Wallace tree 4x4



Σχήμα 3.76 Προσομοίωση πολλαπλασιαστή Wallace tree 4x4

3.10 Περιγραφή Wallace tree πολλαπλασιαστή 8x8 με VHDL και Verilog

Ο πολλαπλασιαστής Wallace Tree 8x8 βασίζεται στην ίδια αρχιτεκτονική με τον 4x4 αλλά είναι τροποποιημένος για αριθμούς πολύ μεγαλύτερου μήκους. Χρησιμοποιεί δεκαοκτώ ημιαθροιστές και σαράντα επτά πλήρεις αθροιστές.



Σχήμα 3.77 Δομή πολλαπλασιαστή Wallace Tree 8x8

3.10.1 Περιγραφή Wallace tree πολλαπλασιαστή 8x8 με VHDL

Παρακάτω παρατίθεται ο κώδικας VHDL του Wallace Tree πολλαπλασιαστή 8x8.

```
LIBRARY ieee;

LIBRARY work;

USE ieee.std_logic_1164.all;

USE work.full_add_package.all;

USE work.half_add_package.all;


ENTITY Walance8 IS

        PORT (                                  a7, a6, a5, a4, a3, a2, a1, a0 : IN
STD_LOGIC;

                                        b7, b6, b5, b4, b3, b2, b1, b0 : IN  STD_LOGIC;

          p15, p14, p13, p12, p11, p10, p9, p8, p7,p6,p5,p4,p3, p2, p1, p0 : OUT
STD_LOGIC);

END Walance8;


ARCHITECTURE structured OF Walance8 IS

        SIGNAL a0b0,a1b0,a2b0,a3b0,a4b0,a5b0,a6b0,a7b0 : STD_LOGIC;

        SIGNAL a0b1,a1b1,a2b1,a3b1,a4b1,a5b1,a6b1,a7b1 : STD_LOGIC;

        SIGNAL a0b2,a1b2,a2b2,a3b2,a4b2,a5b2,a6b2,a7b2 : STD_LOGIC;

        SIGNAL a0b3,a1b3,a2b3,a3b3,a4b3,a5b3,a6b3,a7b3 : STD_LOGIC;

        SIGNAL a0b4,a1b4,a2b4,a3b4,a4b4,a5b4,a6b4,a7b4 : STD_LOGIC;

        SIGNAL a0b5,a1b5,a2b5,a3b5,a4b5,a5b5,a6b5,a7b5 : STD_LOGIC;

        SIGNAL a0b6,a1b6,a2b6,a3b6,a4b6,a5b6,a6b6,a7b6 : STD_LOGIC;

        SIGNAL a0b7,a1b7,a2b7,a3b7,a4b7,a5b7,a6b7,a7b7 : STD_LOGIC;

        SIGNAL
c101,c102,c103,c104,c105,c106,c107,c108,c109,c110,c111,c112,c113,c114,c115,c11
6: STD_LOGIC;

        SIGNAL
```

```
ap101,ap102,ap103,ap104,ap105,ap106,ap107,ap108,ap109,ap110,ap111,ap112,ap113,ap114,ap115,ap116: STD_LOGIC;

        SIGNAL c201,c202,c203,c204,c205,c206,c207,c208,c209,c210,c211,c212,c213,c214,c215,c216: STD_LOGIC;

        SIGNAL ap201,ap202,ap203,ap204,ap205,ap206,ap207,ap208,ap209,ap210,ap211,ap212,ap213,ap214,ap215,ap216: STD_LOGIC;

        SIGNAL c301,c302,c303,c304,c305,c306,c307,c308,c309,c310: STD_LOGIC;

        SIGNAL ap301,ap302,ap303,ap304,ap305,ap306,ap307,ap308,ap309,ap310: STD_LOGIC;

        SIGNAL c401,c402,c403,c404,c405,c406,c407,c408,c409,c410,c411: STD_LOGIC;

        SIGNAL ap401,ap402,ap403,ap404,ap405,ap406,ap407,ap408,ap409,ap410,ap411: STD_LOGIC;

        SIGNAL c501,c502,c503,c504,c505,c506,c507,c508,c509,c510,c511: STD_LOGIC;

        SIGNAL ap501,ap502,ap503,ap504,ap505,ap506,ap507,ap508,ap509,ap510,ap511: STD_LOGIC;




BEGIN

--Stage 1



a0b0<= a0 and b0;

a1b0<= a1 and b0;

a2b0<= a2 and b0;

a3b0<= a3 and b0;
```

```
a4b0<= a4 and b0;

a5b0<= a5 and b0;

a6b0<= a6 and b0;

a7b0<= a7 and b0;


a0b1<= a0 and b1;

a1b1<= a1 and b1;

a2b1<= a2 and b1;

a3b1<= a3 and b1;

a4b1<= a4 and b1;

a5b1<= a5 and b1;

a6b1<= a6 and b1;

a7b1<= a7 and b1;


a0b2<= a0 and b2;

a1b2<= a1 and b2;

a2b2<= a2 and b2;

a3b2<= a3 and b2;

a4b2<= a4 and b2;

a5b2<= a5 and b2;

a6b2<= a6 and b2;

a7b2<= a7 and b2;


a0b3<= a0 and b3;

a1b3<= a1 and b3;

a2b3<= a2 and b3;
```

```
a3b3<= a3 and b3;

a4b3<= a4 and b3;

a5b3<= a5 and b3;

a6b3<= a6 and b3;

a7b3<= a7 and b3;


a0b4<= a0 and b4;

a1b4<= a1 and b4;

a2b4<= a2 and b4;

a3b4<= a3 and b4;

a4b4<= a4 and b4;

a5b4<= a5 and b4;

a6b4<= a6 and b4;

a7b4<= a7 and b4;


a0b5<= a0 and b5;

a1b5<= a1 and b5;

a2b5<= a2 and b5;

a3b5<= a3 and b5;

a4b5<= a4 and b5;

a5b5<= a5 and b5;

a6b5<= a6 and b5;

a7b5<= a7 and b5;


a0b6<= a0 and b6;

a1b6<= a1 and b6;
```

```
a2b6<= a2 and b6;

a3b6<= a3 and b6;

a4b6<= a4 and b6;

a5b6<= a5 and b6;

a6b6<= a6 and b6;

a7b6<= a7 and b6;


a0b7<= a0 and b7;

a1b7<= a1 and b7;

a2b7<= a2 and b7;

a3b7<= a3 and b7;

a4b7<= a4 and b7;

a5b7<= a5 and b7;

a6b7<= a6 and b7;

a7b7<= a7 and b7;


--Stage 2

--Part 1--

HA1:half_add PORT MAP(a1b0,a0b1,c101,ap101);

FA1:full_add PORT MAP(a2b0,a1b1,a0b2,c102,ap102);

FA2:full_add PORT MAP(a3b0,a2b1,a1b2,c103,ap103);

FA3:full_add PORT MAP(a4b0,a3b1,a2b2,c104,ap104);

FA4:full_add PORT MAP(a5b0,a4b1,a3b2,c105,ap105);

FA5:full_add PORT MAP(a6b0,a5b1,a4b2,c106,ap106);

FA6:full_add PORT MAP(a7b0,a6b1,a5b2,c107,ap107);

HA2:half_add PORT MAP(a7b1,a6b2,c108,ap108);
```

```
HA3:half_add PORT MAP(a1b3,a0b4,c109,ap109);
FA7:full_add PORT MAP(a2b3,a1b4,a0b5,c110,ap110);
FA8:full_add PORT MAP(a3b3,a2b4,a1b5,c111,ap111);
FA9:full_add PORT MAP(a4b3,a3b4,a2b5,c112,ap112);
FA10:full_add PORT MAP(a5b3,a4b4,a3b5,c113,ap113);
FA11:full_add PORT MAP(a6b3,a5b4,a4b5,c114,ap114);
FA12:full_add PORT MAP(a7b3,a6b4,a5b5,c115,ap115);
HA4:half_add PORT MAP(a7b4,a6b5,c116,ap116);

--part 2--
HA5:half_add PORT MAP(ap102,c101,c201,ap201);
FA13:full_add PORT MAP(ap103,c102,a0b3,c202,ap202);
FA14:full_add PORT MAP(ap104,c103,ap109,c203,ap203);
FA15:full_add PORT MAP(ap105,c104,ap110,c204,ap204);
FA16:full_add PORT MAP(ap106,c105,ap111,c205,ap205);
FA17:full_add PORT MAP(ap107,c106,ap112,c206,ap206);
FA18:full_add PORT MAP(ap108,c107,ap113,c207,ap207);
FA19:full_add PORT MAP(a7b2,c108,ap114,c208,ap208);

HA6:half_add PORT MAP(c110,a0b6,c209,ap209);
FA20:full_add PORT MAP(c111,a1b6,a0b7,c210,ap210);
FA21:full_add PORT MAP(c112,a2b6,a1b7,c211,ap211);
FA22:full_add PORT MAP(c113,a3b6,a2b7,c212,ap212);
FA23:full_add PORT MAP(c114,a4b6,a3b7,c213,ap213);
FA24:full_add PORT MAP(c115,a5b6,a4b7,c214,ap214);
```

```
FA25:full_add PORT MAP(c116,a6b6,a5b7,c215,ap215);
HA7:half_add PORT MAP(a7b6,a6b7,c216,ap216);



--part 3--
HA8:half_add PORT MAP(ap202,c201,c301,ap301);
HA9:half_add PORT MAP(ap203,c202,c302,ap302);
FA26:full_add PORT MAP(ap204,c203,c109,c303,ap303);
FA27:full_add PORT MAP(ap205,c204,ap209,c304,ap304);
FA28:full_add PORT MAP(ap206,c205,ap210,c305,ap305);
FA29:full_add PORT MAP(ap207,c206,ap211,c306,ap306);
FA30:full_add PORT MAP(ap208,c207,ap212,c307,ap307);
FA31:full_add PORT MAP(ap115,c208,ap213,c308,ap308);
HA11:half_add PORT MAP(ap116,ap214,c309,ap309);
HA12:half_add PORT MAP(a7b5,ap215,c310,ap310);

--part 4--
HA13:half_add PORT MAP(ap302,c301,c401,ap401);
HA14:half_add PORT MAP(ap303,c302,c402,ap402);
HA15:half_add PORT MAP(ap304,c303,c403,ap403);
FA32:full_add PORT MAP(ap305,c304,c209,c404,ap404);
FA33:full_add PORT MAP(ap306,c305,c210,c405,ap405);
FA34:full_add PORT MAP(ap307,c306,c211,c406,ap406);
FA35:full_add PORT MAP(ap308,c307,c212,c407,ap407);
FA36:full_add PORT MAP(ap309,c308,c213,c408,ap408);
FA37:full_add PORT MAP(ap310,c309,c214,c409,ap409);
```

```
FA38:full_add PORT MAP(ap216,c310,c215,c410,ap410);
HA16:half_add PORT MAP(a7b7,c216,c411,ap411);

--part 5--
HA17:half_add PORT MAP(ap402,c401,c501,ap501);
FA39:full_add PORT MAP(ap403,c402,c501,c502,ap502);
FA40:full_add PORT MAP(ap404,c403,c502,c503,ap503);
FA41:full_add PORT MAP(ap405,c404,c503,c504,ap504);
FA42:full_add PORT MAP(ap406,c405,c504,c505,ap505);
FA43:full_add PORT MAP(ap407,c406,c505,c506,ap506);
FA44:full_add PORT MAP(ap408,c407,c506,c507,ap507);
FA45:full_add PORT MAP(ap409,c408,c507,c508,ap508);
FA46:full_add PORT MAP(ap410,c409,c508,c509,ap509);
FA47:full_add PORT MAP(ap411,c410,c509,c510,ap510);
HA18:half_add PORT MAP(c510,c411,c511,ap511);

--Final--

p15<=ap511;
p14<=ap510;
p13<=ap509;
p12<=ap508;
p11<=ap507;
p10<=ap506;
p9<=ap505;
p8<=ap504;
```

```
p7<=ap503;

p6<=ap502;

p5<=ap501;

p4<=ap401;

p3<=ap301;

p2<=ap201;

p1<=ap101;

p0<=a0b0;

END structured;
```

Σχήμα 3.78 Περιγραφή του πολλαπλασιαστή Wallace tree 8x8 με χρήση VHDL

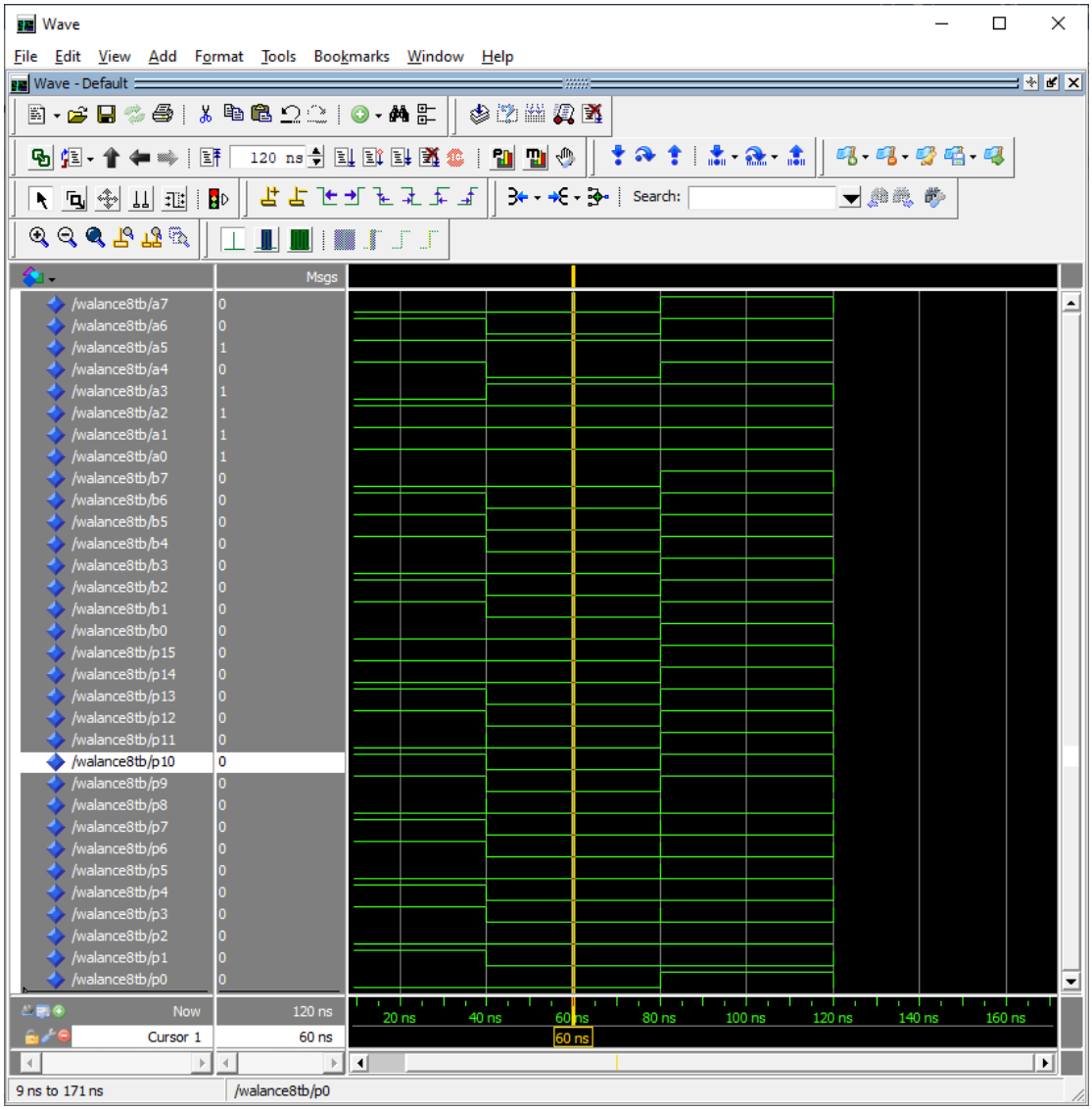
3.10.2 Προσομοίωση πολλαπλασιαστή Wallace tree 8x8 με την VHDL

Ακολουθεί η προσομοίωση του Wallace Tree 8x8 με την βοήθεια Testbench καθώς και τα αποτελέσματα της προσομοίωσης σε Waveform. Κάθε πράξη εκτελείται σε χρόνο 40 ns.

```
LIBRARY IEEE;

USE IEEE.STD_LOGIC_1164.ALL;



ENTITY Walance8tb IS

END Walance8tb;



ARCHITECTURE behavior OF Walance8tb IS



SIGNAL  a7, a6, a5, a4, a3, a2, a1, a0: STD_LOGIC;

SIGNAL  b7, b6, b5, b4, b3, b2, b1, b0: STD_LOGIC;

SIGNAL  p15, p14, p13, p12, p11, p10, p9, p8, p7, p6, p5, p4, p3, p2, p1, p0:
STD_LOGIC;
```

```
COMPONENT Walance8

PORT (a7, a6, a5, a4, a3, a2, a1, a0: IN STD_LOGIC;

    b7, b6, b5, b4, b3, b2, b1, b0: IN STD_LOGIC;

    p15, p14, p13, p12, p11, p10, p9, p8, p7, p6, p5, p4, p3, p2, p1, p0: OUT
STD_LOGIC);

END COMPONENT;



 BEGIN



m1: Walance8 PORT
MAP(a7=>a7,a6=>a6,a5=>a5,a4=>a4,a3=>a3,a2=>a2,a1=>a1,a0=>a0,

                    b7=>b7,b6=>b6,b5=>b5,b4=>b4,b3=>b3,b2=>b2,b1=>b1,b0=>b0,

p15=>p15,p14=>p14,p13=>p13,p12=>p12,p11=>p11,p10=>p10,p9=>p9,p8=>p8,p7=
>p7,p6=>p6,p5=>p5,p4=>p4,p3=>p3,p2=>p2,p1=>p1,p0=>p0);



PROCESS

BEGIN




a7<='0';a6<='1';a5<='1';a4<='1';a3<='0';a2<='1';a1<='1';a0<='1';b7<='0';b6<='1';b5<='
1';b4<='1';b3<='0';b2<='1';b1<='1';b0<='0'; WAIT FOR 40 ns;


a7<='0';a6<='0';a5<='1';a4<='0';a3<='1';a2<='1';a1<='1';a0<='1';b7<='0';b6<='0';b5<='
0';b4<='0';b3<='0';b2<='0';b1<='0';b0<='0'; WAIT FOR 40 ns;


a7<='1';a6<='1';a5<='1';a4<='1';a3<='1';a2<='1';a1<='1';a0<='1';b7<='1';b6<='1';b5<='
1';b4<='1';b3<='1';b2<='1';b1<='1';b0<='1'; WAIT FOR 40 ns;

 END PROCESS;

END behavior;
```

Σχήμα 3.79 Testbench πολλαπλασιαστή Wallace tree 8x8 VHDL.



Σχήμα 3.80 Προσομοίωση πολλαπλασιαστή Wallace tree 8x8 VHDL.

3.10.3 Περιγραφή Wallace tree πολλαπλασιαστή 8x8 με Verilog

Ακολουθεί η υλοποίηση σε Verilog της αρχιτεκτονικής Wallace Tree για οκτώ bit.

```
module HA(output sum, carry, input a, b);
        assign sum = a^b;
        assign carry = (a&b);
endmodule
module FA(output sum, carry, input a, b, cin);
        assign sum =(a^b^cin);
```

```
        assign carry = ((a&b)|(a&cin)|(b&cin));
 endmodule
 module wallace (output [15:0]product, input [7:0] in1,in2,input clk);
 wire [63:0] s, c;
 //stage 1
 HA HA1(s[0],c[0],(in1[1]&in2[0]),(in1[0]&in2[1])); //p1
FA FA1(s[1],c[1],(in1[2]&in2[0]),(in1[1]&in2[1]),(in1[0]&in2[2]));
FA FA2(s[2],c[2],(in1[3]&in2[0]),(in1[2]&in2[1]),(in1[1]&in2[2]));
FA FA3(s[3],c[3],(in1[4]&in2[0]),(in1[3]&in2[1]),(in1[2]&in2[2]));
FA FA4(s[4],c[4],(in1[5]&in2[0]),(in1[4]&in2[1]),(in1[3]&in2[2]));
FA FA5(s[5],c[5],(in1[6]&in2[0]),(in1[5]&in2[1]),(in1[4]&in2[2]));
FA FA6(s[6],c[6],(in1[7]&in2[0]),(in1[6]&in2[1]),(in1[5]&in2[2]));
HA HA2(s[7],c[7],(in1[7]&in2[1]),(in1[6]&in2[2]));
HA HA3 (s[8],c[8],(in1[1]&in2[3]),(in1[0]&in2[4]));
FA FA7 (s[9],c[9],(in1[2]&in2[3]),(in1[1]&in2[4]),(in1[0]&in2[5]));
FA FA8 (s[10],c[10],(in1[3]&in2[3]),(in1[2]&in2[4]),(in1[1]&in2[5]));
FA FA9 (s[11],c[11],(in1[4]&in2[3]),(in1[3]&in2[4]),(in1[2]&in2[5]));
FA FA10(s[12],c[12],(in1[5]&in2[3]),(in1[4]&in2[4]),(in1[3]&in2[5]));
FA FA11(s[13],c[13],(in1[6]&in2[3]),(in1[5]&in2[4]),(in1[4]&in2[5]));
FA FA12(s[14],c[14],(in1[7]&in2[3]),(in1[6]&in2[4]),(in1[5]&in2[5]));
HA HA4 (s[15],c[15],(in1[7]&in2[4]),(in1[6]&in2[5]));
//stage 2
HA HA5 (s[16],c[16],s[1],c[0]);
FA FA13(s[17],c[17],(in1[0]&in2[3]),s[2],c[1]);
FA FA14(s[18],c[18],s[8],s[3],c[2]);
FA FA15(s[19],c[19],s[9],s[4],c[3]);
```

```
FA FA16(s[20],c[20],s[10],s[5],c[4]);
FA FA17(s[21],c[21],s[11],s[6],c[5]);
FA FA18(s[22],c[22],s[12],s[7],c[6]);
FA FA19(s[23],c[23],s[13],(in1[7]&in2[2]),c[7]);
HA HA6 (s[24],c[24],c[9],(in1[0]&in2[6]));
FA FA20(s[25],c[25],c[10],(in1[1]&in2[6]),(in1[0]&in2[7]));
FA FA21(s[26],c[26],c[11],(in1[2]&in2[6]),(in1[1]&in2[7]));
FA FA22(s[27],c[27],c[12],(in1[3]&in2[6]),(in1[2]&in2[7]));
FA FA23(s[28],c[28],c[13],(in1[4]&in2[6]),(in1[3]&in2[7]));
FA FA24(s[29],c[29],c[14],(in1[5]&in2[6]),(in1[4]&in2[7]));
FA FA25(s[30],c[30],c[15],(in1[6]&in2[6]),(in1[5]&in2[7]));
HA HA7 (s[31],c[31],(in1[7]&in2[6]),(in1[6]&in2[7]));
//stage 3
HA HA8 (s[32],c[32],c[16],s[17]);
HA HA9 (s[33],c[33],c[17],s[18]);
FA FA26(s[34],c[34],c[18],s[19],c[8]);
FA FA27(s[35],c[35],c[19],s[20],s[24]);
FA FA28(s[36],c[36],c[20],s[21],s[25]);
FA FA29(s[37],c[37],c[21],s[22],s[26]);
FA FA30(s[38],c[38],c[22],s[23],s[27]);
FA FA31(s[39],c[39],c[23],s[14],s[28]);
HA HA11(s[40],c[40],s[15],s[29]);
HA HA12(s[41],c[41],s[30],(in1[7]&in2[5]));
//stage 4
HA HA13(s[42],c[42],c[32],s[33]);
HA HA14(s[43],c[43],c[33],s[34]);
```

```
HA HA15(s[44],c[44],c[34],s[35]);
FA FA32(s[45],c[45],c[35],s[36],c[24]);
FA FA33(s[46],c[46],c[36],s[37],c[25]);
FA FA34(s[47],c[47],c[37],s[38],c[26]);
FA FA35(s[48],c[48],c[38],s[39],c[27]);
FA FA36(s[49],c[49],c[39],s[40],c[28]);
FA FA37(s[50],c[50],c[40],s[41],c[29]);
FA FA38(s[51],c[51],c[41],s[31],c[30]);
HA HA16(s[52],c[52],c[31],(in1[7]&in2[7]));
//stage 5
HA HA17(s[53],c[53],c[42],s[43]);
FA FA39(s[54],c[54],c[43],c[53],s[44]);
FA FA40(s[55],c[55],c[44],c[54],s[45]);
FA FA41(s[56],c[56],c[45],c[55],s[46]);
FA FA42(s[57],c[57],c[46],c[56],s[47]);
FA FA43(s[58],c[58],c[47],c[57],s[48]);
FA FA44(s[59],c[59],c[48],c[58],s[49]);
FA FA45(s[60],c[60],c[49],c[59],s[50]);
FA FA46(s[61],c[61],c[50],c[60],s[51]);
FA FA47(s[62],c[62],c[51],c[61],s[52]);
HA HA18(s[63],c[63],c[62],c[52]); // final carry always 0
assign product[0]= (in1[0]&in2[0]); //p0
assign product[1] = s[0];
assign product[2] = s[16];
assign product[3] = s[32];
assign product[4] = s[42];
```

```
assign product[5] = s[53];

assign product[6] = s[54];

assign product[7] = s[55];

assign product[8] = s[56];

assign product[9] = s[57];

assign product[10] = s[58];

assign product[11] = s[59];

assign product[12] = s[60];

assign product[13] = s[61];

assign product[14] = s[62];

assign product[15] = s[63];

endmodule
```

