ΑΣΚΗΣΗ 6

Συνοδευτικά αρχεία : 6_1.v, 6_2.v, 6_3.v

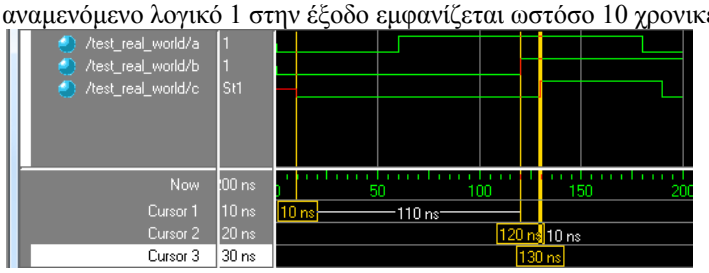
Μέρος 1 : Εισαγωγή

Σκοπός της 6ης άσκησης είναι να σας ενημερώσει σε θέματα περιγραφής αρτηριών (κάποια παραδείγματα με αρτηρίες υπήρξαν και στη προηγούμενη άσκηση προς χάριν ευκολίας), λογικών πράξεων πάνω σε αρτηρίες, αλλά και σε θέματα εκτίμησης της καθυστέρησης ενός συνδυαστικού κυκλώματος. Στο τέλος αυτής της άσκησης, θα πρέπει να μπορείτε να βρίσκετε τη χειρότερη καθυστέρηση ενός κυκλώματος και να γράφετε κώδικα χρησιμοποιώντας αρτηρίες.

Μέχρι τώρα στις ασκήσεις μας υποθέταμε ότι το μοναδικό ζητούμενο ήταν να περιγράφουμε κυκλώματα τα οποία να υλοποιούν κάποιες λογικές συναρτήσεις. Δυστυχώς στο πραγματικό κοσμο αυτό δεν είναι αρκετό˙ χρειάζεται μεταξύ άλλων οι λογικές αυτές συναρτήσεις να υλοποιούνται το δυνατόν γρηγορότερα. Κάθε πύλη του φυσικού κόσμου έχει μια καθυστέρηση για τη παραγωγή των εξόδων της. Αυτό μπορούμε να το μεταφέρουμε στη Verilog προσάπτοντας χρονοκαθυστέρηση στην λειτουργία ενός module. Ας δούμε το εξής παράδειγμα κώδικα :

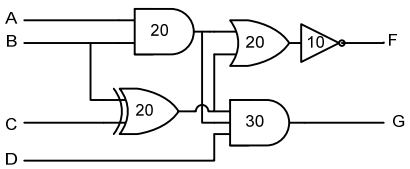
```
module real_and (i1, i2, o);
   input i1, i2;
   output o;

  and #10 real_and (o, i1, i2);
endmodule

module test_real_world();
  reg a, b;
  wire c;

  real_and mine (a, b, c);
  initial begin a = 0; b =0; # 60 a = 1; #60 b=1; #60 a=0; end
 endmodule
```

στο οποίο έχουμε προσάψει 10 χρονικές στιγμές χρονοκαθυστέρηση στη λειτουργία μιας πύλης AND. Οταν εξομοιώσουμε αυτό το κώδικα θα μας προκύψουν κυματομορφές σα κι αυτές της παρακάτω εικόνας. Μπορούμε να δούμε ότι παρότι εφαρμόσαμε 0 στις εισόδους της AND τη χρονική στιγμή 0 της εξομοίωσης, η έξοδος πήρε τη σωστή έξοδο 10 χρονικές στιγμές μετά, ενώ ήταν σε απροσδιόριστη κατάσταση στο χρονικό διάστημα 0 έως 10 της εξομοίωσης (ο εξομοιωτής αυτό το υποδεικνύει με κόκκινο χρώμα). Τη χρονική στιγμή 120 εφαρμόζουμε 1 και στις δύο εισόδους της AND. Το αναμενόμενο λογικό 1 στην έξοδο εμφανίζεται ωστόσο 10 χρονικές στιγμές μετά.