Σχήμα 3.81 Περιγραφή του πολλαπλασιαστή Wallace tree 8x8 με χρήση Verilog

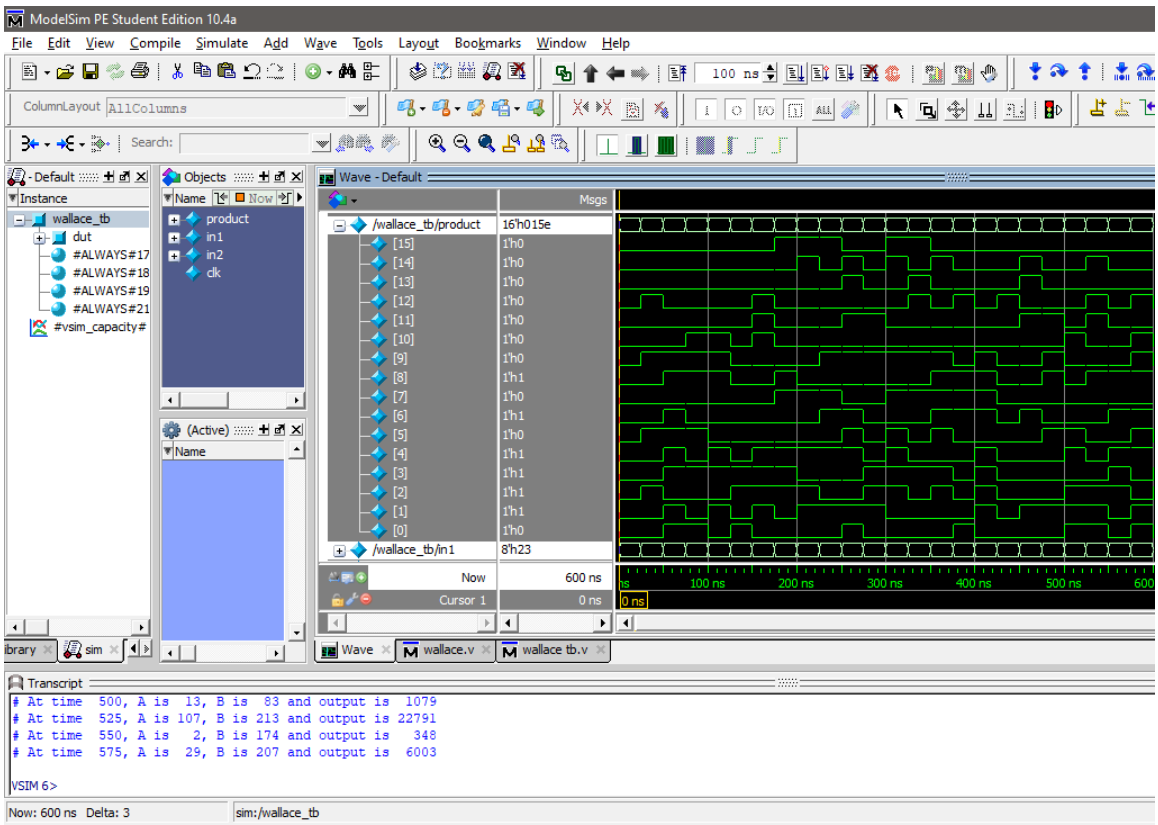
3.10.4 Προσομοίωση πολλαπλασιαστή Wallace tree 8x8 με την Verilog

Ο πολλαπλασιαστής Wallace tree 8x8 έχει την ίδια ακριβώς λειτουργικότητα με τον πολλαπλασιαστή Wallace tree 4x4 αλλά είναι δομημένος για εισόδους των οκτώ bit. Η έξοδος αντίστοιχα είναι δεκαέξι bit και η αρχικοποίηση γίνεται στο μηδέν. Ανά εικοσιπέντε νανοδευτερόλεπτα γίνονται αλλαγές στις εισόδους. Η δομή του φαίνεται στο σχήμα 3.77. Ακολουθούν το testbench και η προσομοίωση.

```
module wallace_tb(); //setting inputs and output

        wire [15:0] product;

        reg [7:0] in1;

        reg [7:0] in2;

        reg clk;

wallace dut (.in1(in1), .in2(in2), .product(product), .clk(clk)); // initialization

        initial begin
```

```
            in1 <= 0;

            in2 <= 0;

            clk <= 0;

        end

        always #100 clk <= ~clk;

        always #25 in1 <= $random; // every 25ns value of input is randomized

        always #25 in2 <= $random;

        always @(in1 or in2) // if any input changes the message appears

$monitor ("At time %4d, A is %d, B is %d and output is %d", $time, in1, in2, product); // message for extra info

endmodule
```

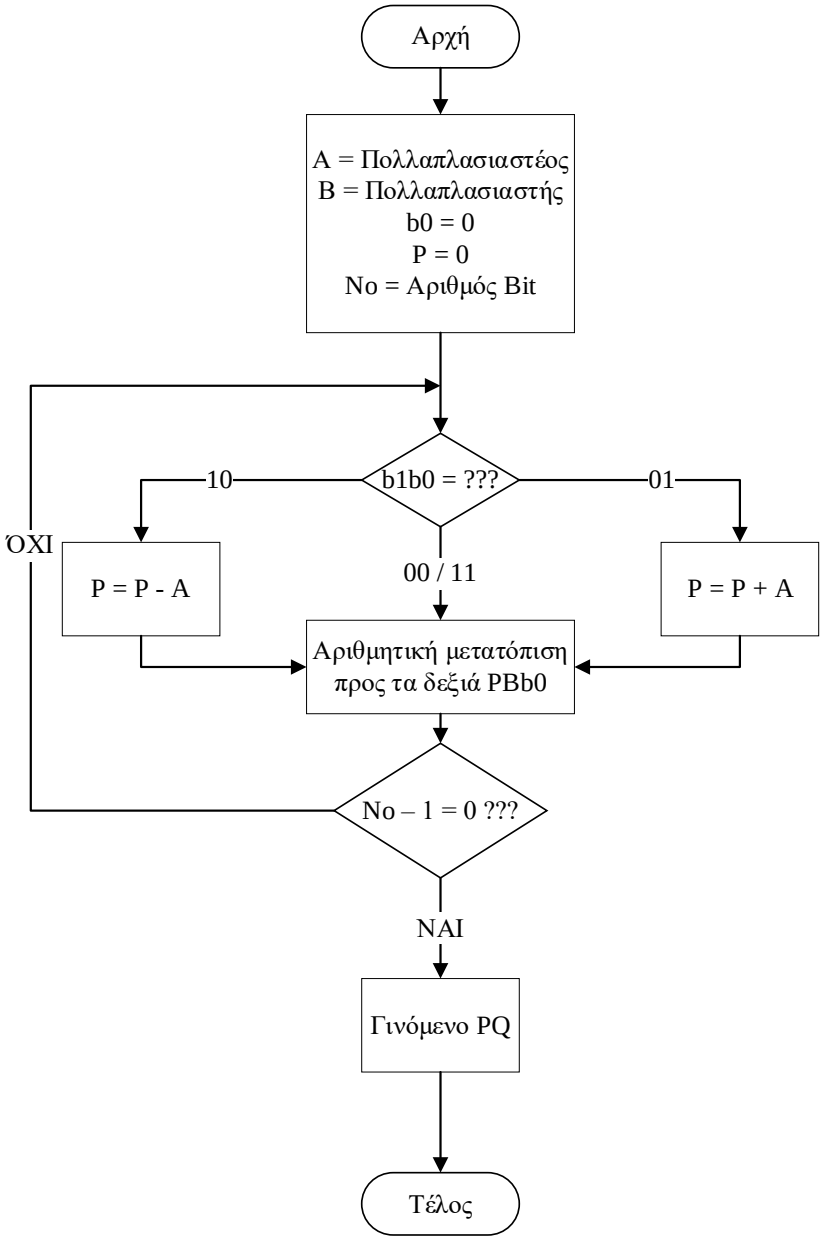
Σχήμα 3.82 Testbench πολλαπλασιαστή Wallace tree 8x8



Σχήμα 3.83 Προσομοίωση πολλαπλασιαστή Wallace tree 8x8

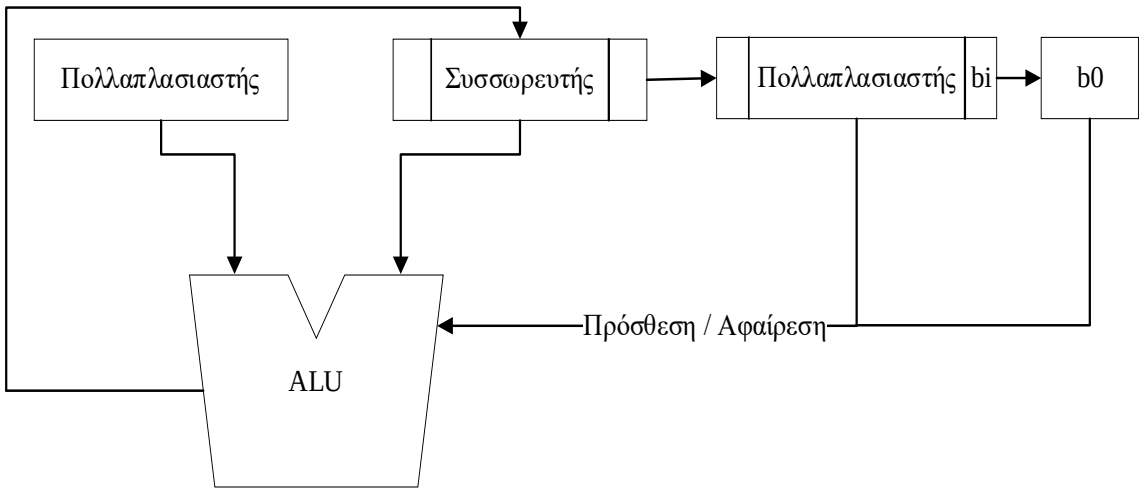
3.11 Περιγραφή πολλαπλασιαστών αρχιτεκτονικής Booth με VHDL και Verilog

Σε αντίθεση με τις προηγούμενες αρχιτεκτονικές οι οποίες αποτελούν διαφοροποιήσεις του κάθετου πολλαπλασιασμού, ο αλγόριθμος πολλαπλασιασμού του Booth λειτουργεί διαφορετικά. Ο αλγόριθμος εξετάζει ζεύγη ψηφιών από τον πολλαπλασιαστή και ανάλογα με τις τιμές τους τροποποιεί τις τιμές ενός συσσωρευτή, καταλήγοντας έτσι στο τελικό αποτέλεσμα. Αν οι τιμές του ζεύγους είναι «01» γίνεται πρόσθεση του πολλαπλασιαστέου, αλλιώς αν οι τιμές του ζεύγους είναι «10» γίνεται αφαίρεση. Σε περίπτωση που το ζεύγος έχει τιμές «00» και «11» τότε δεν πραγματοποιείται καμία αλλαγή. Όταν τελειώσει ο έλεγχος, γίνεται μετατόπιση των ψηφίων προς τα δεξιά και ξεκινά εκ νέου η διαδικασία για όλα τα εναπομείναντα ψηφία του πολλαπλασιαστή.



Σχήμα 3.84 Διάγραμμα Ροής Αλγόριθμου Booth

Για να υπάρχει η δυνατότητα να συμβεί ο παραπάνω έλεγχος και για να μπορεί να λειτουργήσει ο αλγόριθμος για προσημασμένους αριθμούς υπάρχει διαφοροποίηση από την δομή των προηγούμενων αρχιτεκτονικών.



Σχήμα 3.85 Αρχιτεκτονική Πολλαπλασιαστή Booth

Παρακάτω θα ακολουθήσει ένα παράδειγμα για να γίνει πιο απλή η κατανόηση του αλγορίθμου. Θα χρησιμοποιηθούν οι αριθμοί για πολλαπλασιαστή και πολλαπλασιαστέο αντίστοιχα : A = 4 και B = -7. Για να γίνει η ανάλυση θα υλοποιηθεί ένας πίνακας ο οποίος θα εμφανίζει τα στοιχεία σε κάθε βήμα. Ξεκινώντας, όμως $7_{10} = 0111_2$ το οποίο θα υλοποιηθεί σε συμπλήρωμα του 2, όπου καταλήγει σε $–B = 1001_2$. Ο αλγόριθμος επίσης διατηρεί πρόσημα.

- Βήμα 1 : Το τελευταίο ψηφίο του πολλαπλασιαστή και το q0 είναι ίδια άρα δεν υπάρχει αλλαγή και γίνεται μετατόπιση προς τα δεξιά. Το ψηφίο που εισάγεται είναι ίδιο με το προηγούμενο που υπήρχε στην θέση του.
- Βήμα 2 : Συνεχίζει να μην υπάρχει κάποια αλλαγή στα τελευταία ψηφία. Γίνεται μετατόπιση προς τα δεξιά.
- Βήμα 3 : Τα τελευταία ψηφία είναι 10, άρα γίνεται αφαίρεση του πολλαπλασιαστέου στο γινόμενο. Σε αυτή τη περίπτωση επειδή έχει μόνο μηδενικά λειτουργεί σαν πρόσθεση. Γίνεται μετατόπιση προς τα δεξιά.
- Βήμα 4 : Τα τελευταία ψηφία είναι 01, άρα γίνεται πρόσθεση του πολλαπλασιαστέου στο γινόμενο. Ο αλγόριθμος δεν διατηρεί κρατούμενα Αφού γίνει η μετατόπιση, έχοντας επεξεργαστεί το τελευταίο ψηφίο του πολλαπλασιαστή, ο αλγόριθμος τελειώνει. $111000_2 = 28_{10}$.

Βήμα	Γινόμενο / Product, P	Πολλαπλασιαστής A, a4a3a2a1	q0	Περιγραφή
1	0000	0100	0	Αρχικοποίηση, q1q0 = 00, Μετατόπιση δεξιά
2	0000	0010	0	q1q0 = 00, Μετατόπιση δεξιά
3	1001	0001	0	q1q0 = 10, Αφαίρεση
	1100	1000	1	Μετατόπιση δεξιά
4	0011	1000	1	q1q0 = 01, Πρόσθεση
	0001	1100	0	Μετατόπιση δεξιά, Τέλος

Πίνακας 1 Παράδειγμα χρήσης αλγορίθμου Booth

3.11.1 Περιγραφή προσημασμένου Booth πολλαπλασιαστή 4x4 με την VHDL

Ακολουθεί η περιγραφή του προσημασμένου Booth πολλαπλασιαστή 4x4 σε VHDL κώδικα.

```
LIBRARY ieee;

USE ieee.std_logic_1164.ALL;

USE ieee.std_logic_signed.ALL;


ENTITY booth4 IS

  PORT

  (

    A, B: IN  STD_LOGIC_VECTOR(3 DOWNTO 0);

     PQ: OUT STD_LOGIC_VECTOR(7 DOWNTO 0)

  );

END booth4;


ARCHITECTURE Behavioral OF booth4 IS
```

```
BEGIN
        PROCESS(A,B)
        VARIABLE NA, Y, P: STD_LOGIC_VECTOR(3 DOWNTO 0);
        VARIABLE b0: STD_LOGIC;
        BEGIN
        NA:=NOT(A)+1;
        Y:=B;
        P:="0000";
        b0:='0';


                FOR i IN 0 TO 3 LOOP
                        IF(Y(0)='0' AND b0='1') THEN
                                P:=P+A;
                        ELSIF(Y(0)='1' AND b0='0') THEN
                                P:=P+NA;
                        END IF;
                        b0:=Y(0);
                        Y(0):=Y(1);
                        Y(1):=Y(2);
                        Y(2):=Y(3);
                        Y(3):=P(0);
                        P(0):=P(1);
                        P(1):=P(2);
```

```
                P(2):=P(3);
            END LOOP;
        PQ(0)<=Y(0);
        PQ(1)<=Y(1);
        PQ(2)<=Y(2);
        PQ(3)<=Y(3);
        PQ(4)<=P(0);
        PQ(5)<=P(1);
        PQ(6)<=P(2);
        PQ(7)<=P(3);
        END PROCESS;
END ARCHITECTURE Behavioral;
```

Σχήμα 3.86 Υλοποίηση VHDL προσημασμένου πολλαπλασιαστή Booth 4x4.

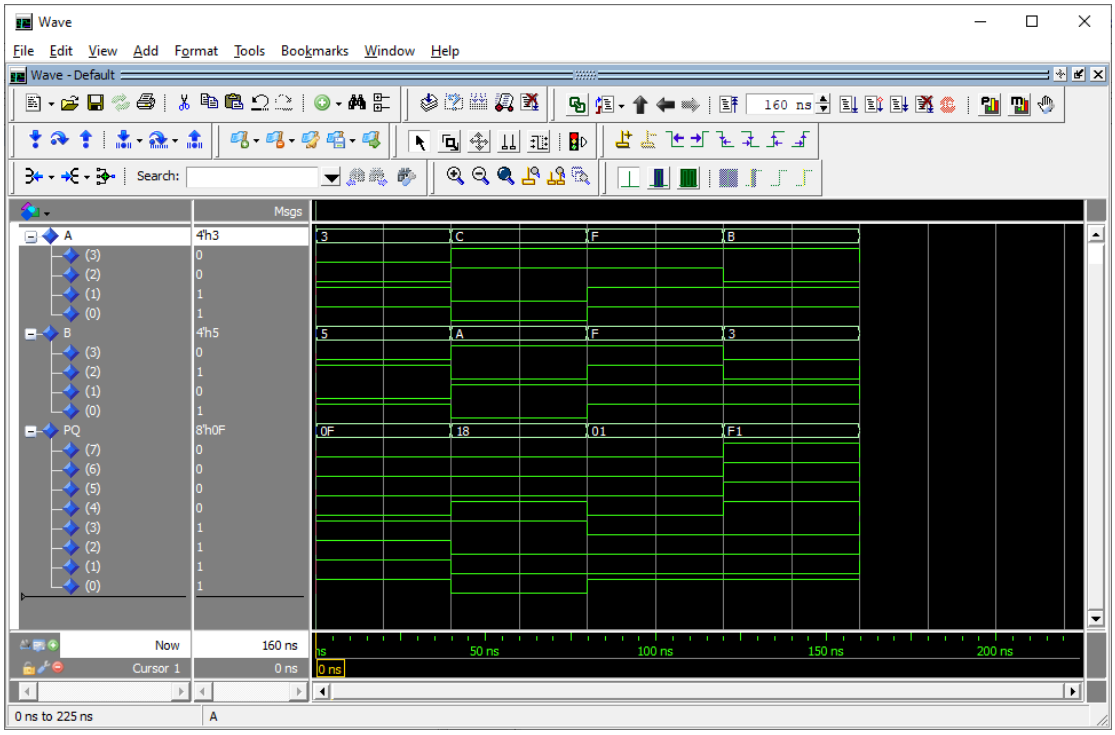
3.11.2 Προσομοίωση προσημασμένου πολλαπλασιαστή Booth 4x4 με την VHDL

Παρακάτω γίνεται προσομοίωση του Booth 4x4 με την VHDL. Ο πολλαπλασιαστής «παίρνει» τιμές «0011» επί «0101», «1100» επί «1010», «1111» επί «1111» και «1011» επί «0011» ανά 40 ns για κάθε πολλαπλασιασμό. Τα αποτελέσματα είναι διακριτά στο διάγραμμα Waveform που ακολουθεί.

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


ENTITY booth4tb IS
END booth4tb;


ARCHITECTURE behavior OF booth4tb IS


SIGNAL A, B : STD_LOGIC_VECTOR(3 DOWNTO 0);
```

```
SIGNAL   PQ : STD_LOGIC_VECTOR(7 DOWNTO 0);


COMPONENT booth4
PORT ( A, B : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
       PQ : OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END COMPONENT;


 BEGIN


m1: booth4 PORT MAP(A=>A,B=>B,PQ=>PQ);


PROCESS
BEGIN


 A<="0011";B<="0101";WAIT FOR 40 ns;
 A<="1100";B<="1010";WAIT FOR 40 ns;
 A<="1111";B<="1111";WAIT FOR 40 ns;
 A<="1011";B<="0011";WAIT FOR 40 ns;


 END PROCESS;
END behavior;
```

Σχήμα 3.87 Testbench προσημασμένου πολλαπλασιαστή Booth 4x4 VHDL.



Σχήμα 3.88 Προσομοίωση προσημασμένου πολλαπλασιαστή Booth 4x4 VHDL.

3.11.3 Περιγραφή προσημασμένου Booth πολλαπλασιαστή 4x4 με την Verilog

Όπως φαίνεται και παραπάνω ο αλγόριθμος του Booth είναι εξαιρετικά ταχύς, απαιτώντας βήματα ίσα με το μήκος των bit που πολλαπλασιάζονται για να ολοκληρωθεί η πράξη. Παρακάτω θα ακολουθήσει η υλοποίηση του αλγορίθμου για αριθμούς τεσσάρων bit με κώδικα Verilog.

```
module booth (input signed [3:0] in1, in2, output reg signed [7:0] product);
        reg signed [1:0] val1; // last 2 bits
        reg val2; // last bit
        reg [3:0] in2_neg; // negative input
        integer i; // num of steps
        always @(in1 , in2) begin // initializations
                product = 8'd0; val2 = 1'd0;
        for (i = 0; i<4; i = i+1) begin // steps needed for completion
                val1 = {in1[i], val2}; in2_neg = -in2;
```

```
case (val1) // addition or subtraction
                2'd2 : product[7:4] = product[7:4] + in2_neg;
                2'd1 : product[7:4] = product[7:4] + in2;
        endcase
                product = product >>1; // shifting
                product[7] = product[6];
                val2 = in1[i]; end
        if (in2 == 4'd8) begin // negative check
                product = - product;
        end
end
endmodule
```

Σχήμα 3.89 Υλοποίηση Verilog προσημασμένου πολλαπλασιαστή Booth 4x4

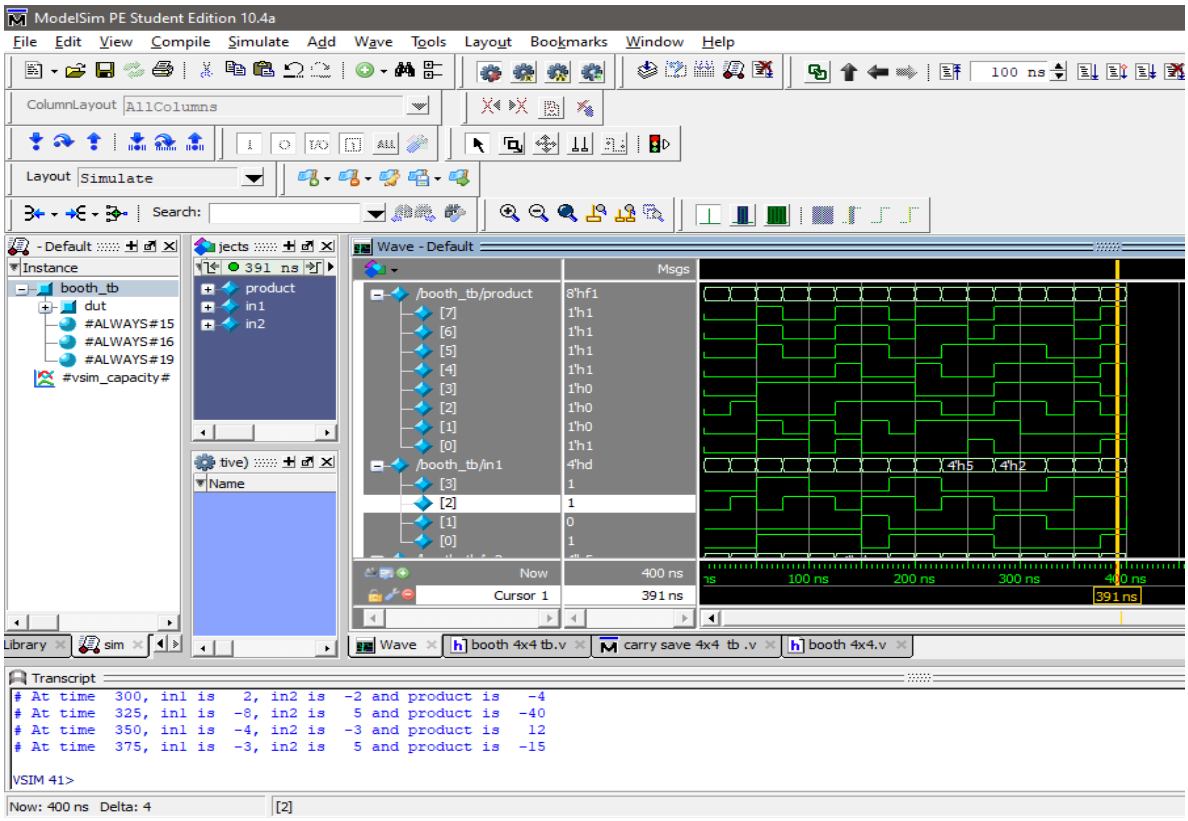
3.11.4 Προσομοίωση προσημασμένου πολλαπλασιαστή Booth 4x4 με την Verilog

Υπάρχουν ραγδαίες αλλαγές στο κύκλωμα σε σχέση με τα προηγούμενα κυκλώματα. Ωστόσο, η δομή του κώδικα ελέγχου δεν αλλάζει. Μοναδική αλλαγή που υπάρχει είναι ο χαρακτηρισμός των μεταβλητών ως προσημασμένες για να είναι δυνατός ο συμβολισμός και των αρνητικών αριθμών. Οι είσοδοι παραμένουν τέσσερα bit και η έξοδος οκτώ bit. Η αρχικοποίηση για τις εισόδους γίνεται στο μηδέν και ανά εικοσιπέντε νανοδευτερόλεπτα πραγματοποιούνται τυχαίες αλλαγές στις εισόδους. Η δομή του φαίνεται στο σχήμα 3.85. Ακολουθούν το testbench και η προσομοίωση.

```
module booth_tb(); //setting inputs and output
        wire signed [7:0] product;
        reg signed [3:0] in1, in2;
booth dut (.in1(in1), .in2(in2), .product(product)); // initialization
        initial begin
                in1 <= 0; in2 <= 0;
```

```
        end
        always #25 in1 <= $random; // every 25ns value of input is randomized
        always #25 in2 <= $random;
        always @(in1 or in2) // if any input changes the message appears
$monitor ("At time %4d, in1 is %d, in2 is %d and product is %d", $time, in1, in2, product);
endmodule
```

Σχήμα 3.90 Testbench προσημασμένου πολλαπλασιαστή Booth 4x4



Σχήμα 3.91 Προσομοίωση προσημασμένου πολλαπλασιαστή Booth 4x4

3.12 Περιγραφή προσημασμένου Booth πολλαπλασιαστή 8x8 με VHDL και Verilog

Επειδή ο κώδικας είναι υλοποίηση αλγορίθμου δεν θα χρειαστούν πολλές αλλαγές, όπως σε προηγούμενους πολλαπλασιαστές. Οι αλλαγές που υπάρχουν πέρα από το μέγεθος των μεταβλητών, είναι ο αριθμός των βημάτων και το μέγεθος των μετατοπίσεων.

3.12.1 Περιγραφή προσημασμένου Booth πολλαπλασιαστή 8x8 με VHDL

Ο παρακάτω κώδικας περιγράφει τον προσημασμένο Booth πολλαπλασιαστή 8x8 με VHDL.

```
LIBRARY ieee;

USE ieee.std_logic_1164.ALL;

USE ieee.std_logic_signed.ALL;


ENTITY booth8 IS

  PORT

  (

    A, B: IN  STD_LOGIC_VECTOR(7 DOWNTO 0);

     PQ: OUT STD_LOGIC_VECTOR(15 DOWNTO 0)

  );

END booth8;


ARCHITECTURE Behavioral OF booth8 IS




BEGIN

        PROCESS(A,B)

        VARIABLE NA, Y, P: STD_LOGIC_VECTOR(7 DOWNTO 0);

        VARIABLE b0: STD_LOGIC;

        BEGIN

        NA:=NOT(A)+1;

        Y:=B;

        P:="00000000";

        b0:='0';
```

```
        FOR i IN 0 TO 7 LOOP
                IF(Y(0)='0' AND b0='1') THEN
                        P:=P+A;
                ELSIF(Y(0)='1' AND b0='0') THEN
                        P:=P+NA;
                END IF;
                b0:=Y(0);
                Y(0):=Y(1);
                Y(1):=Y(2);
                Y(2):=Y(3);
                Y(3):=Y(4);
                Y(4):=Y(5);
                Y(5):=Y(6);
                Y(6):=Y(7);
                Y(7):=P(0);
                P(0):=P(1);
                P(1):=P(2);
                P(2):=P(3);
                P(3):=P(4);
                P(4):=P(5);
                P(5):=P(6);
                P(6):=P(7);
        END LOOP;
PQ(0)<=Y(0);
```

```
        PQ(1)<=Y(1);
        PQ(2)<=Y(2);
        PQ(3)<=Y(3);
        PQ(4)<=Y(4);
        PQ(5)<=Y(5);
        PQ(6)<=Y(6);
        PQ(7)<=Y(7);
        PQ(8)<=P(0);
        PQ(9)<=P(1);
        PQ(10)<=P(2);
        PQ(11)<=P(3);
        PQ(12)<=P(4);
        PQ(13)<=P(5);
        PQ(14)<=P(6);
        PQ(15)<=P(7);
        END PROCESS;
END ARCHITECTURE Behavioral;
```

Σχήμα 3.92 Υλοποίηση VHDL προσημασμένου πολλαπλασιαστή Booth 8x8.

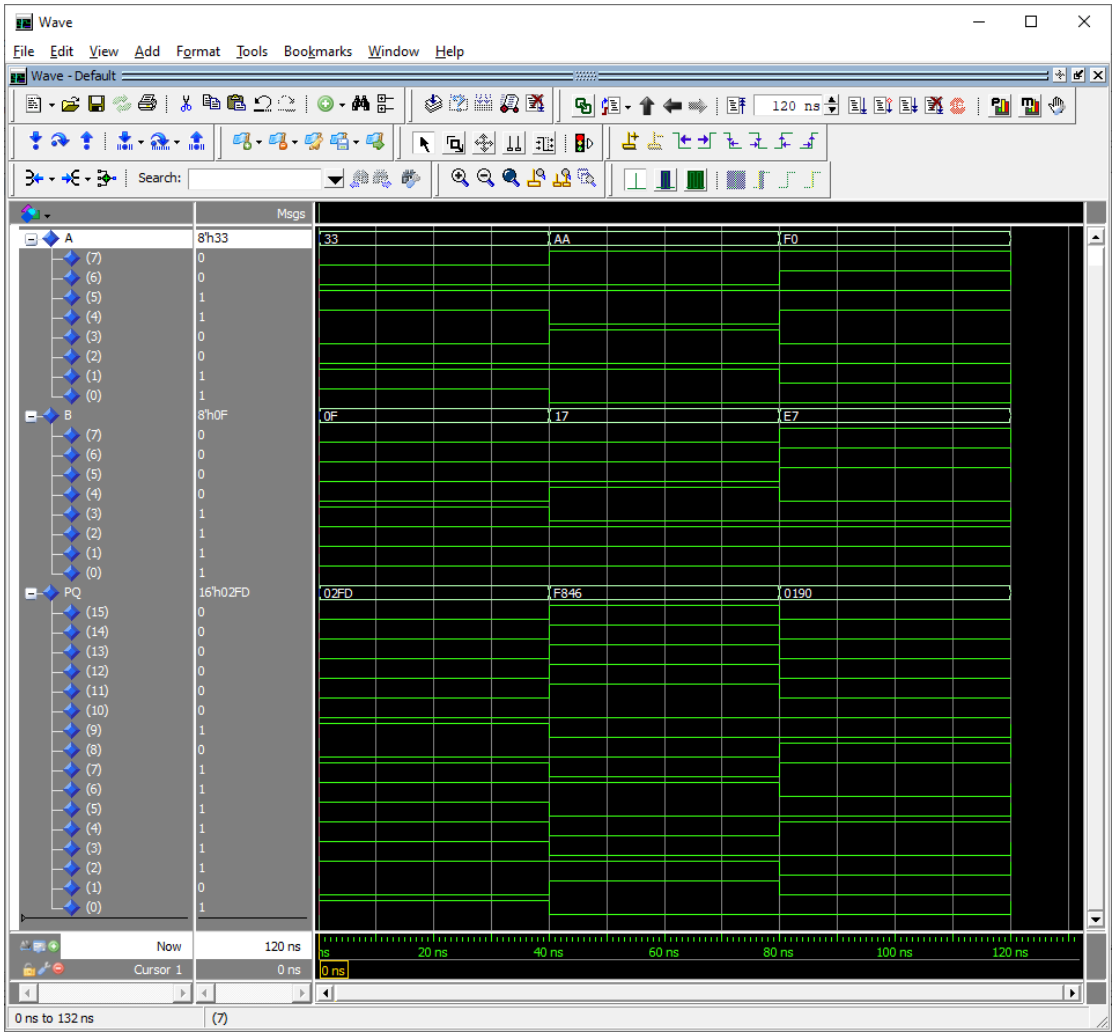
3.12.2 Προσομοίωση προσημασμένου πολλαπλασιαστή Booth 8x8 με την VHDL

Η υλοποίηση της προσομοίωσης του προσημασμένου πολλαπλασιαστή Booth 8x8 είναι αντίστοιχη με αυτή του Booth 4x4. Οι τιμές που έχουν εκχωρηθεί ανά 40 ns είναι «00110011» επί «00001111», «10101010» επί «00010111» και «11110000» επί «11100111». Τέλος, τα αποτελέσματα είναι ορατά στο παρακάτω διάγραμμα Waveform.

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

```
ENTITY booth8tb IS
END booth8tb;

ARCHITECTURE behavior OF booth8tb IS

SIGNAL A, B : STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL   PQ : STD_LOGIC_VECTOR(15 DOWNTO 0);

COMPONENT booth8
PORT ( A, B : IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
       PQ : OUT STD_LOGIC_VECTOR(15 DOWNTO 0));
END COMPONENT;

 BEGIN

m1: booth8 PORT MAP(A=>A,B=>B,PQ=>PQ);

PROCESS
BEGIN

 A<="00110011";B<="00001111";WAIT FOR 40 ns;
 A<="10101010";B<="00010111";WAIT FOR 40 ns;
 A<="11110000";B<="11100111";WAIT FOR 40 ns;

 END PROCESS;
END behavior;
```

Σχήμα 3.93 Testbench προσημασμένου πολλαπλασιαστή Booth 8x8 VHDL.



Σχήμα 3.94 Προσομοίωση προσημασμένου πολλαπλασιαστή Booth 8x8 VHDL.

3.12.3 Περιγραφή προσημασμένου Booth πολλαπλασιαστή 8x8 με Verilog

Ακολουθεί η υλοποίηση του πολλαπλασιαστή Booth 8x8 με χρήση Verilog.

```
module booth_8x8 (input signed [7:0] in1, in2, output reg signed [15:0] product);
        reg signed [1:0] val1;// last 2 bits
        reg val2;// last bit
        reg [7:0] in2_neg; // negative input
        integer i; // num of steps
        always @(in1 , in2) begin // initializations
                product = 15'd0; val2 = 1'd0;
```

```
        for (i = 0; i<8; i = i+1) begin // steps needed for completion
val1 = {in1[i], val2}; in2_neg = -in2;
        case (val1)// addition or subtraction
                2'd2 : product[15:8] = product[15:8] + in2_neg;
                2'd1 : product[15:8] = product[15:8] + in2;
        endcase
                product = product >>1;// shifting
                product[15] = product[14];
                val2 = in1[i]; end
        if (in2 == 8'd16) begin// negative check
                product = - product;
                end
        end
        endmodule
```

Σχήμα 3.95 Υλοποίηση Verilog προσημασμένου πολλαπλασιαστή Booth 8x8

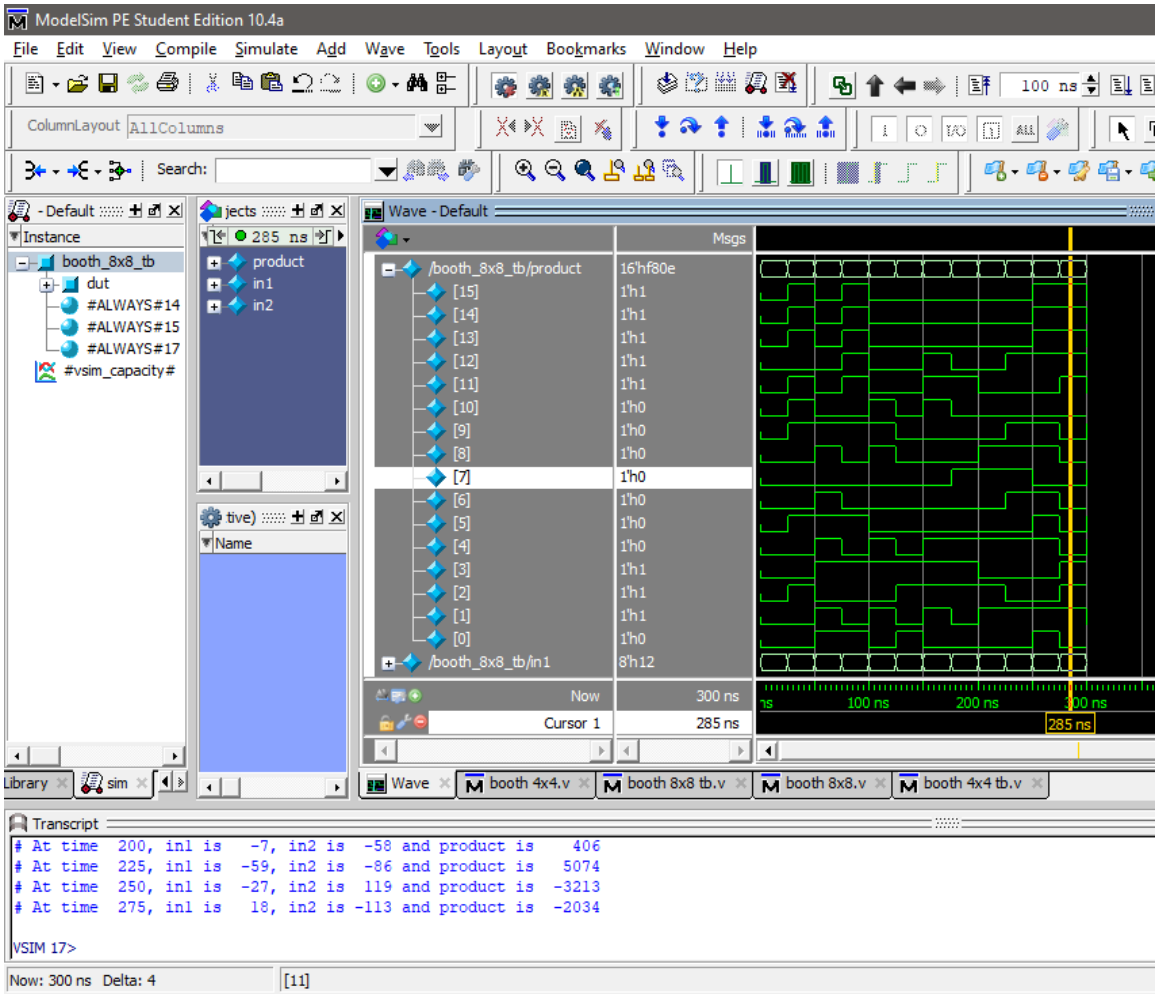
3.12.4 Προσομοίωση προσημασμένου πολλαπλασιαστή Booth 8x8 με την Verilog

Έχει την ίδια ακριβώς λειτουργικότητα με τον προσημασμένο πολλαπλασιαστή Booth 4x4 αλλά είναι δομημένος για εισόδους των οκτώ bit. Αντίστοιχα η έξοδος έχει μήκος δεκαέξι bit. Όπως και σε προηγούμενες προσομοιώσεις, οι είσοδοι αρχικοποιούνται με μηδέν και έχουν αλλαγές ανά εικοσιπέντε νανοδευτερόλεπτα. Η δομή του φαίνεται στο σχήμα 3.85. Ακολουθούν το testbench και η προσομοίωση.

```
module booth_8x8_tb(); //setting inputs and output
        wire signed [15:0] product;
        reg signed [7:0] in1, in2;
booth_8x8 dut (.in1(in1), .in2(in2), .product(product)); // initialization
        initial begin
```

```
            in1 <= 0;
            in2 <= 0;
        end
        always #25 in1 <= $random; // every 25ns value of input is ran1ndomised
        always #25 in2 <= $random;
        always @(in1 or in2) // if any input changes the message appears
$monitor ("At time %4d, in1 is %d, in2 is %d and product is %d", $time, in1, in2, product);
 endmodule
```

Σχήμα 3.96 Testbench προσημασμένου πολλαπλασιαστή Booth 8x8



Σχήμα 3.97 Προσομοίωση προσημασμένου πολλαπλασιαστή Booth 8x8

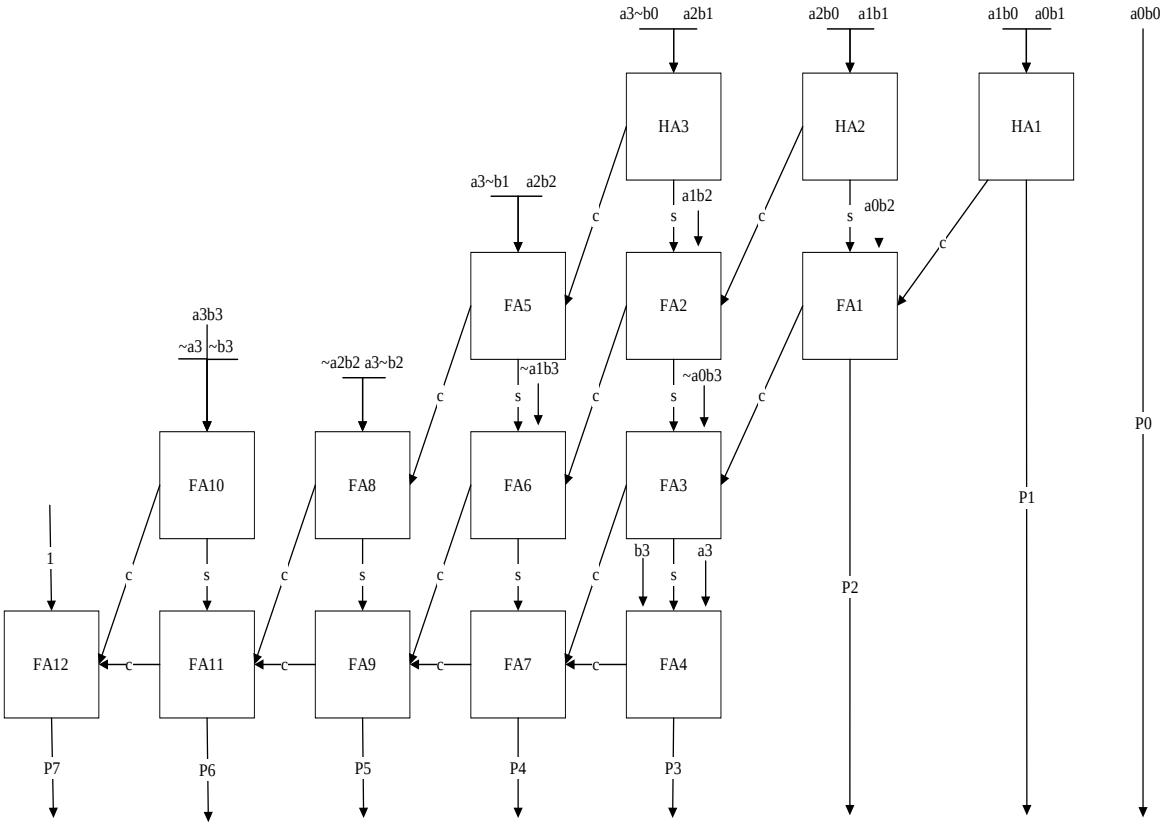
3.13 Περιγραφή πολλαπλασιαστών αρχιτεκτονικής Baugh Wooley με VHDL και Verilog

Οι πολλαπλασιαστές αρχιτεκτονικής Baugh Wooley αποτελούν μια αποδοτική αρχιτεκτονική πολλαπλασιαστή. Έχουν μικρότερες καθυστερήσεις σε σχέση με άλλους πολλαπλασιαστές και απαιτούν λιγότερη ενέργεια από πολλαπλασιαστές αρχιτεκτονικής Booth. Ο αλγόριθμος τους αποτελεί στην ουσία, μια τροποποιημένη μορφή της αρχιτεκτονικής Carry Save, η οποία χρησιμοποιεί συμπληρώματα του δύο για να κάνει και πράξεις προσημασμένων αριθμών.