Μπορεί εύλογα να αναρωτιέστε γιατί βάλαμε 10 σα χρονική καθυστέρηση και όχι κάποια άλλη τιμή. Η απάντηση είναι ότι συνήθως η σύγκριση που ψάχνουμε είναι ποιοτική και όχι ποσοτική. Συνήθως δηλαδή θέλουμε να συγκρίνουμε δύο διαφορετικές υλοποιήσεις μεταξύ τους για το ποια είναι η πιο γρήγορη (ή και με άλλα κριτήρια όπως για παράδειγμα ποια χρησιμοποιεί λιγότερες πύλες ή καταναλώνει λιγότερη ισχύ) και ελάχιστα ενδιαφερόμαστε για την απόλυτη χρονική διαφορά μεταξύ τους. Συνεπώς οποιαδήποτε τυχαία τιμή (διαφορετική από το 0) αρκεί για να μας βοηθήσει στις ποιοτικές συγκρίσεις μας.

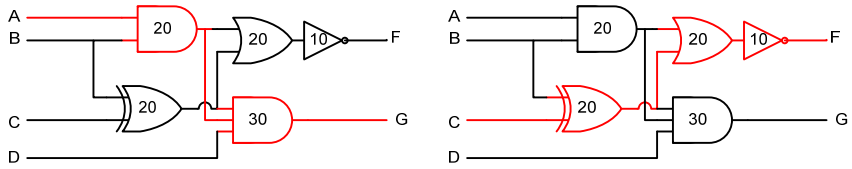
Ας δούμε λίγο περισσότερο το θέμα του καθορισμού της χρονοκαθυστέρησης ενός σχεδιασμού. Ας υποθέσουμε το παρακάτω κύκλωμα :



όπου εντός της κάθε πύλης έχουμε αναγράψει μια αυθαίρετη τιμή χρονοκαθυστέρησης. Σκοπός μας είναι να βρούμε τη χειρότερη χρονοκαθυστέρηση αυτού του κυκλώματος καθώς αυτή θα μας προσδιορίσει και τη μέγιστη συχνότητα λειτουργίας του. Ενδεχόμενα να νομίζετε ότι είναι αρκετό απλά να περιγράψουμε αυτό το κύκλωμα σε Verilog, να

βάλουμε όλους τους πιθανούς συνδυασμούς εισόδων και να καταγράψουμε τη μέγιστη χρονοκαθυστέρηση αποκατάστασης της εξόδου. Δυστυχώς αυτή η μεθοδολογία δεν είναι σωστή, γιατί δυστυχώς πρέπει να λάβουμε υπ'όψιν μας και τη σειρά εφαρμογής των διανυσμάτων εισόδου. Εστω για παράδειγμα ότι εφαρμόζαμε τα εξής διανύσματα εισόδου στο κύκλωμά μας : A=B=D=0, C=1 και A=C=1, B=D=0. Αφού το D είναι 0 και στα 2 διανύσματα, δε θα υπάρξει μετάβαση της εξόδου G. Επίσης, μιας και η έξοδος της XOR πύλης είναι και στα δύο διανύσματα 1, αποτρέπει την ύπαρξη μετάβασης στην έξοδο F. Συνεπώς για τη διαπίστωση της μέγιστης χρονοκαθυστέρησης, απαιτείται πολύ πιο προσεκτική επιλογή των διανυσμάτων μας.