```
                                    a3   a2   a1   a0
                                    b3   b2   b1   b0
                          ------------------------------------
                                a3~b0 a2b0 a1b0 a0b0
                           a3~b1 a2b1 a1b1 a0b1
                      a3~b2 a2b2 a1b2 a0b2
                 a3b3 ~a2b3 ~a1b3 ~a0b3
                      ~a3          a3
                  1   ~b3          b3
                ------------------------------------
                  p7   p6   p5   p4   p3   p2   p1   p0
```

Σχήμα 3.98 Πολλαπλασιασμός Baugh Wooley

Όπως φαίνεται παραπάνω ο αλγόριθμος δεν είναι πολύπλοκος αλλά έχει κάποιες μικρές διαφορές σε σχέση με μια τυπική Carry Save υλοποίηση. Ωστόσο θα αναλυθεί περαιτέρω με την υλοποίηση στις γλώσσες περιγραφής υλικού παρακάτω.



Σχήμα 3.99 Δομή προσημασμένου πολλαπλασιαστή Baugh Wooley 4x4

3.13.1 Περιγραφή προσημασμένου πολλαπλασιαστή Baugh Wooley 4x4 με VHDL

Ο παρακάτω κώδικας σε VHDL περιγράφει τον πολλαπλασιαστή Baugh Wooley 4x4.

```
LIBRARY ieee;

LIBRARY work;

USE ieee.std_logic_1164.all;

USE work.full_add_package.all;

USE work.half_add_package.all;


ENTITY BaughWooley4x4 IS

        PORT (                  a3, a2, a1, a0 : IN  STD_LOGIC;

                        b3, b2, b1, b0 : IN  STD_LOGIC;

          p7,p6,p5,p4,p3, p2, p1, p0 : OUT STD_LOGIC);

END BaughWooley4x4;


ARCHITECTURE structured OF BaughWooley4x4 IS

        SIGNAL a0b0: STD_LOGIC;

        SIGNAL a0b1,a1b0,c01,ap01: STD_LOGIC;

        SIGNAL a1b1,a2b0,c02,ap02,a0b2,c03,ap03: STD_LOGIC;

        SIGNAL a3Nb0,a1b2,a2b1,Na0b3,c04,ap04,c05,ap05,c06,ap06,c07,ap07:
STD_LOGIC;

        SIGNAL a3Nb1,a2b2,Na1b3,c08,ap08,c09,ap09,c10,ap10: STD_LOGIC;

        SIGNAL Na2b3,a3Nb2,c11,ap11,c12,ap12: STD_LOGIC;

        SIGNAL a3b3,Na3,Nb3,c13,ap13,c14,ap14: STD_LOGIC;

        SIGNAL c15,ap15: STD_LOGIC;


BEGIN
```

```
--Stage 1


a0b0<= a0 and b0;


a1b0<= a1 and b0;
a0b1<= a0 and b1;


a2b0<= a2 and b0;
a1b1<= a1 and b1;
a0b2<= a0 and b2;


a3Nb0<= a3 and (not b0);
a2b1<= a2 and b1;
a1b2<= a1 and b2;
Na0b3<= (not a0) and b3;


a3Nb1<= a3 and (not b1);
a2b2<= a2 and b2;
Na1b3<= (not a1) and b3;




Na2b3<= (not a2) and b3;
a3Nb2<= a3 and (not b2);


a3b3<= a3 and b3;
Na3<= not a3;
```

```
Nb3<= not b3;


--Stage 2

--P1

HA1:half_add PORT MAP(a1b0,a0b1,c01,ap01);


--P2

HA2:half_add PORT MAP(a1b1,a2b0,c02,ap02);

FA1:full_add PORT MAP(ap02,c01,a0b2,c03,ap03);


--P3

HA3:half_add PORT MAP(a3Nb0,a2b1,c04,ap04);

FA2:full_add PORT MAP(ap04,c02,a1b2,c05,ap05);

FA3:full_add PORT MAP(ap05,c03,Na0b3,c06,ap06);

FA4:full_add PORT MAP(ap06,b3,a3,c07,ap07);


--P4

FA5:full_add PORT MAP(a3Nb1,a2b2,c04,c08,ap08);

FA6:full_add PORT MAP(ap08,c05,Na1b3,c09,ap09);

FA7:full_add PORT MAP(ap09,c06,c07,c10,ap10);


--P5

FA8:full_add PORT MAP(Na2b3,a3Nb2,c08,c11,ap11);

FA9:full_add PORT MAP(ap11,c09,c10,c12,ap12);


--P6
```

```
FA10:full_add PORT MAP(a3b3,Na3,Nb3,c13,ap13);

FA11:full_add PORT MAP(ap13,c11,c12,c14,ap14);


--P7

FA12:full_add PORT MAP('1',c13,c14,c15,ap15);


--Final--


p7<=ap15;

p6<=ap14;

p5<=ap12;

p4<=ap10;

p3<=ap07;

p2<=ap03;

p1<=ap01;

p0<=a0b0;

END structured;
```

Σχήμα 3.100 Υλοποίηση VHDL προσημασμένου πολλαπλασιαστή Baugh Wooley 4x4.

3.13.2 Προσομοίωση προσημασμένου πολλαπλασιαστή Baugh Wooley 4x4 με VHDL

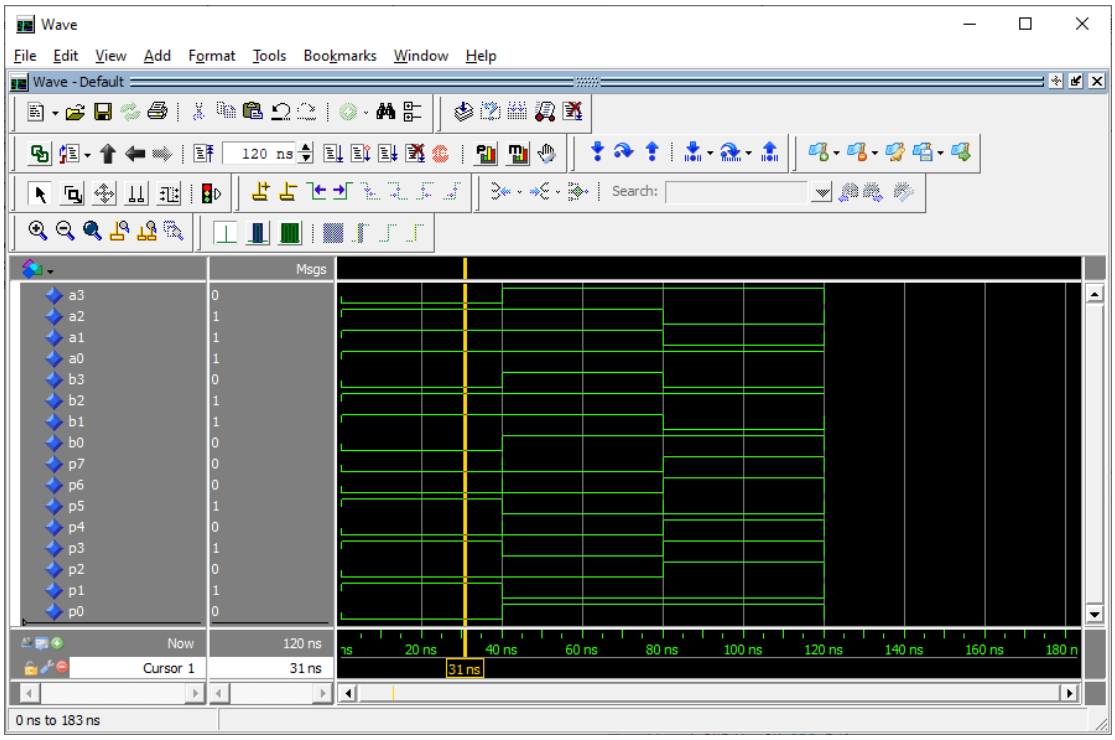
Ακολουθεί η προσομοίωση του πολλαπλασιαστή Baugh Wooley 4x4 VHDL στην οποία εκτελούνται πολλαπλασιασμοί ανά 40 νανοδευτερόλεπτα. Επιπλέον, τα αποτελέσματα της προσομοίωσης είναι εμφανείς στο παρακάτω Waveform.

```
LIBRARY IEEE;

USE IEEE.STD_LOGIC_1164.ALL;


ENTITY BaughWooley4x4tb IS
```

```
END BaughWooley4x4tb;

ARCHITECTURE behavior OF BaughWooley4x4tb IS

SIGNAL a3, a2, a1, a0: STD_LOGIC;
SIGNAL b3, b2, b1, b0: STD_LOGIC;
SIGNAL p7,p6,p5,p4,p3, p2, p1, p0: STD_LOGIC;

COMPONENT BaughWooley4x4
PORT (a3, a2, a1, a0: IN STD_LOGIC;
     b3, b2, b1, b0: IN STD_LOGIC;
     p7,p6,p5,p4,p3, p2, p1, p0: OUT STD_LOGIC);
END COMPONENT;

 BEGIN

m1: BaughWooley4x4 PORT MAP(a3=>a3,a2=>a2,a1=>a1,a0=>a0,
                            b3=>b3,b2=>b2,b1=>b1,b0=>b0,
p7=>p7,p6=>p6,p5=>p5,p4=>p4,p3=>p3,p2=>p2,p1=>p1,p0=>p0);

PROCESS
BEGIN

 a3<='0';a2<='1';a1<='1';a0<='1';b3<='0';b2<='1';b1<='1';b0<='0'; WAIT FOR 40 ns;
 a3<='1';a2<='1';a1<='1';a0<='1';b3<='1';b2<='1';b1<='1';b0<='1'; WAIT FOR 40 ns;
 a3<='1';a2<='0';a1<='0';a0<='1';b3<='0';b2<='1';b1<='0';b0<='1'; WAIT FOR 40 ns;
```

```
END PROCESS;

END behavior;
```

Σχήμα 3.101 Testbench προσημασμένου Baugh Wooley πολλαπλασιαστή 4x4 VHDL.



Σχήμα 3.102 Προσομοίωση προσημασμένου Baugh Wooley πολλαπλασιαστή 4x4 VHDL.

3.13.3 Περιγραφή προσημασμένου πολλαπλασιαστή Baugh Wooley 4x4 με Verilog

Ακολουθεί το κύκλωμα και η υλοποίηση του πολλαπλασιαστή Baugh Wooley 4x4 με Verilog.

```
module HA(output sum, carry, input a, b);

        assign sum = a^b;

        assign carry = (a&b);

endmodule

module FA(output sum, carry, input a, b, cin);

        assign sum =(a^b^cin);
```

```
        assign carry = ((a&b)|(a&cin)|(b&cin));
endmodule
module baugh_wooley (input signed [3:0] in1, in2, output signed [7:0] product);
// constant logic-one value
supply1 one;
wire [15:0] s, c;
assign product[0] = in1[0]&in2[0];
//row 1
HA HA1(product[1], c[0], (in1[1]&in2[0]), (in1[0]&in2[1]));
//row 2
HA HA2(s[1], c[1], (in1[2]&in2[0]), (in1[1]&in2[1]));
FA FA1(product[2], c[2], s[1], c[0], (in1[0]&in2[2]));
// row 3
HA HA3(s[3], c[3], (in1[3]&~in2[0]), (in1[2]&in2[1]));
FA FA2(s[4], c[4], s[3], c[1], (in1[1]&in2[2]));
FA FA3(s[5], c[5], s[4], c[2], (~in1[0]&in2[3]));
FA FA4(product[3], c[6], s[5], in1[3], in2[3]);
// row 4
FA FA5(s[7], c[7], (in1[3]&~in2[1]), c[3], (in1[2]&in2[2]));
FA FA6(s[8], c[8], s[7], c[4], (~in1[1]&in2[3]));
FA FA7(product[4], c[9], s[8], c[5], c[6]);
// row 5
FA FA8(s[10], c[10], (in1[3]&~in2[2]), c[7], (~in1[2]&in2[3]));
FA FA9(product[5], c[11], s[10], c[8], c[9]);
//row 6
FA FA10(s[12], c[12], ~in1[3], ~in2[3], (in1[3]&in2[3]));
```

```
FA FA11(product[6], c[13], s[12], c[10], c[11]);

//row 7

FA FA12(product[7], c[14], one, c[12], c[13]);

endmodule
```

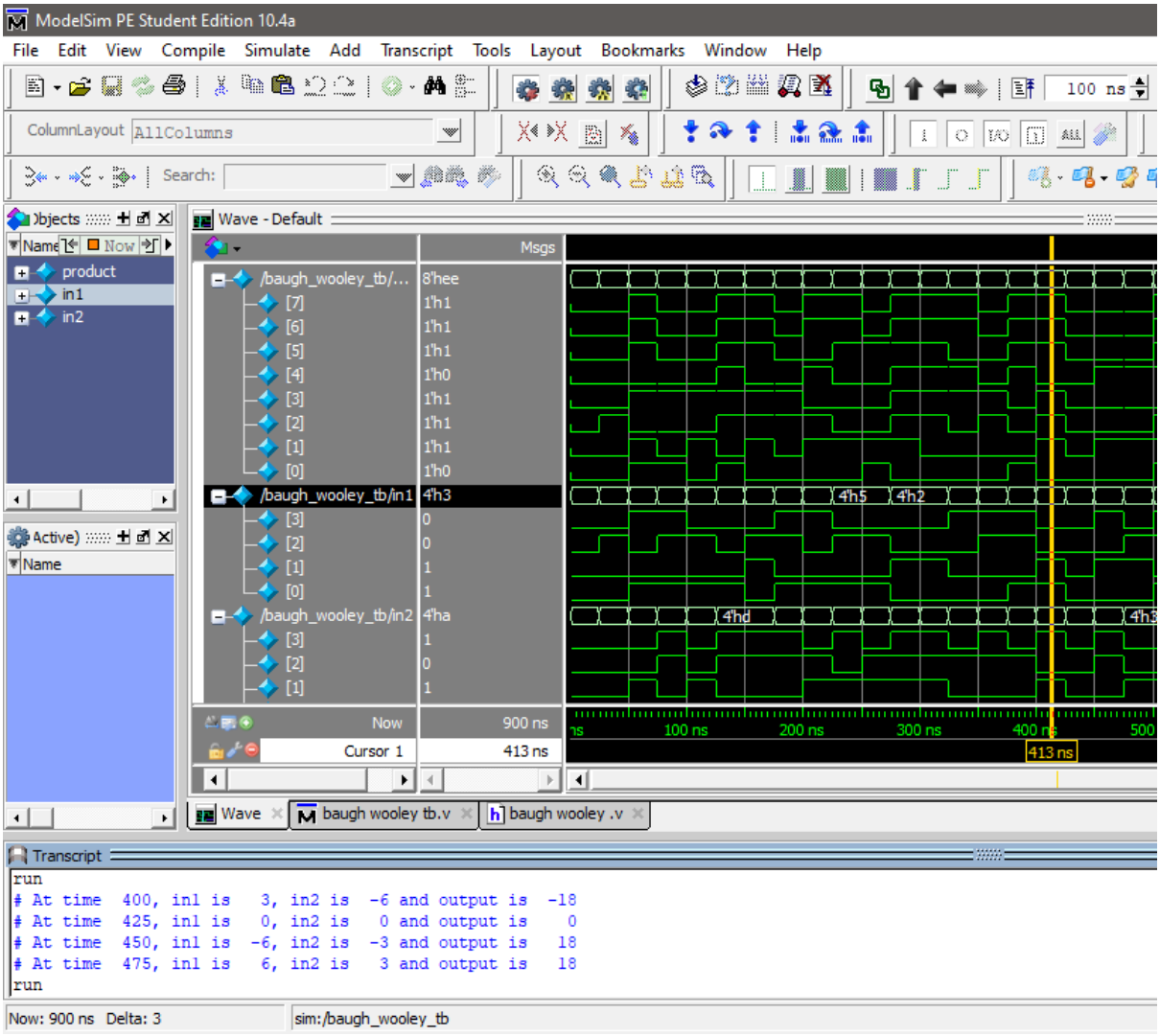
Σχήμα 3.103 Υλοποίηση Verilog προσημασμένου πολλαπλασιαστή Baugh Wooley 4x4

3.13.4 Προσομοίωση προσημασμένου πολλαπλασιαστή Baugh Wooley 4x4 με Verilog

Όπως και στον πολλαπλασιαστή αρχιτεκτονικής Carry Save 4x4, υπάρχουν δύο είσοδοι για πολλαπλασιαστή και πολλαπλασιαστέο μήκους τεσσάρων bit και μια έξοδος για το γινόμενο μήκους οκτώ bit. Όπως και σε προηγούμενες προσομοιώσεις, οι είσοδοι αρχικοποιούνται με μηδέν και έχουν αλλαγές ανά εικοσιπέντε νανοδευτερόλεπτα. Ακολουθούν το testbench και η προσομοίωση.

```
module baugh_wooley_tb(); //setting inputs in1nd output

        wire signed [7:0] product;

        reg signed [3:0] in1;

        reg signed [3:0] in2;

  baugh_wooley dut (.in1(in1), .in2(in2), .product(product)); // initialization

        initial begin

                in1 <= 0;

                in2 <= 0;

        end

        always #25 in1 <= $random; // every 25ns value of input is randomized

        always #25 in2 <= $random;

        always @(in1 or in2) // if any input changes the message appears

$monitor ("At time %4d, in1 is %d, in2 is %d and output is %d", $time, in1, in2, product);

  endmodule
```

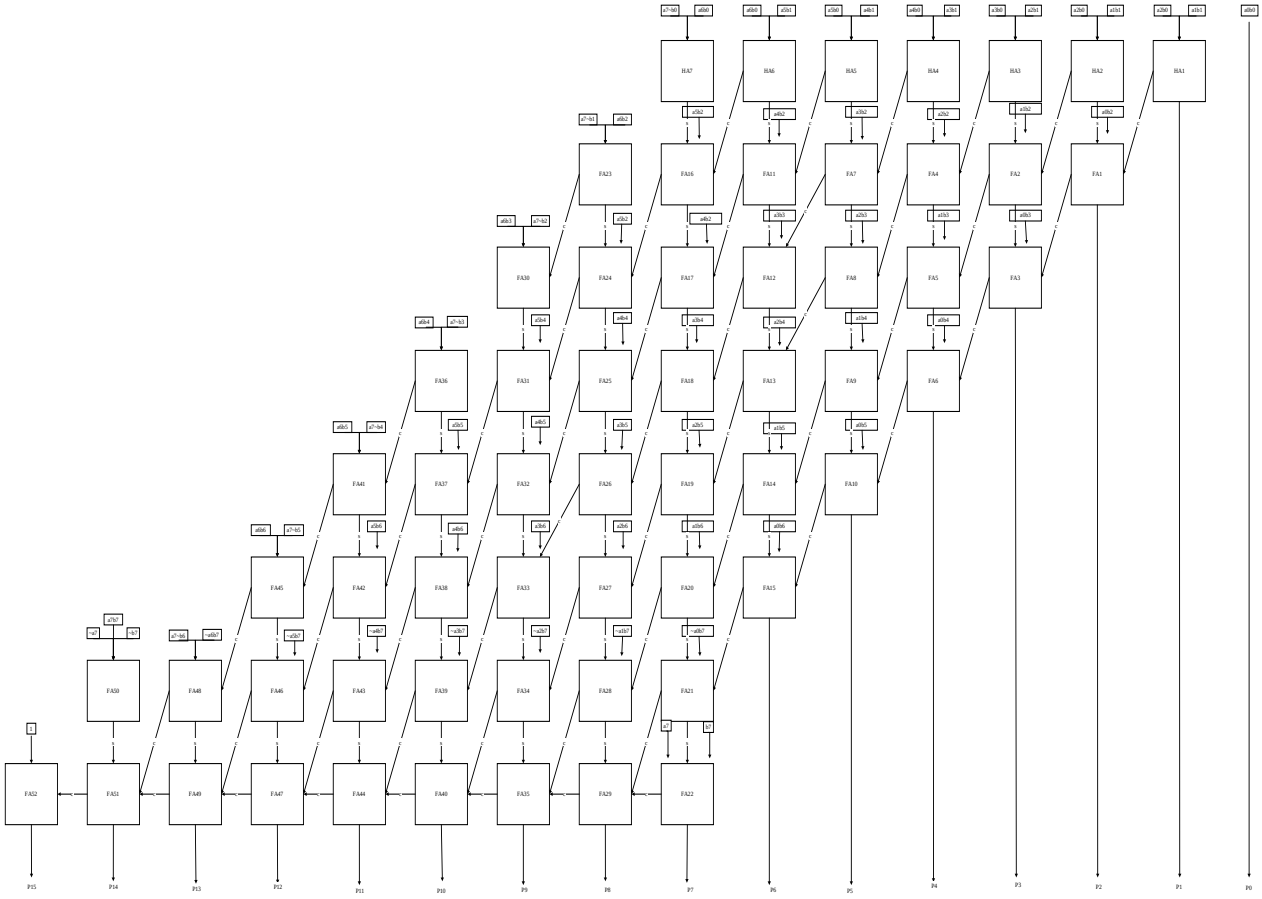
Σχήμα 3.104 Testbench προσημασμένου Baugh Wooley πολλαπλασιαστή 4x4



Σχήμα 3.105 Προσομοίωση προσημασμένου Baugh Wooley πολλαπλασιαστή 4x4

3.14 Περιγραφή προσημασμένου πολλαπλασιαστή Baugh Wooley 8x8 με VHDL και Verilog

Η λειτουργικότητα του πολλαπλασιαστή Baugh Wooley 8x8 είναι ίδια με αυτή του 4x4 άλλα λειτουργεί για τιμές μέχρι οκτώ bit. Ακολουθεί το κύκλωμα και η υλοποίηση του πολλαπλασιαστή Baugh Wooley 8x8 με Verilog.



Σχήμα 3.106 Δομή προσημασμένου πολλαπλασιαστή Baugh Wooley 8x8

3.14.1 Περιγραφή προσημασμένου πολλαπλασιαστή Baugh Wooley 8x8 με VHDL

Ακολουθεί η περιγραφή του προσημασμένου πολλαπλασιαστή Baugh Wooley 8x8 με VHDL κώδικα.

```
LIBRARY ieee;

LIBRARY work;

USE ieee.std_logic_1164.all;

USE work.full_add_package.all;

USE work.half_add_package.all;



ENTITY BaughWooley8x8 IS

        PORT (                                  a7, a6, a5, a4, a3, a2, a1, a0 :
IN  STD_LOGIC;
```

```
                                        b7, b6, b5, b4, b3, b2, b1, b0 : IN
STD_LOGIC;

          p15, p14, p13, p12, p11, p10, p9, p8, p7, p6, p5, p4, p3, p2, p1, p0 : OUT
STD_LOGIC);

END BaughWooley8x8;


ARCHITECTURE structured OF BaughWooley8x8 IS

        SIGNAL a1b0,a0b1,c1 : STD_LOGIC;

      SIGNAL a2b0,a1b1,a0b2,c2,c20,hp2 : STD_LOGIC;

      SIGNAL a3b0,a2b1,a1b2,a0b3,c3,c30,c31,hp3,fp30 : STD_LOGIC;

      SIGNAL a4b0,a3b1,a2b2,a1b3,a0b4,c4,c40,c41,c42,fp40,fp41,hp4 :
STD_LOGIC;

        SIGNAL
a5b0,a4b1,a3b2,a2b3,a1b4,a0b5,c5,c50,c51,c52,c53,fp50,fp51,fp52,hp5 :
STD_LOGIC;

        SIGNAL
a6b0,a5b1,a4b2,a3b3,a2b4,a1b5,a0b6,c6,c60,c61,c62,c63,c64,fp60,fp61,fp62,fp63,hp
6 : STD_LOGIC;

        SIGNAL
a7Nb0,a6b1,a5b2,a4b3,a3b4,a2b5,a1b6,Na0b7,c7,c70,c71,c72,c73,c74,c75,c76,fp70,f
p71,fp72,fp73,fp74,fp75,hp7 : STD_LOGIC;

        SIGNAL
a7Nb1,a6b2,a5b3,a4b4,a3b5,a2b6,Na1b7,c8,c80,c81,c82,c83,c84,c85,c86,fp80,fp81,f
p82,fp83,fp84,fp85 : STD_LOGIC;

        SIGNAL
a7Nb2,a6b3,a5b4,a4b5,a3b6,Na2b7,c90,c91,c92,c93,c94,c95,fp90,fp91,fp92,fp93,fp9
4 : STD_LOGIC;

        SIGNAL a7Nb3,a6b4,a5b5,a4b6,Na3b7,
c100,c101,c102,c103,c104,fp100,fp101,fp102,fp103 : STD_LOGIC;

        SIGNAL a7Nb4,a6b5,a5b6,Na4b7, c110,c111,c112,c113,fp110,fp111,fp112 :
STD_LOGIC;

        SIGNAL a7Nb5,a6b6,Na5b7,c120,c121,c122,fp120,fp121 : STD_LOGIC;
```

```
        SIGNAL a7Nb6,Na6b7,c130,c131,fp130 : STD_LOGIC;
        SIGNAL Na7,Nb7,a7b7,c140,c141,c150,fp140 : STD_LOGIC;



BEGIN
--STAGE 1
a1b0<=a1 AND b0;
a0b1<=a0 AND b1;

a2b0<=a2 AND b0;
a1b1<=a1 AND b1;
a0b2<=a0 AND b2;

a3b0<=a3 AND b0;
a1b2<=a1 AND b2;
a2b1<=a2 AND b1;
a0b3<=a0 AND b3;

a4b0<=a4 AND b0;
a3b1<=a3 AND b1;
a2b2<=a2 AND b2;
a1b3<=a1 AND b3;
a0b4<=a0 AND b4;

a5b0<=a5 AND b0;
a4b1<=a4 AND b1;
```

```
a3b2<=a3 AND b2;
a2b3<=a2 AND b3;
a1b4<=a1 AND b4;
a0b5<=a0 AND b5;

a6b0<=a6 AND b0;
a5b1<=a5 AND b1;
a4b2<=a4 AND b2;
a3b3<=a3 AND b3;
a2b4<=a2 AND b4;
a1b5<=a1 AND b5;
a0b6<=a0 AND b6;

a7Nb0<=a7 AND NOT b0;
a6b1<=a6 AND b1;
a5b2<=a5 AND b2;
a4b3<=a4 AND b3;
a3b4<=a3 AND b4;
a2b5<=a2 AND b5;
a1b6<=a1 AND b6;
Na0b7<=NOT a0 AND b7;

a7Nb1<=a7 AND NOT b1;
a6b2<=a6 AND b2;
a5b3<=a5 AND b3;
a4b4<=a4 AND b4;
```

```
a3b5<=a3 AND b5;

a2b6<=a2 AND b6;

Na1b7<=NOT a1 AND b7;


a7Nb2<=a7 AND NOT b2;

a6b3<=a6 AND b3;

a5b4<=a5 AND b4;

a4b5<=a4 AND b5;

a3b6<=a3 AND b6;

Na2b7<=NOT a2 AND b7;


a7Nb3<=a7 AND NOT b3;

a6b4<=a6 AND b4;

a5b5<=a5 AND b5;

a4b6<=a4 AND b6;

Na3b7<=NOT a3 AND b7;


a7Nb4<=a7 AND NOT b4;

a6b5<=a6 AND b5;

a5b6<=a5 AND b6;

Na4b7<=NOT a4 AND b7;


a7Nb4<=a7 AND NOT b4;

a6b5<=a6 AND b5;

a5b6<=a5 AND b6;

Na4b7<=NOT a4 AND b7;
```

```
a7Nb5<=a7 AND NOT b5;
a6b6<=a6 AND b6;
Na5b7<=NOT a5 AND b7;

a7Nb6<=a7 AND NOT b6;
Na6b7<=NOT a6 AND b7;

a7b7<=a7 AND b7;
Na7<=NOT a7;
Nb7<=NOT b7;

--STAGE 2
--P0:
  p0<=a0 AND b0;

--P1:
  HA1:half_add PORT MAP (a1b0, a0b1, c1, p1);

--P2:
  HA2: half_add PORT MAP (a2b0, a1b1, c2, hp2);
  FA2: full_add PORT MAP (a0b2,hp2, c1, c20, p2);

--P3:
  HA3: half_add PORT MAP (a3b0, a2b1, c3, hp3);
  FA30: full_add PORT MAP (a1b2, hp3, c2, c30, fp30);
```

```
  FA31: full_add PORT MAP (a0b3, fp30, c20, c31, p3);


--P4:
  HA4:  half_add PORT MAP (a4b0, a3b1, c4, hp4);
  FA40: full_add PORT MAP (hp4, a2b2, c3, c40, fp40);
  FA41: full_add PORT MAP (fp40, a1b3, c30, c41, fp41);
  FA42: full_add PORT MAP (fp41, a0b4, c31, c42, p4);


--P5:
  HA5:  half_add PORT MAP (a5b0, a4b1, c5, hp5);
  FA50: full_add PORT MAP (a3b2, hp5, c4, c50, fp50);
  FA51: full_add PORT MAP (a2b3, fp50, c40, c51, fp51);
  FA52: full_add PORT MAP (a1b4, fp51, c41, c52, fp52);
  FA53: full_add PORT MAP (a0b5, fp52, c42, c53, p5);


--P6:
  HA6:  half_add PORT MAP (a6b0, a5b1, c6, hp6);
  FA60: full_add PORT MAP (a4b2, hp6, c5, c60, fp60);
  FA61: full_add PORT MAP (a3b3, fp60, c50, c61, fp61);
  FA62: full_add PORT MAP (a2b4, fp61, c51, c62, fp62);
  FA63: full_add PORT MAP (a1b5, fp62, c52, c63, fp63);
  FA64: full_add PORT MAP (a0b6, fp63, c53, c64, p6);


--P7:
  HA7:  half_add PORT MAP (a7Nb0, a6b1, c7, hp7);
  FA70: full_add PORT MAP (a5b2, hp7, c6, c70, fp70);
```

```
  FA71: full_add PORT MAP (a4b3, fp70, c60, c71, fp71);
  FA72: full_add PORT MAP (a3b4, fp71, c61, c72, fp72);
  FA73: full_add PORT MAP (a2b5, fp72, c62, c73, fp73);
  FA74: full_add PORT MAP (a1b6, fp73, c63, c74, fp74);
  FA75: full_add PORT MAP (Na0b7, fp74, c64, c75, fp75);
  FA76: full_add PORT MAP (a7, b7, fp75, c76, p7);

--P8:
  FA80: full_add PORT MAP (a7Nb1, a6b2, c7, c80, fp80);
  FA81: full_add PORT MAP (fp80, a5b3, c70, c81, fp81);
  FA82: full_add PORT MAP (fp81, a4b4, c71, c82, fp82);
  FA83: full_add PORT MAP (fp82, a3b5, c72, c83, fp83);
  FA84: full_add PORT MAP (fp83, a2b6, c73, c84, fp84);
  FA85: full_add PORT MAP (fp84, Na1b7, c74, c85, fp85);
  FA86: full_add PORT MAP (fp85, c75, c76, c86, p8);

--P9:
  FA90: full_add PORT MAP (a7Nb2, a6b3, c80, c90, fp90);
  FA91: full_add PORT MAP (fp90, a5b4, c81, c91, fp91);
  FA92: full_add PORT MAP (fp91, a4b5, c82, c92, fp92);
  FA93: full_add PORT MAP (fp92, a3b6, c83, c93, fp93);
  FA94: full_add PORT MAP (fp93, Na2b7, c84, c94, fp94);
  FA95: full_add PORT MAP (fp94, c85, c86, c95, p9);

--P10:
  FA100: full_add PORT MAP (a7Nb3, a6b4, c90, c100, fp100);
```

```
  FA101: full_add PORT MAP (fp100, a5b5, c91, c101, fp101);
  FA102: full_add PORT MAP (fp101, a4b6, c92, c102, fp102);
  FA103: full_add PORT MAP (fp102, Na3b7, c93, c103, fp103);
  FA104: full_add PORT MAP (fp103, c94, c95, c104, p10);

--P11:
  FA110: full_add PORT MAP (a7Nb4, a6b5, c100, c110, fp110);
  FA111: full_add PORT MAP (fp110, a5b6, c101, c111, fp111);
  FA112: full_add PORT MAP (fp111, Na4b7, c102, c112, fp112);
  FA113: full_add PORT MAP (fp112, c103, c104, c113, p11);

--P12:
  FA120: full_add PORT MAP (a7Nb5, a6b6, c110, c120, fp120);
  FA121: full_add PORT MAP (fp120, Na5b7, c111, c121, fp121);
  FA122: full_add PORT MAP (fp121, c112, c113, c122, p12);

--P13:
  FA130: full_add PORT MAP (a7Nb6, Na6b7, c120, c130, fp130);
  FA131: full_add PORT MAP (fp130, c121, c122, c131, p13);

--P14:
  FA140: full_add PORT MAP (a7b7, Na7, Nb7, c140, fp140);
  FA141: full_add PORT MAP (fp140, c130, c131, c141, p14);

--p15
  FA150: full_add PORT MAP ('1', c140, c141, c150, p15);
```

```
END structured;
```

Σχήμα 3.107 Υλοποίηση VHDL προσημασμένου πολλαπλασιαστή Baugh Wooley 8x8.

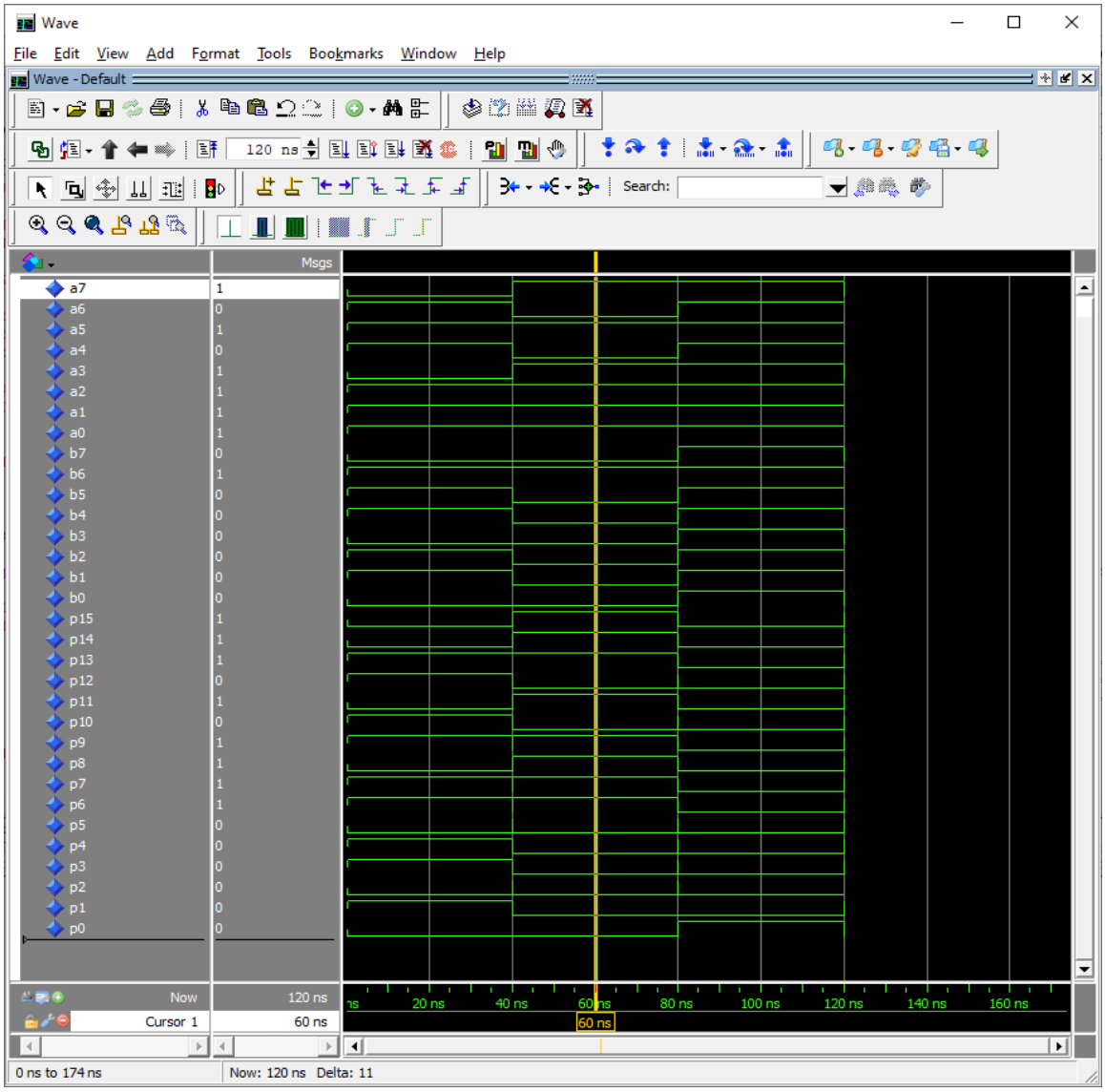
3.14.2 Προσομοίωση προσημασμένου πολλαπλασιαστή Baugh Wooley 8x8 με VHDL

Ο ακόλουθος κώδικας VHDL είναι ένα Testbench που προσομοιώνει τον πολλαπλασιαστή Baugh Wooley 8x8. Κάθε 40 ns εκτελείται και μια νέα πράξη της οποίας τα αποτελέσματα είναι εμφανείς στο ακόλουθο διάγραμμα Waveform.

```
LIBRARY IEEE;

USE IEEE.STD_LOGIC_1164.ALL;


ENTITY BaughWooley8x8tb IS

END BaughWooley8x8tb;


ARCHITECTURE behavior OF BaughWooley8x8tb IS


SIGNAL  a7, a6, a5, a4, a3, a2, a1, a0: STD_LOGIC;

SIGNAL  b7, b6, b5, b4, b3, b2, b1, b0: STD_LOGIC;

SIGNAL  p15, p14, p13, p12, p11, p10, p9, p8, p7, p6, p5, p4, p3, p2, p1, p0:
STD_LOGIC;


COMPONENT BaughWooley8x8

PORT (a7, a6, a5, a4, a3, a2, a1, a0: IN STD_LOGIC;

     b7, b6, b5, b4, b3, b2, b1, b0: IN STD_LOGIC;

     p15, p14, p13, p12, p11, p10, p9, p8, p7, p6, p5, p4, p3, p2, p1, p0: OUT
STD_LOGIC);

END COMPONENT;


 BEGIN
```

```
m1: BaughWooley8x8 PORT
MAP(a7=>a7,a6=>a6,a5=>a5,a4=>a4,a3=>a3,a2=>a2,a1=>a1,a0=>a0,

                    b7=>b7,b6=>b6,b5=>b5,b4=>b4,b3=>b3,b2=>b2,b1=>b1,b0=>b0,

p15=>p15,p14=>p14,p13=>p13,p12=>p12,p11=>p11,p10=>p10,p9=>p9,p8=>p8,p7=
>p7,p6=>p6,p5=>p5,p4=>p4,p3=>p3,p2=>p2,p1=>p1,p0=>p0);



PROCESS

BEGIN




a7<='0';a6<='1';a5<='1';a4<='1';a3<='0';a2<='1';a1<='1';a0<='1';b7<='0';b6<='1';b5<='
1';b4<='1';b3<='0';b2<='1';b1<='1';b0<='0'; WAIT FOR 40 ns;


a7<='1';a6<='0';a5<='1';a4<='0';a3<='1';a2<='1';a1<='1';a0<='1';b7<='0';b6<='1';b5<='
0';b4<='0';b3<='0';b2<='0';b1<='0';b0<='0'; WAIT FOR 40 ns;


a7<='1';a6<='1';a5<='1';a4<='1';a3<='1';a2<='1';a1<='1';a0<='1';b7<='1';b6<='1';b5<='
1';b4<='1';b3<='1';b2<='1';b1<='1';b0<='1'; WAIT FOR 40 ns;



 END PROCESS;

END behavior;
```

Σχήμα 3.108 Testbench προσημασμένου Baugh Wooley πολλαπλασιαστή 8x8 VHDL.



Σχήμα 3.109 Προσομοίωση προσημασμένου Baugh Wooley πολλαπλασιαστή 8x8 VHDL.

3.14.3 Περιγραφή προσημασμένου πολλαπλασιαστή Baugh Wooley 8x8 με Verilog

Ακολουθεί το κύκλωμα και η υλοποίηση του πολλαπλασιαστή Baugh Wooley 8x8 με Verilog.

```
module HA(output sum, carry, input a, b);

    assign sum = a^b;

    assign carry = (a&b);

endmodule
```

```
   module FA(output sum, carry, input a, b, cin);
        assign sum =(a^b^cin);
        assign carry = ((a&b)|(a&cin)|(b&cin));
   endmodule
   module baugh_wooley_8x8(input signed [7:0] in1, in2, output signed [15:0]
   product);
 // constant logic-one value
 supply1 one;
 wire [63:0] s, c;
   assign product[0] = in1[0]&in2[0];

//row 1
HA HA1(product[1], c[0], (in1[1]&in2[0]), (in1[0]&in2[1]));

//row 2
HA HA2(s[1], c[1], (in1[2]&in2[0]), (in1[1]&in2[1]));
FA FA1(product[2], c[2],(in1[0]&in2[2]), s[1], c[0]);

//row 3
HA HA3(s[3], c[3], (in1[3]&in2[0]), (in1[2]&in2[1]));
FA FA2(s[4], c[4], (in1[1]&in2[2]), s[3], c[1]);
FA FA3(product[3], c[5],(in1[0]&in2[3]), s[4], c[2]);

//row 4
HA HA4(s[6], c[6], (in1[4]&in2[0]), (in1[3]&in2[1]));
FA FA4(s[7], c[7], (in1[2]&in2[2]), s[6], c[3]);
FA FA5(s[8], c[8], (in1[1]&in2[3]), s[7], c[4]);
```

```
FA FA6(product[4], c[9], (in1[0]&in2[4]), s[8], c[5]);


//row 5
HA HA5(s[10], c[10], (in1[5]&in2[0]), (in1[4]&in2[1]));
FA FA7(s[11], c[11], (in1[3]&in2[2]), s[10], c[6]);
FA FA8(s[12], c[12], (in1[2]&in2[3]), s[11], c[7]);
FA FA9(s[13], c[13], (in1[1]&in2[4]), s[12], c[8]);
FA FA10(product[5], c[14], (in1[0]&in2[5]), s[13], c[9]);


//row 6
HA HA6(s[15], c[15], (in1[6]&in2[0]), (in1[5]&in2[1]));
FA FA11(s[16], c[16], (in1[4]&in2[2]), s[15], c[10]);
FA FA12(s[17], c[17], (in1[3]&in2[3]), s[16], c[11]);
FA FA13(s[18], c[18], (in1[2]&in2[4]), s[17], c[12]);
FA FA14(s[19], c[19], (in1[1]&in2[5]), s[18], c[13]);
FA FA15(product[6], c[20], (in1[0]&in2[6]), s[19], c[14]);


//row 7
HA HA7(s[21], c[21], (in1[7]&~in2[0]), (in1[6]&in2[1]));
FA FA16(s[22], c[22], (in1[5]&in2[2]), s[21], c[15]);
FA FA17(s[23], c[23], (in1[4]&in2[3]), s[22], c[16]);
FA FA18(s[24], c[24], (in1[3]&in2[4]), s[23], c[17]);
FA FA19(s[25], c[25], (in1[2]&in2[5]), s[24], c[18]);
FA FA20(s[26], c[26], (in1[1]&in2[6]), s[25], c[19]);
FA FA21(s[27], c[27], (~in1[0]&in2[7]), s[26], c[20]);
FA FA22(product[7], c[28], in1[7], in2[7], s[27]);
```

```
//row 8
FA FA23(s[29], c[29], (in1[7]&~in2[1]), (in1[6]&in2[2]), c[21]);
FA FA24(s[30], c[30], (in1[5]&in2[3]), s[29], c[22]);
FA FA25(s[31], c[31], (in1[4]&in2[4]), s[30], c[23]);
FA FA26(s[32], c[32], (in1[3]&in2[5]), s[31], c[24]);
FA FA27(s[33], c[33], (in1[2]&in2[6]), s[32], c[25]);
FA FA28(s[34], c[34], (~in1[1]&in2[7]), s[33], c[26]);
FA FA29(product[8], c[35], s[34], c[27], c[28]);