Μελετώντας το σχήμα μας βλέπουμε ότι πιθανή μέγιστη χρονοκαθυστέρηση μπορεί να προκύψει αν χρειαστεί μετάδοση των τιμών των σημάτων σε μονοπάτια όπως αυτά που σημειώνονται με κόκκινο στα παρακάτω σχήματα :

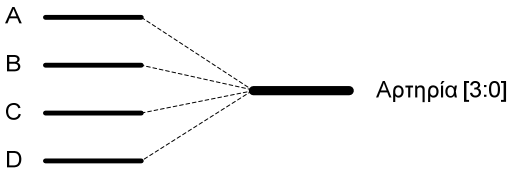


Το ερώτημα συνεπώς που προκύπτει είναι πως κατά την εξομοίωση θα εξασφαλίσουμε διάδοση πάνω από τα αντίστοιχα μονοπάτια. Ας δούμε πρώτα το αριστερό σχήμα. Θέλουμε διάδοση πάνω από τη πύλη AND 2 εισόδων, της αλλαγής της τιμής του A. Συνεπώς θα πρέπει το B να είναι στο 1. Η αλλαγή στην έξοδο της AND 2 εισόδων θέλουμε να φτάσει στην έξοδο G, συνεπώς πρέπει όλες οι "πλαινές" είσοδές της να το επιτρέπουν, άρα το D πρέπει να είναι 1 και η έξοδος της XOR στο 1 επίσης. Αφού όμως το B είναι στο 1, για να έχω έξοδο της XOR στο 1 θα πρέπει να βάλω C=0. Αρα για να πετύχω διάδοση πάνω από το μονοπάτι στο αριστερό σχήμα πρέπει να βάλω B=1, C=0, D=1 και μετάβαση στο A.

Ας εξετάσουμε το δεξί σχήμα στη συνέχεια. Το D δε παίζει κανένα ρόλο, άρα είναι αδιάφορη η τιμή του. Η XOR επίσης δε μας θέτει κανένα περιορισμό ως προς τη διάδοση του σήματος. Για διάδοση μέσα από την OR, θα πρέπει η έξοδος της AND να είναι 0. Αρα μπορούμε να χρησιμοποιήσουμε τους συνδυασμούς 0, 0, 0 ή 0, 0, 1 ή 0, 1, 0 ή 0, 1, 1 ή 1, 0, 0 ή 1, 0, 1 στις εισόδους του κυκλώματός μας A, B και D αντίστοιχα για τη μετάδοση της αλλαγής του C στην έξοδο F πάνω από το μονοπάτι του δεξιού σχήματος.

Στη καθημερινότητά μας πολλές φορές ομαδοποιούμε διάφορα αντικείμενα, με σκοπό να διευκολύνουμε την αναφορά μας σε αυτά αλλά και να ορίζουμε πολύ πιο περιεκτικά πράξεις πάνω σε κάθε αντικείμενο του συνόλου. Για παράδειγμα, παρότι κάθε αυτοκίνητο είναι μια ξεχωριστή οντότητα, πολλές φορές τα ομαδοποιούμε με διάφορους τρόπους, όπως για παράδειγμα ανάλογα με τη χρονολογία κατασκευής τους, με το εργοστάσιο κατασκευής τους, το μοντέλο τους κλπ. Οταν μια εταιρεία για λόγους ασφαλείας χρειάζεται να ανακαλέσει μια σειρά οχημάτων για εξέταση, αντί να βγάλει μια πολύ μεγάλη λίστα κωδικών αριθμών κατασκευής, συνήθως εκδίδει μια λιγόλογη ανακοίνωση που καλεί για παράδειγμα για εξέταση ένα συγκεκριμένο μοντέλο.

Η Verilog μας δίνει τη δυνατότητα να διαχειριζόμαστε δυαδικά σήματα είτε ξεχωριστά είτε σα μέλη μιας συλλογής, που ονομάζεται αρτηρία. Για παράδειγμα τα σήματα A, B, C και D του παρακάτω σχήματος μπορούμε είτε να τα διαχειριζόμαστε ξεχωριστά είτε σα μέλη της "Αρτηρίας".