//row 9
FA FA30(s[36], c[36], (in1[7]&~in2[2]), (in1[6]&in2[3]), c[29]);
FA FA31(s[37], c[37], (in1[5]&in2[4]), s[36], c[30]);
FA FA32(s[38], c[38], (in1[4]&in2[5]), s[37], c[31]);
FA FA33(s[39], c[39], (in1[3]&in2[6]), s[38], c[32]);
FA FA34(s[40], c[40], (~in1[2]&in2[7]), s[39], c[33]);
FA FA35(product[9], c[41], s[40], c[35], c[34]);
//row 10
FA FA36(s[42], c[42], (in1[7]&~in2[3]), (in1[6]&in2[4]), c[36]);
FA FA37(s[43], c[43], (in1[5]&in2[5]), s[42], c[37]);
FA FA38(s[44], c[44], (in1[4]&in2[6]), s[43], c[38]);
FA FA39(s[45], c[45], (~in1[3]&in2[7]), s[44], c[39]);
FA FA40(product[10], c[46], s[45], c[41], c[40]);
//row 11
FA FA41(s[47], c[47], (in1[7]&~in2[4]), (in1[6]&in2[5]), c[42]);
FA FA42(s[48], c[48], (in1[5]&in2[6]), s[47], c[43]);
```

```
FA FA43(s[49], c[49], (~in1[4]&in2[7]), s[48], c[44]);
FA FA44(product[11], c[50], s[49], c[45], c[46]);
//row 12
FA FA45(s[51], c[51], (in1[7]&~in2[5]), (in1[6]&in2[6]), c[47]);
FA FA46(s[52], c[52], (~in1[5]&in2[7]), s[51], c[48]);
FA FA47(product[12], c[53], s[52], c[49], c[50]);
//row 13
FA FA48(s[54], c[54], (in1[7]&~in2[6]), (~in1[6]&in2[7]), c[51]);
FA FA49(product[13], c[55], s[54], c[52], c[53]);
//row 14
FA FA50(s[56], c[56], ~in1[7], ~in2[7], (in1[7]&in2[7]));
FA FA51(product[14], c[57], s[56], c[54], c[55]);
//row 15
FA FA52(product[15], c[58], one, c[57], c[56]);
 endmodule
```

Σχήμα 3.110 Υλοποίηση Verilog προσημασμένου πολλαπλασιαστή Baugh Wooley 8x8

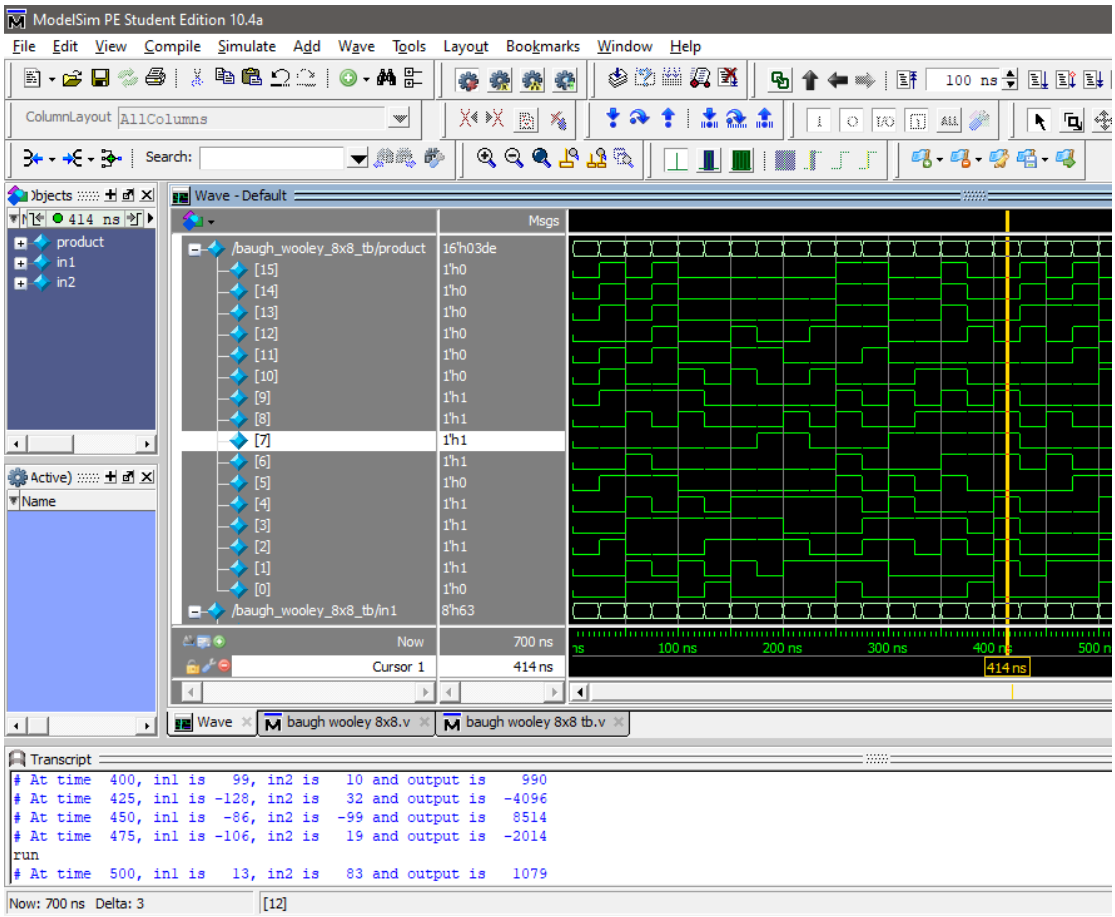
3.14.4 Προσομοίωση προσημασμένου πολλαπλασιαστή Baugh Wooley 8x8 με Verilog

Όπως και στον Carry Save 8x8, υπάρχουν δύο είσοδοι για πολλαπλασιαστή και πολλαπλασιαστέο μήκους οκτώ bit και μια έξοδος για το γινόμενο μήκους δεκαέξι bit. Οι είσοδοι αρχικοποιούνται με μηδέν και έχουν αλλαγές ανά εικοσιπέντε νανοδευτερόλεπτα. Ακολουθούν το testbench και η προσομοίωση.

```
module baugh_wooley_8x8_tb(); //setting inputs in1nd output
    wire signed [15:0] product;
    reg signed [7:0] in1;
    reg signed [7:0] in2;
```

```
baugh_wooley_8x8 dut (.in1(in1), .in2(in2), .product(product)); // initialization
        initial begin
                in1 <= 0;
                in2 <= 0;
        end
        always #25 in1 <= $random; // every 25ns value of input is randomized
        always #25 in2 <= $random;
        always @(in1 or in2) // if any input changes the message appears
$monitor ("At time %4d, in1 is %d, in2 is %d and output is %d", $time, in1, in2, product);
 endmodule
```

Σχήμα 3.111 Testbench προσημασμένου Baugh Wooley πολλαπλασιαστή 8x8



Σχήμα 3.112 Προσομοίωση προσημασμένου Baugh Wooley πολλαπλασιαστή 8x8

4. Συγκρίσεις και παρατηρήσεις μεταξύ γλωσσών και ανάλυση συμπερασμάτων

4.1 Συγκρίσεις μεταξύ κώδικα VHDL και Verilog

Σε αυτό το κεφάλαιο θα πραγματοποιηθούν συγκρίσεις μεταξύ των γλωσσών VHDL και Verilog, θα αναλυθούν ομοιότητες και διαφορές του κώδικα των δύο γλωσσών που έχει γραφτεί, θα γίνουν ορισμένες παρατηρήσεις και θα εξαχθούν συμπεράσματα σχετικά με αυτές. Στην συνέχεια θα πραγματοποιηθούν συγκρίσεις στο παράδειγμα το οποίο υλοποιεί την λογική συνάρτηση $f = x_1 x_2 + \bar{x}_2 x_3$.

```
LIBRARY IEEE;

USE IEEE.STD_LOGIC_1164.ALL;

ENTITY example1 IS
PORT (x1, x2, x3 : IN  STD_LOGIC;
        f : OUT STD_LOGIC);

END example1;

ARCHITECTURE LogicFunc OF example1 IS

  BEGIN

    f <=(x1 AND x2) OR ((NOT x2) AND x3);

END LogicFunc;
```

```
module example (x1, x2, x3, f);

input x1,x2,x3;

output f;
assign f= (x1 & x2) | (~x2 & x3);

endmodule
```

Σχήμα 4.1 Σύγκριση του $f = x_1 x_2 + \bar{x}_2 x_3$, για την γενική δομή του κώδικα μεταξύ VHDL και Verilog

Αρχικά είναι εμφανές ότι ο κώδικας της VHDL είναι πολύ μεγαλύτερος από τον κώδικα της Verilog ενώ οι δύο παραπάνω κώδικες είναι ισοδύναμοι. Αυτό συμβαίνει εξαιτίας της δομής του κώδικα της VHDL με την ξεχωριστή δήλωση οντότητας, αρχιτεκτονικής και βιβλιοθηκών. Έπειτα παρατηρούνται διαφορές και στους τύπους μεταβλητών καθώς, ενώ στην VHDL πρέπει να υπάρχει και τύπος μεταβλητής π.χ. «STD_LOGIC» πέραν της δήλωσης εισόδου και εξόδου ενώ στην Verilog δεν είναι απαραίτητο. Τέλος υπάρχει μια διαφορά μεταξύ των τελεστών στις λογικές πράξεις. Και στις δύο γλώσσες μπορούν να χρησιμοποιηθούν οι τελεστές «AND» και «OR» αλλά η VHDL δεν υποστηρίζει σύμβολα ως λογικούς τελεστές.

Ενώ η παραπάνω σύγκριση έδειξε διαφορές ανάμεσα στις δύο γλώσσες, η παρακάτω θα δώσει έμφαση ομοιότητες και κοινά σημεία. Οι συγκρίσεις θα συνεχίσουν με τον πολυπλέκτη 2 σε 1.

```
LIBRARY ieee;                                   module mux_2to1(input x0, x1, s, output z);
USE ieee.std_logic_1164.all;                          assign z = (x0 & ~s) | (x1 & s);
ENTITY Mux2to1 IS                                                          endmodule
PORT (x0, x1, s : IN  STD_LOGIC;
            z : OUT STD_LOGIC);

  END Mux2to1;

  ARCHITECTURE LogicFunc OF Mux2to1 IS

    BEGIN

       z<= (x0 AND (NOT s)) OR (x1 AND s);

  END LogicFunc;
```

Σχήμα 4.2 Σύγκριση πολυπλέκτη 2 σε 1 για την γενική δομή του κώδικα μεταξύ VHDL και Verilog

Κατά τα γνωστά, η VHDL χωρίζεται σε δύο τμήματα κώδικα, την οντότητα και την αρχιτεκτονική όπου, στην πρώτη ορίζονται είσοδοι και έξοδοι και στην δεύτερη περιγράφεται η συμπεριφορά του κυκλώματος. Η Verilog εμφανώς εμπνευσμένη από την VHDL χρησιμοποιεί ένα παρόμοιο σύστημα στο οποίο δηλώνονται οι είσοδοι και έξοδοι πρώτα, και έπειτα περιγράφεται η συμπεριφορά του κυκλώματος. Όπως φαίνεται και στους παραπάνω κώδικες, το τμήμα του κώδικα VHDL που βρίσκεται στο «PORT» και το τμήμα του κώδικα Verilog που βρίσκεται στο «module» είναι ίδια με ορισμένες διαφορές, όπως οι τύποι μεταβλητών. Επιπλέον, πρέπει να γίνει αναφορά και στις δηλώσεις πολλαπλών bit καθώς με εξαίρεση το πώς περιγράφεται από κάθε γλώσσα δεν υπάρχουν ουσιώδεις διαφορές.

```
ENTITY name IS PORT                                  module name (A, B);
(A : IN   STD_LOGIC_VECTOR(3 DOWNTO 0);               input   [3:0]   A;
(B : OUT   STD_LOGIC_VECTOR(3 DOWNTO 0);              output  [3:0]   B;
```

Σχήμα 4.3 Σύγκριση δηλώσεων πολλαπλών bit μεταξύ VHDL και Verilog

Παρακάτω θα γίνει σύγκριση δύο υλοποιήσεων ενός πολυπλέκτη 2 σε 1. Παρατηρείται ότι ενώ υπάρχει πληθώρα τρόπων που θα μπορούσε να υλοποιηθεί το κύκλωμα, επιλέχθηκαν οι δύο παρακάτω τρόποι. Και στις δύο περιπτώσεις ο κώδικας είναι ισοδύναμος του αντίστοιχου βρόγχου «if» που θα αναλυθεί μετέπειτα. Χρησιμοποιούνται οι εντολές «When-Else» και «Assign ?» αντίστοιχα. Η μόνη διαφορά βρίσκεται στην λειτουργία των δύο εντολών. Δηλαδή ενώ στην «Assign ?»

πρέπει να πραγματοποιηθεί ο έλεγχος απαραίτητα για θετική τιμή, στην «When-Else» δεν υπάρχει τέτοιος περιορισμός.

```
LIBRARY ieee;                                   module mux_2to1(A, B, Sel, Q);
USE ieee.std_logic_1164.all;                                    input A, B, Sel;
ENTITY Mux_2_to_1C IS                                           output Q;
 PORT (x0, x1, s : IN  STD_LOGIC;                        assign  Q=(Sel)?B:A;
       z : OUT STD_LOGIC);                                      endmodule
END Mux_2_to_1C;
ARCHITECTURE Behavior OF Mux_2_to_1C IS
 BEGIN
    z <=x0 WHEN s='0' ELSE x1;
END Behavior;
```

Σχήμα 4.4 Σύγκριση πολυπλέκτη 2 σε 1, για ειδικές περιπτώσεις υλοποίησης εντολής if με εναλλακτικούς τρόπους μεταξύ VHDL και Verilog

Συνεχίζοντας, θα πραγματοποιηθεί σύγκριση μεταξύ τμημάτων υλοποιήσεων του αποκωδικοποιητή 2 σε 4. Η υλοποίηση και στις δύο γλώσσες έχει γίνει με χρήση βρόγχου εντολών «case». Η εντολή «case» αποτελεί ένα τρόπο επιλογής μεταξύ διαφόρων τμημάτων κώδικα. Είναι πολύ βολική για τον συνεχόμενους έλεγχους καθώς η δομή της είναι σύντομη και περιεκτική. Και στις δύο περιπτώσεις οι εντολές «case» βρίσκονται εντός κάποιου μεγαλύτερου βρόγχου. Υπάρχουν διαφορές ως προς την συγγραφή των εντολών και διακρίνεται έντονα ο τρόπος αναπαράστασης αριθμών της Verilog. Επίσης πρέπει να γίνει αναφορά και στις «casex» και «casez» της Verilog οι οποίες επιτρέπουν και την χρήση αδιάφορων σημάτων ή σημάτων υψηλής αντίστασης.

```
PROCESS (s, e)                                  always @(s or e) begin
  BEGIN                                                   z <= s and e;
   IF e = '1' THEN                                         if (e)
     CASE s IS                                            casex(z)
    WHEN "00" => y <= "1000";                    3'b100: y <= 'b1000;
```

WHEN "01" => y <= "0100";	3'b101: y <= 'b0100;
WHEN "10" => y <= "0010";	3'b110: y <= 'b0010;
WHEN OTHERS => y <= "0001";	3'b111: y <= 'b0001;
END CASE;	endcase
ELSE	else
y <= "0000";	y<=0;
END IF;	end
END PROCESS;	

Σχήμα 4.5 Σύγκριση τμήματος αποκωδικοποιητή 2 σε 4, για την εντολή case μεταξύ VHDL και Verilog

Εν συνεχεία, θα γίνει σύγκριση μεταξύ δύο τμημάτων κώδικα από κύκλωμα D flip-flop, με ασύγχρονη είσοδο μηδενισμού. Παρακάτω φαίνεται η δομή των εντολών «if» της εκάστοτε γλώσσας. Η εντολή «if» είναι, όπως και σε άλλες γλώσσες προγραμματισμού ο τρόπος επιλογής μεταξύ διάφορων τμημάτων κώδικα. Λειτουργεί παρόμοια με την εντολή «case», με βάση την ιδέα ότι υπάρχουν πολλαπλές (όχι απαραίτητα) επιλογές για να συνεχιστεί ο κώδικας που υλοποιείται. Παρατηρείται ότι και στις δύο γλώσσες η δομή των «if» είναι ίδια με εξαίρεση μερικές μικρές διαφορές ως προς την σύνταξη. Ωστόσο, γίνεται εμφανής η διαφορά των δύο γλωσσών ως προς τον έλεγχο του ρολογιού. Στην Verilog ο έλεγχος πραγματοποιείται μέσα στον βρόγχο «always» ενώ στην VHDL υπάρχει ξεχωριστός έλεγχος μόνο για το ρολόι που γίνεται εντός της «if».

PROCESS (Clr, Clk)	always@(posedge clk) begin
BEGIN	if (reset)
IF Clr= '0' THEN	Q <= 1’b0;
Q<='0';	else
ELSIF Clk'EVENT AND Clk='1' THEN	Q <= D;
Q<=D;	end
END IF;	
END PROCESS;	

Σχήμα 4.6 Σύγκριση τμήματος D flip-flop, για την εντολή if μεταξύ VHDL και Verilog

Παρακάτω αναλύονται δύο υλοποιήσεις πλήρους αθροιστή τεσσάρων bit με χρήση της εντολής «generate». Μια ακόμα σύγκριση μεταξύ των δύο γλωσσών είναι αυτή ανάμεσα στις εντολές «generate». Η εντολή «generate» παρέχει την δυνατότητα να δημιουργείται νέος κώδικας από ήδη υλοποιημένο κώδικα. Βολεύει για περιπτώσεις όπου χρειάζεται η επανάληψη κώδικα πολλές φορές. Μεταξύ των δύο γλωσσών εμφανίζονται μερικές μικρές διαφορές. Ενώ και στις δύο περιπτώσεις η εντολή χρειάζεται βρόγχο επανάληψης «for» για να λειτουργήσει, η «generate» της Verilog χρειάζεται απαραίτητα έναν βρόγχο «for» στο εσωτερικό της όπου θα χρησιμοποιηθεί μια ειδική μεταβλητή «genvar». Επίσης, η VHDL έχει ήδη δημιουργημένο component στο υπόλοιπο τμήμα της αρχιτεκτονικής όποτε δεν είναι απαραίτητη η συγγραφή περαιτέρω κώδικα. Κάτι αντίστοιχο θα μπορούσε να είχε γίνει και στην Verilog όπου ο κώδικας του κυκλώματος σε μορφή μοντελοποίησης δομής ωστόσο δεν υπάρχουν διαφορές στο αποτέλεσμα.

```
BEGIN
C(0) <= cin ;
G1: FOR i IN 0 TO 3 GENERATE
stages: full_add PORT MAP (C(i), X(i), Y(i), S(i), C(i+1));
END GENERATE;
Cout <= C(4);
END Structured;
```

```
genvar k;
generate
for (k=0; k<n; k=k+1) begin: fa
wire w1,w2,w3;
xor(w1, x[k], y[k]);
xor (s[k], w1, cin);
and (w2, w1, cin);
and (w3, x[k], y[k]);
or (c[k+1], w2, w3);
end
endgenerate
```

Σχήμα 4.7 Σύγκριση τμήματος πλήρους αθροιστή, για την εντολή generate μεταξύ VHDL και Verilog

Τα testbench, οι κώδικες προσομοίωσης των γλωσσών περιγραφής υλικού αποτελούν μεγάλο τμήμα της διαδικασίας υλοποίησης κυκλωμάτων μέσω των εν λόγω γλωσσών. Οι κώδικες αυτοί χρησιμοποιούνται για τον έλεγχο του κυκλώματος και σκοπός τους είναι η οδήγηση του προσομοιωτή. Η δομή του κώδικα

προσομοίωσης είναι ίδια, δηλαδή η εισαγωγή των μεταβλητών του κυκλώματος, η αντιστοίχηση τους με μεταβλητές που έχουν δοθεί στο testbench, η αρχικοποίηση τους αν χρειάζεται, η εισαγωγή τιμών στις μεταβλητές που θα ελεγχθούν και εμφάνιση κάποιου μηνύματος αν χρειαστεί. Ωστόσο υπάρχουν κάποιες βασικές διαφορές μεταξύ των δύο. Αρχικά η Verilog απαιτεί αρχικοποίηση όλων των εισόδων ενώ η VHDL δεν το απαιτεί και τους παραθέτει τιμές όπως X μέχρι να λάβουν άλλες. Έπειτα, ενώ και οι δύο γλώσσες παρέχουν την δυνατότητα εισαγωγής τιμών συνεχόμενα στην Verilog γίνεται εύκολα με την χρήση εντολών «forever» και «always» ενώ η VHDL μπορεί να μιμηθεί κάτι αντίστοιχο χρησιμοποιώντας μόνο βρόγχους επανάληψης που κάνει την διαδικασία ιδιαίτερα δύσκολη. Πέρα από τα παραπάνω υπάρχουν και μικρές διαφορές στην ίδια την δομή της τοποθέτησης τιμών σε μεταβλητές, μια διαδικασία η οποία πρέπει να γίνει απαραίτητα εντός κάποιας διεργασίας για την VHDL και εντός ενός μπλοκ «always» ή «initial» για την Verilog. Και στις δύο γλώσσες μπορούν να εισαχθούν εντολές που να εμφανίζουν μηνύματα σε προσομοιωτή σχετικά με την λειτουργικότητα του κυκλώματος.

```
LIBRARY IEEE;                                          module dff_reset_tb;
USE IEEE.STD_LOGIC_1164.ALL;                                   reg D, clk, reset;
ENTITY D_flip_flopAtb IS                                       wire Q;
END D_flip_flopAtb;              dff_set_reset dut(.Q(Q), .reset(reset), .D(D), .clk(clk));


ARCHITECTURE behavior OF D_flip_flopAtb IS                     initial begin
SIGNAL D, Clk, Clr, Q : STD_LOGIC;                                   clk = 0;
COMPONENT D_flip_flopA                                         D = 0;
PORT (Clk, Clr, D : IN  STD_LOGIC;                                   reset = 0;
Q : OUT STD_LOGIC);                                    forever #10 clk = ~clk;
END COMPONENT;                                                 end
 BEGIN                                                         initial begin
  m1: D_flip_flopA PORT MAP(Clk=>Clk,Clr=>Clr,D=>D,Q=>Q);
PROCESS                                                #100; reset=1; D <= 0;
BEGIN                                                  #100; reset=0; D <= 1;
 Clr<='0';Clk<='0';D<='0'; WAIT FOR 40 ns;             #100; D <= 0;
```

```
Clr<='0';Clk<='1';D<='0'; WAIT FOR 40 ns;
Clr<='0';Clk<='0';D<='1'; WAIT FOR 40 ns;
Clr<='0';Clk<='1';D<='1'; WAIT FOR 40 ns;
Clr<='1';Clk<='0';D<='0'; WAIT FOR 40 ns;
Clr<='1';Clk<='1';D<='0'; WAIT FOR 40 ns;
Clr<='1';Clk<='0';D<='1'; WAIT FOR 40 ns;
Clr<='1';Clk<='1';D<='1'; WAIT FOR 40 ns;
END PROCESS;
END behavior;
```

```
#100; reset=1; D <= 1;
#100; D <= 0;
#100; reset=0; D <= 1;
#100; D <= 0;
#100; D <= 1;
#100; reset=1; D <= 0;
#100; reset=0; D <= 1;
#100; D <= 0;
#100; reset=1; D <= 1;
#100; D <= 0;
#100; D <= 1;
end
always@(posedge clk)
$monitor("At time = %4d, clk = %b, D = %b, reset = %b, Q = %b", $time, clk, D, reset, Q);
endmodule
```

Σχήμα 4.8 Σύγκριση testbench μεταξύ VHDL και Verilog

Μια επιπλέον παρατήρηση η οποία αφορά την εμφάνιση τιμών κατά την προσομοίωση είναι ότι και οι δύο γλώσσες προσφέρουν την δυνατότητα εμφάνισης τιμών κατά την προσομοίωση. Η VHDL έχει μια εντολή την «report», η οποία έχει την μορφή «report “message”», μπορεί να εμφανίζει μηνύματα σφαλμάτων ανάλογα με την σοβαρότητα (severity warning/error) και έχει την δυνατότητα να χρησιμοποιηθεί συνδυαστικά με την εντολή «assert» για να βελτιστοποιήσει την ανίχνευση σφαλμάτων. Σε αντίθεση με την VHDL, η Verilog έχει τέσσερις εντολές όλες παρόμοιες η μια με την άλλη. Η εντολή «$display» χρησιμοποιείται για εμφάνιση τιμών την στιγμή που εκτελείται, η «$strobe» η οποία λειτουργεί στο τέλος κάθε βήματος χρόνου, η «$monitor» η οποία εμφανίζει τιμές στο τέλος του βήματος χρόνου αν έχουν υπάρξει αλλαγές και η «$write» η οποία λειτουργεί σαν την «$display» αλλά δεν σταματάει στις αλλαγές γραμμών.

Συνεχίζοντας θα γίνει ανάλυση δύο διαφορετικών μορφών υλοποίησης ενός κωδικοποιητή προτεραιότητας 4 σε 2. Παραπάνω έγινε αναφορά στις εντολές «casez»

και «casex» της Verilog οι οποίες λειτουργούν ακριβώς όπως η «case» αλλά μπορούν να επεξεργαστούν τις τιμές «Z» και «X». Πιο συγκεκριμένα η «casez» μπορεί να επεξεργαστεί τιμές με «Z» στο εσωτερικό τους ενώ η «casex» μπορεί να επεξεργαστεί τιμές και με «X» και με «Z». Οι εντολές αυτές δεν έχουν περιορισμούς και μπορούν να χρησιμοποιούνται ελεύθερα σε κώδικα. Η VHDL δεν έχει κάτι αντίστοιχο και σε περιπτώσεις όπου χρειάζεται να γίνουν έλεγχοι και περιέχονται οι τιμές «X» και «Z» πρέπει να βρεθεί άλλος πιο πολύπλοκος τρόπος.

```
ARCHITECTURE Behavioral OF prior_enc_4_to_2 IS
   BEGIN
     PROCESS (x)
   BEGIN
       IF x(3) = '1' THEN
         y <= "11" ;
       ELSIF x(2) = '1' THEN
         y <= "10" ;
       ELSIF x(1) = '1' THEN
         y <= "01" ;
       ELSE
         y <= "00" ;
       END IF ;
     END PROCESS ;
      z <= '0' WHEN x = "0000" ELSE '1' ;
END Behavioral;
```

```
always@(y) begin
casex(y)
4'b0001:a = 2'b00;
4'b001x:a = 2'b01;
4'b01xx:a = 2'b10;
4'b1xxx:a = 2'b11;
default:
$display("Error!");
endcase
end
endmodule
```

Σχήμα 4.9 Σύγκριση τμήματος κωδικοποιητή προτεραιότητας 4 σε 2 για ένδειξη διαφορών εντολών if και casex μεταξύ VHDL και Verilog

Οι γλώσσες περιγραφής υλικού έχουν ενσωματωμένες τις βασικές λογικές πύλες για σκοπούς όπως η υλοποίηση λογικών συναρτήσεων. Ωστόσο υπάρχουν περιπτώσεις, όπου η Verilog είναι σχεδιασμένη να τις χρησιμοποιεί ενώ η VHDL δεν είναι. Όπως φαίνεται και στο σχήμα 4.7, όπου πραγματοποιούνται συγκρίσεις μεταξύ των εντολών «generate» η Verilog έχει την ικανότητα να χρησιμοποιεί λογικές πύλες

για μοντελοποίηση δομής. Δηλαδή, χρησιμοποιώντας μόνο λογικές πύλες, η Verilog παρέχει την δυνατότητα σχεδιασμού κυκλωμάτων. Η διαδικασία είναι αρκετά απλή, βολική για μικρά κυκλώματα και με βάση αυτή γράφτηκε ο κώδικας κάποιων εκ των πολλαπλασιαστών που περιγράφηκαν νωρίτερα. Βέβαια πρέπει να αναφερθεί ότι και οι δύο γλώσσες παρέχουν την δυνατότητα μοντελοποίησης βάση δομής αλλά ενώ η Verilog επιτρέπει την χρήση πυλών χωρίς περαιτέρω διαδικασίες, η VHDL χρειάζεται να κάνει χρήση «package».

```
module mux2to1(output out, input A, B, Sel);
        wire and_A,and_B,Selout;
        not ( Selout, Sel);
        and (and_A, A, Selout), (and_B, B, Sel);
        or (out, and_A, and_B);
    endmodule
```

Σχήμα 4.10 Υλοποίηση πολυπλέκτη 2 σε 1 με χρήση μοντελοποίησης πυλών

```
HA1:half_add PORT MAP(a1b0,a0b1,c01,ap01);

FA1:full_add PORT MAP(a2b0,a1b1,a0b2,c02,ap02);

FA2:full_add PORT MAP(a3b0,a2b1,a1b2,c03,ap03);

HA2:half_add PORT MAP(a3b1,a2b2,c04,ap04);
```

Σχήμα 4.11 Τμήμα κώδικα VHDL πολλαπλασιαστή αρχιτεκτονικής Wallace Tree που περιγράφεται χρησιμοποιώντας μοντελοποίηση δομής με αθροιστές που έχουν εισαχθεί από package

Τέλος θα γίνει σύγκριση μεταξύ δύο πολλαπλασιαστών 4x4 αρχιτεκτονικής carry save. Οι κώδικες που υλοποιούν τους πολλαπλασιαστές είναι ισοδύναμοι αλλά έχουν διαφορετική ονοματολογία. Επίσης αναγκαστικά για λόγους μήκους του κειμένου οι κώδικες θα γραφούν κάθετα αντί για οριζόντια. Ωστόσο με αυτή την σύγκριση μπορούν να διακριθούν κάποιες σημαντικές ομοιότητες και διαφορές μεταξύ των δύο γλωσσών. Το πρώτο που παρατηρείται είναι οι τρόποι εισαγωγής βοηθητικών υποκυκλωμάτων, με packages και με απλή συγγραφή τους στον κώδικα. Ο μεν τρόπος είναι συνοπτικός και ο δε είναι ο μακροσκελής, αγνοώντας το μικρό μέγεθος των κωδικών που εισάγονται σε αυτό το παράδειγμα. Όμως είναι εμφανές, παρά την χρήση των packages ότι ο κώδικας VHDL είναι μακρύτερος κατά πολύ από τον κώδικα Verilog. Το παραπάνω γίνεται λόγω δυσκολίας της VHDL για τον υπολογισμό των λογικών «AND». Έτσι επιλέχθηκε μια πιο χειροκίνητη λύση η οποία απαιτεί την χρήση πολλαπλών σημάτων μαζί με τον χειροκίνητο υπολογισμό όλων των μεταβλητών που θα χρειαστούν. Παράλληλα ο κώδικας της Verilog είναι πολύ

μικρότερος επειδή, μιμούμενη την γλώσσα C μπορεί να χειριστεί τις τιμές που χρειάζεται ως πίνακες δύο διαστάσεων. Όποτε χρησιμοποιώντας βρόγχους επανάληψης και «if» θα μπορούσε να υλοποιήσει τις πράξεις χωρίς κόπο. Σε αυτή τη περίπτωση όμως επιλέχθηκε η πιο χειροκίνητη έκδοση αυτού. Δηλαδή κατά την διάρκεια της υλοποίησης δέχεται δύο τιμές στις οποίες εκτελείται το λογικό «AND» και το αποτέλεσμα χρησιμοποιείται ακαριαία σε άλλους υπολογισμούς.

```
LIBRARY ieee;
LIBRARY work;
USE ieee.std_logic_1164.all;
USE work.full_add_package.all;
USE work.half_add_package.all;
ENTITY carrysave4X4 IS
        PORT (                  a3, a2, a1, a0 : IN  STD_LOGIC;
                        b3, b2, b1, b0 : IN  STD_LOGIC;
          p7,p6,p5,p4,p3, p2, p1, p0 : OUT STD_LOGIC);
END carrysave4X4;
ARCHITECTURE structured OF carrysave4X4 IS
        SIGNAL a1b0,a0b1,c1 : STD_LOGIC;
      SIGNAL a2b0,a1b1,a0b2,c2,c20,hp2 : STD_LOGIC;
      SIGNAL a3b0,a2b1,a1b2,a0b3,c3,c30,c31,hp3,fp3 : STD_LOGIC;
      SIGNAL a3b1,a2b2,a1b3,c4,c40,c41,fp40,fp41 : STD_LOGIC;
        SIGNAL a3b2,a2b3,c50,c51,fp50 : STD_LOGIC;
        SIGNAL a3b3 : STD_LOGIC;
        SIGNAL cout: STD_LOGIC;
BEGIN
--P0:
 p0<=a0 AND b0;
```

```
--P1:
  a1b0<=a1 AND b0;
  a0b1<=a0 AND b1;
  HA1:half_add PORT MAP (a1b0, a0b1, c1, p1);
--P2:
  a2b0<=a2 AND b0;
  a1b1<=a1 AND b1;
  HA2: half_add PORT MAP (a2b0, a1b1, c2, hp2);
  a0b2<=a0 AND b2;
  FA2: full_add PORT MAP (hp2, a0b2, c1, c20, p2);
--P3:
  a3b0<=a3 AND b0;
     a2b1<=a2 AND b1;  HA3: half_add PORT MAP (a3b0, a2b1, c3, hp3);
     a1b2<=a1 AND b2;
     FA30: full_add PORT MAP (hp3, a1b2, c2, c30, fp3);
     a0b3<=a0 AND b3;
     FA31: full_add PORT MAP (fp3, a0b3, c20, c31, p3);
   --P4:
     a3b1<=a3 AND b1;
     a2b2<=a2 AND b2;
     FA40: full_add PORT MAP (a3b1, a2b2, c3, c40, fp40);
     a1b3<=a1 AND b3;
     FA41: full_add PORT MAP (fp40, a1b3, c30,c41, fp41);
     HA4:  half_add PORT MAP (fp41, c31, c4, p4);
   --P5:
     a3b2<=a3 AND b2;
```

```
    a2b3<=a2 AND b3;
    FA50: full_add PORT MAP (a3b2, a2b3, c40, c50, fp50);
    FA51: full_add PORT MAP (fp50, c4, c41, c51, p5);
  --P6:
    a3b3<=a3 AND b3;
    FA6: full_add PORT MAP (a3b3, c50, c51, cout, p6);
  --P7:
    p7<=cout;
  END structured;
```

Σχήμα 4.12 Πολλαπλασιαστής αρχιτεκτονικής Carry Save με χρήση VHDL

```
    module HA(output sum, carry, input a, b);
          assign sum = a^b;
          assign carry = (a&b);
    endmodule
    module FA(output sum, carry, input a, b, cin);
          assign sum =(a^b^cin);
          assign carry = ((a&b)|(a&cin)|(b&cin));
  endmodule
  module carry_save_4x4(output [7:0] product, input [3:0] in1,in2,input clock);
          wire [10:0] s, c;
  assign product[0]= (in1[0]&in2[0]);
//row 1
HA HA1(product[1],c[0],(in1[1]&in2[0]),(in1[0]&in2[1])); //p1
//row 2
```

```
HA HA2(s[1],c[1],(in1[2]&in2[0]),(in1[1]&in2[1]));

FA FA1(product[2],c[2],c[0],(in1[0]&in2[2]),s[1]);//p2

//row 3

HA HA3(s[3],c[3],(in1[3]&in2[0]),(in1[2]&in2[1]));

FA FA2(s[4],c[4],s[3],(in1[1]&in2[2]),c[1]);

FA FA3(product[3],c[5],c[2],(in1[0]&in2[3]),s[4]);//p3

// row 4

FA FA4(s[6],c[6],(in1[3]&in2[1]),(in1[2]&in2[2]),c[3]);

FA FA5(s[7],c[7],s[6],(in1[1]&in2[3]),c[4]);

HA HA4(product[4],c[8],c[5],s[7]);//p4

//row 5

FA FA6(s[9],c[9],(in1[3]&in2[2]),(in1[2]&in2[3]),c[6]);