Στη Verilog μια αρτηρία δηλώνεται σαν ένας μονοδιάστατος πίνακας `[x : y]`, όπου η διάσταση (ο αριθμός των σημάτων της αρτηρίας) είναι `x-y+1`. Για την αναφορά σε κάποιο συγκεκριμένο σήμα της αρτηρίας χρησιμοποιούμε τη γραφή `Αρτηρία[z]` όπου `z` το στοιχείο στο οποίο θέλουμε να αναφερθούμε. Επίσης μπρούμε να δημιουργούμε αρτηρίες από απλά καλώδια, χρησιμοποιοώντας το τελεστή της συνένωσης `{ }`. Για παράδειγμα με την εντολή `assign Artiria[3:0] = {A, B, C, D}` συνενώνουμε τα απλά σήματα σε μια αρτηρία. Στον παραπάνω κώδικα το σήμα `A` είναι και το σήμα `Artiria[3]`, το σήμα `B` και το σήμα `Artiria[2]` κ.ο.κ. Προφανώς τη συνένωση τη χρειαζόμαστε ώστε να ορίσουμε περιεκτικά συναρτήσεις πάνω στην ομάδα σημάτων. Στη ψηφιακή σχεδίαση, υπάρχουν δύο κατηγορίες πράξεων πανω στις αρτηρίες :

- Bit-wise πράξεις και
- Reduction πράξεις.

Στις bit-wise πράξεις, λαμβάνουν μέρος δύο αρτηρίες με ίδιο αριθμό σημάτων και παράγουν ως έξοδο μια ίδια αρτηρία. Κάθε σήμα της αρτηρίας εξόδου προκύπτει από την εφαρμογή της πράξης πάνω σε αντίστοιχα σήματα των δύο αρτηριών. Για παράδειγμα ας δούμε τον κώδικα :

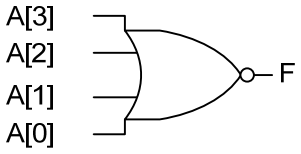
```
module many_xors (a, b, c);
  input  [15:0] a;
  input  [18:3] b;
  output [17:2] c;

  assign c = a^b;
endmodule
```

Με τον κώδικα αυτό περιγράφουμε ένα κύκλωμα, το οποίο δέχεται ως εισόδους 2 αρτηρίες, `a` και `b`, των 16 σημάτων (ισοδύναμα δυαδικών ψηφίων) η κάθε μία. Εξοδος του κυκλώματος είναι η αρτηρία `c` των 16 δυαδικών ψηφίων επίσης. Η εντολή ανάθεσης χρησιμοποιεί το λογικό τελεστή της πράξης XOR. Προσέξτε όμως ότι το χρησιμοποιεί πάνω σε 2 αρτηρίες και αναθέτει τα αποτελέσματά του σε μία άλλη αρτηρία. Η εντολή αυτή λοιπόν είναι συντομογραφία για την εντολή `assign c[17:2] = a[15:0] ^ b[18:3]`, που με τη σειρά της είναι μια περιεκτική περιγραφή του κώδικα :

```
assign c[17] = a[15] ^ b[18];
assign c[16] = a[14] ^ b[17];
…
assign c[2] = a[0] ^ b[3];
```

Δηλαδή, ο παραπάνω κώδικας περιγράφει όχι μία αλλά 16 XOR πύλες, που κάθε μία παίρνει τη μία εισοδό της από την αρτηρία `a`, την άλλη από την αρτηρία `b` και δίνει ως έξοδο ένα σήμα της αρτηρίας `c`.

Σε άλλες περιπτώσεις θέλουμε να φτιάξουμε ένα κύκλωμα για τη διαπίστωση της τιμής μιας αρτηρίας, όπως για παράδειγμα ένα κύκλωμα το οποίο μας δίνει λογικό 1 αν όλα τα σήματα της αρτηρίας μας είναι στο λογικό 0. Αυτά τα κυκλώματα παίρνουν σαν είσοδο μια αρτηρία και βγάζουν στην έξοδο ένα σήμα. Η Verilog για την περιγραφή τέτοιων κυκλωμάτων διαθέτει τις reduction πράξεις. Εστω για παράδειγμα η αρτηρία A[3:0]. Για τη διαπίστωση του αν αυτή έχει όλα τα σήματά της στο 0, θα χρειαζόμασταν ένα κύκλωμα της μορφής :



Προφανώς μπορούμε να γράψουμε κώδικα της μορφής `assign F = ~(A[3] | A[2] | A[1] | A[0]);` για τη περιγραφή του επιθυμητού κυκλώματος. Θυμηθείτε όμως ότι ο πρωταρχικός στόχος των ομαδοποιήσεων είναι να γράφουμε πολύ πιο περιεκτικά τις συναρτήσεις μας. Ετσι χρησιμοποιώντας το τελεστή του λογικού OR σα reduction τελεστή μπορούμε ισοδύναμα να γράψουμε `assign F = ~|A;`

Για να καταλάβουμε λίγο περισσότερο πως αναπτύσσουμε το κώδικά μας χρησιμοποιώντας αρτηρίες, θα καταφύγουμε σε ένα παράδειγμα στο οποίο αναπτύσσουμε έναν συγκριτή των 4 δυαδικών ψηφίων. Ο συγκριτής μας δέχεται σαν είσοδο 2 μη προσημασμένους αριθμούς των 4 δυαδικών ψηφίων έστω X και Y. Εστω ότι ο συγκριτής μας βγάζει στην έξοδο 2 σήματα G, L με την ακόλουθη σημασία :

G	*L*	*Σχέση X και Y*
0	0	X = Y
0	1	X < Y
1	0	X > Y

Η πιο εύκολη λύση είναι να καταφύγουμε σε behavioral κώδικα για τη περιγραφή του συγκριτή μας :

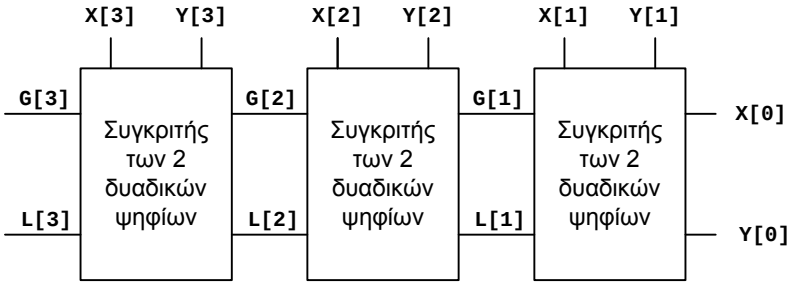
```
module four_bit_comp (X, Y, G, L);
  input  [3:0] X, Y;
  output G, L;

 assign G = (X > Y) ? 1 : 0;
 assign L = (Y > X) ? 1 : 0;
endmodule
```

Σε αυτό το κώδικα χρησιμοποιούμε 2 αρτηρίες εισόδου (`X` και `Y`). Παρότι αυτή η περιγραφή είναι απολύτως σωστή συντακτικά και λογικά, αποκρύπτει σε πολύ μεγάλο βαθμό τη στοχευόμενη υλοποίηση. Θα πρέπει λοιπόν να αποφεύγεται κάθε φορά που θέλουμε κάποια συγκεκριμένη υλοποίηση του κυκλώματός μας.