FA FA7(product[5],c[10],c[8],c[7],s[9]);//p5

//row 6

FA FA8(product[6],product[7],(in1[3]&in2[3]),c[10],c[9]);//p6,7

endmodule
```

Σχήμα 4.13 Πολλαπλασιαστής αρχιτεκτονικής Carry Save με χρήση Verilog

4.2 Ανάλυση και συμπεράσματα

Σε συνέχεια των παραπάνω συγκρίσεων και παρατηρήσεων θα πραγματοποιηθεί ανάλυση τους και έπειτα θα εξαχθούν συμπεράσματα. Τα χαρακτηριστικά των δύο γλωσσών θα περιγραφούν σε έναν πίνακα ο οποίος θα περιέχει τα εν λόγω χαρακτηριστικά και παρατηρήσεις σχετικά με αυτά. Έπειτα θα γίνει ανάλυση σε όλα τα περιεχόμενα του πίνακα με λεπτομέρειες.

Χαρακτηριστικό	VHDL	Verilog	Παρατηρήσεις
Μήκος κώδικα	Μεγάλο	Μικρό	Η VHDL είναι πιο φλύαρη
Ευαναγνωσία	Μεγάλη	Μικρή	Η VHDL έχει σαφής εντολές
Σαφήνεια κώδικα	Μεγάλη	Μικρή	Η Verilog έχει πιο πολύπλοκές εντολές
Ευελιξία	Μικρή	Μεγάλη	Η Verilog έχει πιο πολλές δυνατότητες

Ευκολία εκμάθησης	Μεγάλη	Μικρή	Η Verilog είναι όμοια με C σε αρκετούς τομείς
Βιβλιοθήκες	Ναι	Όχι	Η Verilog δεν χρειάζονται
Δηλώσεις μεταβλητών	Μακροσκελείς	Σύντομες	Η Verilog παρέχει επιλογές γραφής
Είδη μεταβλητών	Απαραίτητες στην δήλωση	Όχι απαραίτητες	Διαφορετικά είδη μεταβλητών
Δομή Κώδικα	Συγκεκριμένη	Πιο ελεύθερη	Παρόμοια δομή σε γενικό πλαίσιο
Βρόγχοι Επανάληψης	Βασικοί βρόγχοι	Ποικιλία βρόγχων	Η Verilog έχει βρόγχους όπως always και initial
Βρόγχοι If	Τυπικό	Τυπικό	Ίδιες σε λειτουργία
Βρόγχοι Case	Τυπικό	Περισσότερες ευκολίες	Η Verilog έχει εντολές όπως casex, casez
Βρόγχος Generate	Απλός	Σύνθετος	Η Verilog απαιτεί περαιτέρω διαδικασίες
Μηνύματα κονσόλας	Λίγες επιλογές	Πολλές επιλογές	Η VHDL έχει λιγότερες εντολές και πιο δύσχρηστες
Μοντελοποίηση δομής	Μέσω packages	Χωρίς περιορισμούς	Η Verilog έχει μπορεί να χρησιμοποιήσει λογικές πύλες
Χρήση πινάκων	Δυσκολότερη	Ευκολότερη	Η Verilog διαχειρίζεται ευκολότερα πίνακες λόγω ομοιότητας με C
Χρήση κωδικών εκτός του κυρίως κώδικα	Εύκολη	Αντιγραφή	Η VHDL χρησιμοποιεί packages

Πίνακας 4.1 Ανάλυση χαρακτηριστικών των VHDL και Verilog

1. Μήκος κώδικα : Η VHDL ως γλώσσα έχει σε γενικό πλαίσιο πιο μακροσκελή γραφή, εντολές που απαιτούν μεγαλύτερο μήκος και χρειάζεται να συμπεριλάβει πράγματα όπως βιβλιοθήκες που μεγαλώνουν το μήκος του μέσου κώδικα. Σε αντίθεση η Verilog δεν απαιτείται να συμπεριλάβει τίποτα στον κώδικα και έχει την δυνατότητα σε κάποια πλαίσια όπως δήλωση μεταβλητών να «ανοίξει» ή να «μαζέψει» τον κώδικα όσο θέλει.

2. Ευαναγνωσία : Η ευαναγνωσία είναι κατά ένα βαθμό συνδεδεμένη με το μήκος κώδικα. Η VHDL έχει πιο μακριές αλλά πιο εξηγήσιμες εντολές και γραφή. Αποτέλεσμα είναι ένα τμήμα κώδικα VHDL ακόμα κάποιος που δεν γνωρίζει την γλώσσα να μπορεί να διαβάσει τον κώδικα και να εξάγει κάποια συμπεράσματα. Αντίθετα το παραπάνω είναι πιο δύσκολο στην Verilog γιατί ο μειωμένος όγκος κώδικα μπορεί να λειτουργήσει και υπέρ αλλά και κατά της.

3. Σαφήνεια κώδικα : Η σαφήνεια κώδικα αποτελείται εν μέρει από το μήκος κώδικα και την ευαναγνωσία. Όπως επεξηγήθηκε πριν, η VHDL έχει πιο μακρύ και ευανάγνωστο κώδικα με εντολές οι οποίες περιγράφουν ακριβώς την λειτουργία τους. Με δεδομένο αυτό η VHDL έχει πιο μακρύ και αναλυτικό κώδικα και κατ' επέκταση τον πιο σαφή εκ των δύο. Η Verilog, όπως θα αναλυθεί παρακάτω είναι πιο ευέλικτη αλλά και πιο δύσκολη στην ανάγνωση και την κατανόηση, ειδικότερα από αρχάριους.

4. Ευελιξία : Η ευελιξία είναι αποτέλεσμα του καιρού κατά τον οποίο δημιουργήθηκαν οι γλώσσες. Δηλαδή, η VHDL όντας η παλαιότερη εκ των δύο γλωσσών και πρώτη στο είδος της, έχει χτιστεί με διαφορετικές προδιαγραφές και εμπειρίες σχεδιασμού από ότι η Verilog. Η Verilog μπορούσε να βασιστεί και βασίστηκε σε μαθήματα και ιδέες που βγήκαν από την VHDL. Παράλληλα με την VHDL βασίστηκε και στην γλώσσα C με αποτέλεσμα να συμπεριλαμβάνει και να βελτιώνει ιδέες και από τις δύο γλώσσες. Είναι πιο ρευστή, με μεγαλύτερη ποικιλία εντολών και περισσότερες δυνατότητες από την VHDL. Η Verilog είναι ικανή να πραγματοποιήσει εύκολα δομές κώδικα που μπορεί να δυσκολεύουν την VHDL.

5. Ευκολία εκμάθησης : Η VHDL με δεδομένο ότι είναι η πιο σαφής και ευανάγνωστη από τις δύο μπορεί να διδαχτεί πιο εύκολα από τις δύο, ειδικά σε περιπτώσεις, όπου ο μαθητευόμενος δεν έχει περαιτέρω εμπειρία στον τομέα. Ωστόσο πρέπει να αναφερθεί ότι σε περίπτωση που ο μαθητευόμενος γνωρίζει την γλώσσα C τότε μπορεί πολύ ευκολότερα να μάθει Verilog. Δηλαδή ένα πανεπιστήμιο με πρόγραμμα σπουδών που περιέχει C ή (και) C++ μπορεί με μεγάλη άνεση να παρέχει μαθήματα εκμάθησης και των δύο γλωσσών.

6. Βιβλιοθήκες : Οι βιβλιοθήκες αποτελούν βασικό τμήμα της VHDL καθώς κάθε κώδικας περιέχει την βιβλιοθήκη IEEE. Μπορούν να δημιουργηθούν περαιτέρω βιβλιοθήκες για να καλύψουν τις ανάγκες οποιουδήποτε κώδικα χρειάζεται. Οι βιβλιοθήκες δεν υπάρχουν στην Verilog. Η Verilog δεν απαιτεί βιβλιοθήκες για την λειτουργία της καθώς βάσει σχεδιασμού δεν χρειάζονται. Η Verilog ως τμήμα της System Verilog μπορεί να έχει βιβλιοθήκες αλλά μόνη της προ του προτύπου της IEEE 1800-2009 δεν έχει.

7. Δηλώσεις μεταβλητών : Οι δηλώσεις μεταβλητών της VHDL συγκροτούν το δικό τους τμήμα κώδικα, η επονομαζόμενη «οντότητα». Είναι πιο μακροσκελές από το αντίστοιχο τμήμα της Verilog με συγκεκριμένη δομή και υποχρεωτική δήλωση τύπου μεταβλητών σε κάθε μεταβλητή. Και στις δύο περιπτώσεις μπορεί να γίνει δήλωση τύπου μεταβλητής σε πολλαπλές μεταβλητές αλλά η Verilog μπορεί να συμπτύξει ή να αναπτύξει την δήλωση ανάλογα.

8. Είδη μεταβλητών : Και οι δύο γλώσσες περιέχουν μεταβλητές τύπου εισόδου/εξόδου, αριθμητικών ακέραιων για πίνακες και δημιουργία πινάκων. Ωστόσο, η Verilog δεν έχει πολλά είδη μεταβλητών όπως το STD_LOGIC της VHDL γιατί δεν είναι απαραίτητα για την λειτουργία της. Στην Verilog κάθε μεταβλητή μπορεί να δεχτεί οποιαδήποτε τιμή χρειάζεται. Οι μεταβλητές τύπου wire και register που χρησιμοποιούνται σχετίζονται κυρίως με την έξοδο των μεταβλητών παρά με τις ίδιες της μεταβλητές.
9. Δομή κώδικα : Η VHDL έχει μια συγκεκριμένη δομή η οποία πρέπει να ακολουθείται πάντα. Δηλαδή πρώτον ορισμός οντότητας και δηλώσεις μεταβλητών και δεύτερον ορισμός αρχιτεκτονικής και συμπεριφοράς του κυκλώματος. Η Verilog ενώ έχει παρόμοια δομή δεν έχει περιορισμούς. Ορισμένες ενέργειες πρέπει να γίνουν πριν από άλλες όπως η δήλωση μεταβλητών αλλά πέρα από κάποιες βασικές ενέργειες ο κώδικας Verilog μπορεί να έχει όποια δομή επιθυμεί.

10. Βρόγχοι επανάληψης : Η VHDL χρησιμοποιεί τυπικούς βρόγχους επανάληψης όπως «For» και «While» για όλες τις ανάγκες της. Η Verilog λειτουργεί πιο περίεργα με τους βρόγχους επαναλήψεων. Χρησιμοποιεί κανονικά τους τυπικούς βρόγχους επανάληψης όπως η VHDL. Περιέχει εντολές όπως «Always» και «Initial» οι οποίες λειτουργούν διαφορετικά από την μέση εντολή επανάληψης. Οι βρόγχοι αυτοί ονομάζονται και «Μπλοκ» για να δείξουν την διαφορά μεταξύ των δύο.

11. Βρόγχοι If : Έχουν την ίδια λειτουργικότητα και στις δύο γλώσσες ακριβώς με εξαίρεση μόνο την γραφή. Η λειτουργία και η δομή τους είναι ίδια με μερικές μικρές διαφορές και αλλαγές.

12. Βρόγχοι Case : Όπως και οι βρόγχοι If έχουν την ίδια λειτουργικότητα με εξαιρέσεις στην γραφή τους. Ωστόσο μια πολύ βασική διαφορά είναι η δυνατότητα της Verilog να μπορεί να επεξεργαστεί τιμές για υψηλή αντίσταση και «don’t care» με τις εντολές «casex» και «casez». Στην συγκεκριμένη περίπτωση η VHDL θα πρέπει να βρει εναλλακτική ενώ η Verilog παρέχει απευθείας λύση.

13. Βρόγχος Generate : Παρόμοια με τους βρόγχους If και Case, δεν υπάρχουν ιδιαίτερες διαφορές μεταξύ τους έτσι και οι Generate δεν έχουν ιδιαίτερες διαφορές. Αλλά πρέπει να ειπωθεί ότι στην Verilog είναι πιο δύσκολη η χρήση της Generate και λίγο πιο σύνθετη απαιτώντας και χρήση δικής της ειδικής μεταβλητής.

14. Μηνύματα κονσόλας : Η VHDL έχει λίγα και δύσχρηστα μηνύματα κονσόλας με πιο συνηθισμένο το report, τα οποία συνήθως χρησιμοποιούνται κυρίως για

αποσφαλμάτωση. Αντίθετα η Verilog έχει πολλές και εύχρηστες επιλογές για μηνύματα κονσόλας, όπως $display ή $monitor το καθένα από τα οποία μπορεί να δουλέψει με τον δικό του μοναδικό τρόπο.

15. Μοντελοποίηση δομής : Η μοντελοποίηση δομής που αναλύθηκε στην δομή των γλωσσών αποτελεί έναν πολύ σημαντικό τρόπο για την περιγραφή κυκλωμάτων. Όπως φαίνεται και σε κάποια κυκλώματα πολλαπλασιαστών η VHDL και η Verilog λειτουργούν πολύ διαφορετικά όταν πρέπει να χρησιμοποιήσουν αυτό τον τρόπο σχεδίασης. Η VHDL χρησιμοποιεί έτοιμα υποκυκλώματα δηλώνοντας τα ως package, από το πιο απλό ως το πιο σύνθετο. Η Verilog αντίθετα επιτρέπει την χρήση κωδικών με δεδομένο ότι έχουν περιγραφεί ως module στο ίδιο αρχείο με τον κυρίως κώδικα. Η Verilog παρέχει επίσης την ικανότητα της χρήσης λογικών πυλών σε μοντελοποίηση δομής χωρίς να χρειαστεί επιπλέον κώδικας.

16. Χρήση πινάκων : Η χρήση πινάκων και στις δύο γλώσσες είναι πιο υποτυπώδης από ότι σε άλλες γλώσσες προγραμματισμού. Η χρήση τους στην Verilog είναι πάρα πολύ απλή και λειτουργεί με τρόπο παρόμοιο αν όχι ίδιο με την C. Η VHDL ενώ μπορεί να διαχειριστεί πίνακες μίας διάστασης με τεράστια ευκολία, βρίσκει εμπόδια σε πίνακες μεγαλύτερων διαστάσεων. Η μεγαλύτερη ευελιξία της Verilog όμως επιτρέπει την ευκολότερη χρήση των εν λόγω πινάκων.

17. Χρήση κωδικών εκτός του κυρίως κώδικα : Και οι δύο γλώσσες έχουν αυτή τη δυνατότητα όπως αναφέρθηκε παραπάνω. Η VHDL εισάγει αυτούς τους κώδικες με την χρήση packages τα οποία αποτελούνται από τους κώδικες που χρειάζεται να συμπεριληφθούν. Ανάλογα αν χρειαστεί μπορεί να εισαγχθούν και βιβλιοθήκες. Η Verilog λειτουργεί διακρίνοντας διαφορετικά module. Με βάση το παραπάνω μπορούν να γραφούν όλοι οι κώδικες που απαιτούνται στο ίδιο αρχείο. Έτσι υπάρχει το κυρίως module του κυρίως προγράμματος και διαφορετικά μικρότερα που βοηθούν την υλοποίηση στο ίδιο αρχείο.

Με βάση τα παραπάνω εξάγονται τα συμπεράσματα ως προς την ευχρηστία των δύο γλωσσών. Οι δύο γλώσσες περιγραφής υλικού VHDL και Verilog, οι πρώτες δύο στον κλάδο τους αποτελούν δύο διαφορετικά προϊόντα δύο διαφορετικών δεκαετιών. Η VHDL είναι μια γλώσσα ευανάγνωστη και σαφής, ιδανική για νέους μαθητευόμενους παρά την ηλικία της. Η Verilog είναι μια γλώσσα ευέλικτη αν και πιο πολύπλοκη, ιδανική για άτομα που σκοπεύουν να διαθέσουν παραπάνω χρόνο στην εκμάθηση της ή γνωρίζουν C. Και οι δύο γλώσσες έχουν τα υπέρ και τα κατά, τα θετικά και τα αρνητικά τους. Η VHDL είναι μακροσκελής και φλύαρη με επεξηγηματικό κώδικα ενώ η Verilog είναι σύντομη και περιεκτική με δυνατότητα για μεγάλο όγκο πληροφορίας σε κάθε γραμμή. Όσο μεγαλύτερη ανάλυση γίνεται τόσες πιο πολλές διαφορές αλλά και ομοιότητες μπορούν να βρεθούν. Ωστόσο καμία

δεν είναι ανώτερη της άλλης. Και οι δύο είναι δημοφιλείς στην αγορά για κάποιους λόγους. Και οι δύο διδάσκονται για κάποιους λόγους. Δεν είναι ίσες σε κάθε τομέα αλλά η καθεμία ξεχωριστά αποτελεί το δικό της μοναδικό πακέτο εντολών, βρόγχων και ιδεών, τα οποία έχουν χρήσεις στην αγορά και στην επιστήμη της πληροφορικής και των υπολογιστών σε γενικό επίπεδο.

Βιβλιογραφία - Διαδικτυακές πηγές

Κωνσταντίνος Π. Ευσταθίου, 2013. *Ψηφιακή Σχεδίαση 2η Έκδοση*. Αθήνα : Εκδόσεις Νέων Τεχνολογιών

Κωνσταντίνος Π. Ευσταθίου, 2019. *Τεχνολογία Υλικού Υπολογιστικών Συστημάτων*. Αθήνα : Εκδόσεις Νέων Τεχνολογιών

Stephen Brown and Zvonko Vranesic, 2013. *Fundamentals of Digital Logic with Verilog Design 3rd Edition*. New York: McGraw-Hill Education

Stephen Brown και Zvonko Vranesic, 2015. *Σχεδίαση Ψηφιακών Συστημάτων με τη Γλώσσα VHDL 3η Έκδοση*. Μεταφράστηκε από Αγγλικά από Μ.Γ. Δημόπουλος, Ν.Ι. Μάργαρης, Π.Χρ. Κούρος, Χρ.Β. Τζίκας, Ι. Πεταλάς. Θεσσαλονίκη Εκδόσεις Τζίολα

Ιωάννης Βογιατζής, 2018. *Τεχνολογία και Σχεδίαση Ψηφιακών Συστημάτων*. Αθήνα : Εκδόσεις Τσότρας

Neil H. E. Weste και David M. Harris, 2011. *Σχεδίαση Ολοκληρωμένων Συστημάτων CMOS VLSI 4η Έκδοση*. Μετάφραση από Αγγλικά από Ε. Γκαγκάτσιου. Αθήνα : Εκδόσεις Παπασωτηρίου

M. Morris Mano, Michael D. Ciletti, 2014. *Ψηφιακή Σχεδίαση με Εισαγωγή στην Verilog HDL 5η Έκδοση*. Μετάφραση από Αγγλικά από Ε. Γκαγκάτσιου. Άθηνα : Εκδόσεις Παπασωτηρίου

A. Nejat Ince and Arnold Bragg, 2007. *Recent Advances in Modeling and Simulation Tools for Communication Networks and Services.* New York : Springer Science + Business Media, LLC.

Mohamed Khalgui and Hans-Michael Hanisch, 2011. *Reconfigurable Embedded Control Systems : Applications for Flexibility and Agility*. New York : Information Science Reference.

Stuart Sutherland, Simon Davidmann and Peter Flake, 2006. *System Verilog for Design Second Edition, A Guide to Using System Verilog for Hardware Design and Modeling.* New York: Springer Science + Business Media, LLC.

University of Maryland, Baltimore County, 2005. *Verilog Types and Constants*. Διαθέσιμο από: https://www.csee.umbc.edu/portal/help/VHDL/verilog/types.html [Accessed 24 November 2020]

Xilinx Corporation, 2013. *Modeling Concepts.* Διαθέσιμο από: https://www.xilinx.com/support/documentation/university/Vivado-Teaching/HDL-Design/2013x/Nexys4/Verilog/docs-pdf/lab1.pdf [Accessed 24 December 2020]

Carleton School of Information Technology. *Registers*. Διαθέσιμο από: http://www.csit-

sun.pub.ro/courses/Masterat/Xilinx%20Synthesis%20Technology/toolbox.xilinx.com/docsan/xilinx4/data/docs/xst/hdlcode4.html [Accessed 3 February 2021]

Dominique Thiebaut, Professor Emeritus of Computer Science Smith College, 2012. Διαθέσιμο από: http://www.science.smith.edu/dftwiki/index.php/Xilinx_ISE_Four-Bit_Adder_in_Verilog [Accessed 11 January 2021]

Xilinx Corporation. *Behavioral Modeling and Timing Constraints.* Διαθέσιμο από: https://www.xilinx.com/support/documentation/university/ISE-Teaching/HDL-Design/14x/Nexys3/Verilog/docs-pdf/lab7.pdf [Accessed 8 May 2021]

Department of Computer Science, Georgetown University, 2006. *Lab 3 : Dataflow and Behavioral Modeling of Combinational Circuits with Verilog HDL*. Διαθέσιμο από: http://people.cs.georgetown.edu/~squier/Teaching/HardwareFundamentals/LC3-trunk/docs/verilog/Verilog-DataflowAndBehavioralModelingOfCircuits-2006.pdf [Accessed 28 March 2021]

National Programme on Technology Enhanced Learning, 2006. *Module 4 Design of Embedded Processors*. Διαθέσιμο από: https://nptel.ac.in/content/storage2/courses/108105057/Pdf/Lesson-21.pdf [Accessed 27 March 2021]

Cathcart & District Housing Association, 2018. *16 bit ripple carry adder Verilog code examples.* Διαθέσιμο από: https://cathcartha.co.uk/sites/16-bit-ripple-carry-adder-verilog-code-examples-3101.php [Accessed 22 February 2021]

Auburn University, Victor P. Nelson 2017. *VHDL Simulation.* Διαθέσιμο από: https://www.eng.auburn.edu/~nelsovp/courses/elec4200/Slides/VHDL%206%20Test bench.pdf [Accessed 16 January 2021]

University of Utah, Erik Brunvand, 2013. *cs6710-testbench.pptx*. Διαθέσιμο από: https://my.eng.utah.edu/~cs6710/slides/cs6710-testbenchx2.pdf [Accessed 17 November 2021]

Cornell Engineering, 1999. *A Verilog HDL Test Bench Primer*. Διαθέσιμο από: https://people.ece.cornell.edu/land/courses/ece5760/Verilog/LatticeTestbenchPrimer.pdf [Accessed 17 November 2021]

Πανεπιστήμιο Ιωαννίνων, Τμήμα Μηχανικών Πληροφορικής και Υπολογιστών, Χρυσοβαλάντης Καβουσιανός, 2014. *4η Θεματική Ενότητα : Συνδυαστική Λογική.* Διαθέσιμο από: https://www.cs.uoi.gr/~kabousia/pdf/BasicCircuitTheory/Th4_CombinLog.pdf [Accessed 9 June 2021]

Μιχαήλ, Μ. 2009. *ΗΜΥ-211: Εργαστήριο Σχεδιασμού Ψηφιακών Συστημάτων Χειμερινό Εξάμηνο 2009.* Πανεπιστήμιο Κύπρου. Διαθέσιμο από:

http://www.eng.ucy.ac.cy/mmichael/courses/ECE210/lab_lecture_seq%20%5BCompatibility%20Mode%5D.pdf [Accessed 9 June 2021]

PowerSim Corporation, 1994. *PSIM MODCOUPLER.* https://powersimtech.com/products/psim/psim-modules/modcoupler/ [Accessed 9 June 2021]

Siemens Corporation, 2020. *Siemens SW HDL Simulation ModelSim FS 78330 C2.* Διαθέσιμο από: https://static.sw.cdn.siemens.com/siemens-disw-assets/public/6gMVEbq0wlPDGdqA1yvNuE/en-US/Siemens%20SW%20HDL%20Simulation%20ModelSim%20FS%2078330%20C2.pdf [Accessed 9 June 2021]

Xilinx Corporation, 2009. *Xilinx Xcell Issue 32 Quarterly Journal (Q299).* Διαθέσιμο από: https://www.xilinx.com/publications/archives/xcell/Xcell32.pdf [Accessed 9 June 2021]

Numato Systems Pvt. Ltd, 2017. *CPLD vs FPGA: Differences between them and which one to use?.* Διαθέσιμο από: https://numato.com/kb/cpld-vs-fpga-differences-one-use/ [Accessed 9 June 2021]

University of Idaho, Robert Prew, 2016. *Battle Over the FPGA: VHDL vs Verilog! Who is the True Champ?.* Διαθέσιμο από: https://blog.digilentinc.com/battle-over-the-fpga-vhdl-vs-verilog-who-is-the-true-champ/ [Accessed 9 June 2021]

web.archive.org, 2007. *Archive of "http://filebox.vt.edu/users/tmagin/history.htm"* Διαθέσιμο από: https://web.archive.org/web/20070412183416/http://filebox.vt.edu/users/tmagin/history.htm [Accessed 9 June 2021]

ELPROCUS, 20021. *Designing of 2 to 4 Line Decoder.* Διαθέσιμο από: https://www.elprocus.com/designing-of-2-to-4-line-decoder/ [Accessed 5 July 2021]

Electronics Hub, Ravi Teja, 2021. *Multiplexer (MUX) and Multiplexing.* Διαθέσιμο από: https://www.electronicshub.org/multiplexerandmultiplexing/ [Accessed 5 July 2021]

web.achive.org, 2007 *Archive of "http://www.csc.uvic.ca/~csc556/notes/FPGA.doc.html"* Διαθέσιμο από: https://web.archive.org/web/20070406203518/http://www.csc.uvic.ca/~csc556/notes/FPGA.doc.html [Accessed 9 June 2021]

Bohsali, M. , Doan, M. ,2010. *Rectangular Styled Wallace Tree Multipliers*
web.achive.org, 2010 *Archive of "http://www.veech.com/index_files/Wallace%20Tree.pdfl"* Διαθέσιμο από: https://web.archive.org/web/20100215152142/http://www.veech.com/index_files/Wallace%20Tree.pdf [Accessed 9 June 2021]

Monika Vaishnav, 2012. *Design of multi-precision reconfigurable Wallace Tree Multiplier for high performance applications.* Διαθέσιμο από: https://www.academia.edu/2118355/Design_of_multi-precision_reconfigurable_Wallace_Tree_Multiplier_for_high_performance_applications [Accessed 9 June 2021]

Subhodip Maulik, Sayani Sarkar and Srismrita Basu, 2019. *Relative Study of Power and Delay in 8X8 Precision Multipliers.* Διαθέσιμο από: https://www.academia.edu/41653210/Relative_Study_of_Power_and_Delay_in_8X8_Precision_Multipliers [Accessed 9 June 2021]

Ζάννας Κωνσταντίνος, Πτυχιακή Εργασία, 2010. *Πτυχιακή Ζάννας Κωνσταντίνος ΑΕΜ 11547.* Διαθέσιμο από: http://ikee.lib.auth.gr/record/124065/files/%CE%A0%CF%84%CF%85%CF%87%CE%B9%CE%B1%CE%BA%CE%AE%20%CE%96%CE%AC%CE%BD%CE%BD%CE%B1%CF%82%20%CE%9A%CF%89%CE%BD%CF%83%CF%84%CE%B1%CE%BD%CF%84%CE%AF%CE%BD%CE%BF%CF%82%20%CE%91%CE%95%CE%9C%2011547.pdf?fbclid=IwAR3L9XsOd4svSD9X6fC64ZQO1840q7RD10NGRHEm6NLM_-KLvs3IJNCvi9A [Accessed 9 June 2021]

Shaori Guo and Wayne Luk, 2006. *Compiling Ruby into FPGAs.* PhD Thesis. Oxford University. Διαθέσιμο από: http://www.doc.ic.ac.uk/~wl/papers/fpl95.pdf [Accessed 9 June 2021]

Oxford University, 2000. *Ruby.* Διαθέσιμο από: http://www.cs.ox.ac.uk/people/geraint.jones/ruby/ [Accessed 9 June 2021]

MyHDL community, 2015. *Overview.* Διαθέσιμο από: http://www.myhdl.org/start/overview.html [Accessed 9 June 2021]

Chisel community, 2019. *Chisel/FIRRTL Hardware Compiler Framework.* Διαθέσιμο από: https://www.chisel-lang.org/ [Accessed 9 June 2021]

Chris Spear, 2006. OpenVera. Διαθέσιμο από: http://chris.spear.net/openvera/default.htm [Accessed 9 June 2021]

Endeavor Business Media, LLC, 2021. *What's the Difference Between VHDL, Verilog, and System Verilog?.* Διαθέσιμο από: https://www.electronicdesign.com/resources/whats-the-difference-between/article/21800239/whats-the-difference-between-vhdl-verilog-and-systemverilog?accession=ysu1578311566143241&disposition=inline [Accessed 9 June 2021]

Institute of Electrical and Electronics Engineers, 2019. *IEEE 1076-2019 - IEEE Standard for VHDL Language Reference Manual.* Διαθέσιμο από: https://standards.ieee.org/standard/1076-2019.html [Accessed 5 July 2021]

Institute of Electrical and Electronics Engineers, 2021. *IEEE SA - About Us.* Διαθέσιμο από: https://standards.ieee.org/about/index.html [Accessed 5 July 2021]

Institute of Electrical and Electronics Engineers, 2006. *1364-2005 - IEEE Standard for Verilog Hardware Description Language.* Διαθέσιμο από: https://ieeexplore.ieee.org/document/1620780 [Accessed 5 July 2021]

Institute of Electrical and Electronics Engineers, 2018. *IEEE 1800-2017 - IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language.* Διαθέσιμο από: https://standards.ieee.org/standard/1800-2017.html [Accessed 5 July 2021]

Institute of Electrical and Electronics Engineers, 2012. *1666-2011 - IEEE Standard for Standard SystemC Language Reference Manual.* Διαθέσιμο από: https://ieeexplore.ieee.org/document/6134619 [Accessed 5 July 2021]

Institute of Electrical and Electronics Engineers, 2011. *P1666 - Standard for Standard SystemC Language Reference Manual.* Διαθέσιμο από: https://standards.ieee.org/project/1666.html [Accessed 5 July 2021]

Cleveland State University, 2018. *SystemVerilog vs Verilog in RTL design.* Διαθέσιμο από: https://academic.csuohio.edu/chu_p/rtl/fpga_mcs_vlog_book/SystemVerilog%20vs%20Verilog%20in%20RTL%20design.pdf [Accessed 5 July 2021]

Accellera Systems Initiative, 2021. *About SystemC.* Διαθέσιμο από: https://www.accellera.org/community/systemc/about-systemc [Accessed 5 July 2021]

Open SystemC Initiative, 2008. *About SystemC (Archived).* Διαθέσιμο από: https://web.archive.org/web/20081021173343/http://www.systemc.org/community/about_systemc/ [Accessed 5 July 2021]

Mentor Graphics, 2015. *ModelSim® Command Reference Manual Software Version 10.4c.* Oregon: Mentor Graphics Corporation. Διαθέσιμο από: https://www.microsemi.com/document-portal/doc_view/136364-modelsim-me-10-4c-command-reference-manual-for-libero-soc-v11-7 [Accessed 5 July 2021]

Mentor Graphics, 2016. *ModelSim® Command Reference Manual Software Version 10.5c.* Oregon: Mentor Graphics Corporation. Διαθέσιμο από: https://www.microsemi.com/document-portal/doc_view/136660-modelsim-me-10-5c-reference-manual-for-libero-soc-v11-8 [Accessed 5 July 2021]

Model Technology Incorporated, 2010. *ModelSim® User's Manual & Command Reference.* Διαθέσιμο από: http://www.pldworld.com/_hdl/2/_ref/se_html/manual_html/a_tnt5.html [Accessed 5 July 2021]

Model Technology Incorporated, 2010. *Lesson 5 - Running a batch-mode simulation.* Διαθέσιμο από: http://www.pldworld.com/_hdl/2/_ref/se_html/tutorial_html/t_batch.html [Accessed 5 July 2021]

Synario Design Automation, 1997. *VHDL Reference Manual.* Washington: Synario Design Automation. Διαθέσιμο από: https://www.ics.uci.edu/~jmoorkan/vhdlref/Synario%20VHDL%20Manual.pdf [Accessed 5 July 2021]

Sigasi, Philippe Faes, 2015. *VHDL Assert and Report.* Διαθέσιμο από: https://insights.sigasi.com/tech/vhdl-assert-and-report/ [Accessed 5 July 2021]

B.Maha Lakshmi, M.Bhavani, 2019. *Design and Implementation of 16-Bit Baugh Wooley Multiplier*. Διαθέσιμο από: http://www.internationaljournalssrg.org/IJECE/2018/Volume5-Issue12/IJECE-V5I12P101.pdf [Accessed 30 July 2021]

Pramodini Mohanty, 2013. *An Efficient Baugh-Wooley Architecture for Signed & Unsigned Fast Multiplication.* Διαθέσιμο από: https://www.niet.co.in/nietpfd/4_An%20Efficient%20Baugh-Wooley%20Architecture%20for%20Signed%20&%20Unsigned%20Fast%20Multiplication.pdf [Accessed 30 July 2021]

P. Yasodha Devi, S. Kalaivani, S. Manimegalai, Andril Alagusabai, 2013. *Design of Fixed-Width Multiplier Using Baugh-Wooley Algorithm.* Διαθέσιμο από: https://www.ijert.org/research/design-of-fixed-width-multiplier-using-baugh-wooley-algorithm-IJERTV2IS110751.pdf [Accessed 30 July 2021]

Gudivada, A.A., Sudha, G.F. *Design of Baugh–Wooley multiplier in quantum-dot cellular automata using a novel 1-bit full adder with power dissipation analysis.* SN Appl. Sci. **2,** 813 (2020). https://doi.org/10.1007/s42452-020-2595-5