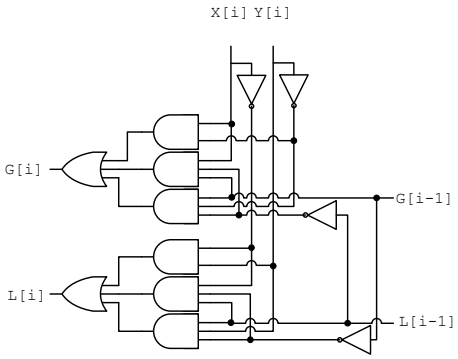
Παρακάτω υλοποιούμε το συγκριτή μας με εναλλακτικούς τρόπους. Ο πρώτος τρόπος προκύπτει θεωρώντας το πρόβλημα σαν αναδρομικό. Εστω ότι κάποιος μας έδινε το συγκριτή των 3 δυαδικών ψηφίων με εξόδους G_2 και L_2 και μας ζήταγε βάσει αυτού να σχεδιάσουμε αυτόν των 4 δυαδικών ψηφίων. Εστω οι έξοδοι του τελευταίου G_3 και L_3. Ο παρακάτω πίνακας αληθείας μας δείχνει πως εξαρτώνται οι G_3 και L_3 από τις G_2 και L_2 καθώς και τα πιο σημαντικά ψηφία των X και Y.

X[3] Y[3] G_2 L_2	G_3 L_3
0 0 0 0	0 0
0 0 0 1	0 1
0 0 1 0	1 0
0 0 1 1	0 0
0 1 0 0	0 1
0 1 0 1	0 1
0 1 1 0	0 1
0 1 1 1	0 1
1 0 0 0	1 0
1 0 0 1	1 0
1 0 1 0	1 0
1 0 1 1	1 0
1 1 0 0	0 0
1 1 0 1	0 1
1 1 1 0	1 0
1 1 1 1	0 0

Με τον ίδιο τρόπο μπορούμε να κατασκευάσουμε το συγκριτή των 3 δυαδικών ψηφίων από αυτόν των 2 δυαδικών ψηφίων, ενώ ο ίδιος πίνακας αληθείας μπορεί να χρησιμοποιηθεί και για το συγκριτή των 2 δυαδικών ψηφίων όπου οι είσοδες είναι οι X[1], Y[1] και X[0], Y[0]. Σχηματικά ο σειριακός συγκριτής μας θα έχει την ακόλουθη δομή :



όπου κάθε συγκριτής των 2 δυαδικών ψηφίων ακολουθεί τον πιο πάνω πίνακα αληθείας, δηλαδή είναι υλοποιημένος ως:



Μπορούμε συνεπώς για το σειριακό συγκριτή μας να γράψουμε τον ακόλουθο κώδικα :

```
module fbc_serial (X, Y, g, l);
  input  [3:0] X, Y;
  output       g, l;

  wire [3:0] G, L;

  assign g = G[3];
  assign l = L[3];

  assign G[3:0] = (X[3:0] & ~Y[3:0]) | (X[3:0] & {G[2:0], X[0]} & ~{L[2:0], Y[0]}) | (~Y[3:0] &
                 {G[2:0], X[0]} & ~{L[2:0], Y[0]});
  assign L[3:0] = (~X[3:0] & Y[3:0]) | (~X[3:0] & ~{G[2:0], X[0]} & {L[2:0], Y[0]}) | (Y[3:0] &
                 ~{G[2:0], X[0]} & {L[2:0], Y[0]});
endmodule
```

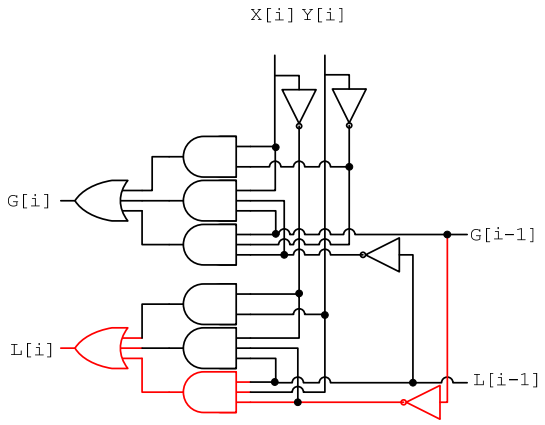
Αξίζει να δούμε με περισσότερη προσοχή τη προτελευταία εντολή του module. Ολες οι λογικές πράξεις που περιγράφονται με την εντολή αυτή είναι bit-wise πράξεις πάνω σε αρτηρίες των 4 δυαδικών ψηφίων. Η πράξη `X[3:0] & ~Y[3:0]` περιγράφει όλες τις πύλες AND 2 εισόδων σα κι αυτή που βρίσκεται υψηλότερα στο πιο πάνω σχήμα. Παρατηρείστε ότι η αναδρομή περιγράφεται χρησιμοποιώντας για τον υπολογισμό του `G[3]` το `G[2]`, `γι'` αυτόν του `G[2]` το `G[1]` κοκ, μέχρι που στη λιγότερο σημαντική θέση χρησιμοποιούμε τα `X[0]` και `Y[0]`. Τέλος ο τελεστής της συνένωσης χρησιμοποιείται για τη δημιουργία αρτηριών των 4 δυαδικών ψηφίων μεταξύ σημάτων `G` και `X` καθώς και σημάτων `L` και `Y`. Για την εξομοίωση των παραπάνω κωδίκων μπορούμε να χρησιμοποιήσουμε τη συνάρτηση `$random` της Verilog για να γεννήσουμε τυχαίες τιμές στις εισόδους μας :

```
module testfbc ();
  reg [3:0] X, Y;
  wire      G1, L1, G2, L2;

four_bit_comp i0 (X, Y, G1, L1);
fbc_serial     i1 (X, Y, G2, L2);

initial begin X=$random; Y=$random; end
always #40 begin X=$random; Y=$random; end
endmodule
```

ΠΑΡΑΔΟΤΕΟ 1 : (χρησιμοποιείστε το αρχείο 6_1.v) Αφού εκτελέσετε εξομοίωση του παραπάνω κώδικα για 1500 χρονικές στιγμές, δώστε ένα screenshot των κυματομορφών εξομοίωσης. Θα πρέπει να συμπεριλάβετε όλα τα σήματα του module testfbc.

Σκοπός του υπολοίπου της άσκησης είναι να προσπαθήσουμε να καταλάβουμε πόσο καλός είναι ο σειριακός τρόπος υλοποίησης του συγκριτή μας. Για την εκτίμηση της απόδοσής του, θα πρέπει να ορίσουμε κάποιο μέτρο σύγκρισης. Υποθέτουμε ότι το μέτρο αυτό είναι η ταχύτητα εκτέλεσης της πράξης της σύγκρισης.

Ας κάνουμε συνεπώς μια αυθαίρετη υπόθεση, ότι δηλαδή κάθε αντιστροφέας, πύλη 2 εισόδων, πύλη 3 εισόδων έχουν αντίστοιχα καθυστερήσεις 10, 20 και 30 χρονικών μονάδων. Οι τιμές αυτές παρότι τυχαίες ποσοτικά ακολουθούν τη γενική παρατήρηση ότι όσο μεγαλώνει ο αριθμός των εισόδων μιας πύλης μεγαλώνει και η καθυστέρησή της. Για να βρούμε πόσο αποδοτικός είναι ο σειριακός συγκριτής μας θα πρέπει πρώτα να έχουμε μια άποψη του πόσο αποδοτικό είναι το κύταρο που χρησιμοποιούμε γι' αυτόν, δηλαδή ο συγκριτής των 2 δυαδικών ψηφίων. Στο σχήμα του συγκριτή των 2 δυαδικών ψηφίων έχουμε με κόκκινο χρώμα σημειώσει ένα από τα χειρότερα σε χρόνο μονοπάτια διάδοσης των σημάτων από τις εισόδους προς τις εξόδους του συγκριτή. Το μονοπάτι αυτό (μαζί με άλλα) αναμένουμε να εμφανίσει χρονοκαθυστέρηση 70 χρονικών μονάδων (10 λόγω του αντιστροφέα, 30 λόγω της AND 3 εισόδων και άλλες 30 λόγω της OR 3 εισόδων). Για να το διαπιστώσουμε αυτό χρησιμοποιώντας Verilog, πρέπει πέρα από το να περιγράψουμε το κύκλωμά μας χρησιμοποιώντας πύλες με χρονοκαθυστερήσεις, στην εξομοίωση να "εγκαθιδρύσουμε" αυτό το μονοπάτι, δηλαδή όντως η διάδοση του σήματός μας κατά την εξομοίωση να γίνει πάνω από αυτό το μονοπάτι.



Θα πρέπει δηλαδή το αρχικό και το τελικό μας διάνυσμα εισόδου να είναι έτσι διαμορφωμένα ώστε μια μετάβαση που ξεκινά από το `G[i-1]` να φτάσει στην έξοδο ***μόνο*** από το σημειωμένο δρόμο. Θα πρέπει συνεπώς να βάλουμε `L[i-1]= 1, Y[i] = 1` και `X[i]=1`. Παρακάτω φαίνεται ο κώδικας για τη διαπίστωση της χρονικής καθυστέρησης αυτού του μονοπατιού :

```
module two_bit_comp(Xi, Yi, G_i_1, L_i_1, Gi, Li);
  input  Xi, Yi, G_i_1, L_i_1;
  output Gi, Li;
```

```
  not #10 inv0 (notXi,    Xi);
  not #10 inv1 (notYi,    Yi);
  not #10 inv2 (notG_i_1, G_i_1);
  not #10 inv3 (notL_i_1, L_i_1);

  and #20 and2_0 (Gi_0, Xi, notYi);
  and #20 and2_1 (Li_0, notXi, Yi);

  and #30 and3_0 (Gi_1,    Xi, G_i_1, notL_i_1);
  and #30 and3_1 (Gi_2, notYi, G_i_1, notL_i_1);
  and #30 and3_2 (Li_1, notXi, L_i_1, notG_i_1);
  and #30 and3_3 (Li_2,    Yi, L_i_1, notG_i_1);

  or  #30 or3_0 (Gi, Gi_0, Gi_1, Gi_2);
  or  #30 or3_1 (Li, Li_0, Li_1, Li_2);
endmodule

module testdelay();
  reg  Xi, Yi, G_i_1, L_i_1;
  wire Gi, Li;

two_bit_comp i0 (Xi, Yi, G_i_1, L_i_1, Gi, Li);
initial
  begin
        Xi=1; Yi=1; G_i_1=1; L_i_1=1;
    #100 G_i_1=0;
  end
endmodule
```

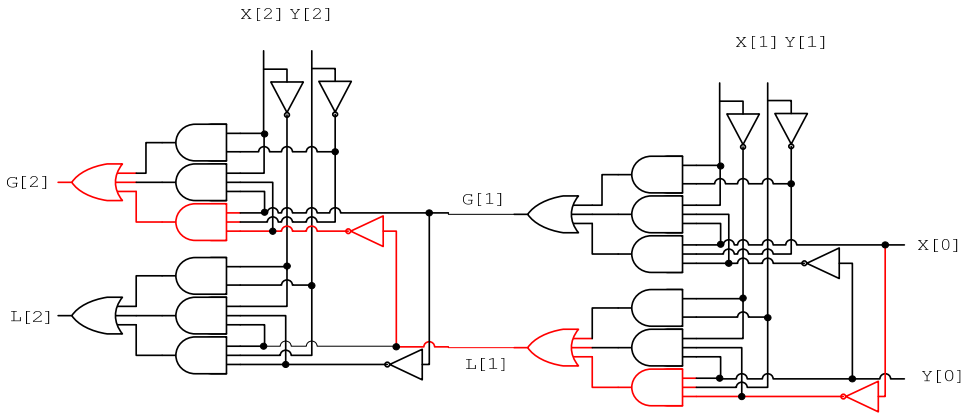
ΠΑΡΑΔΟΤΕΟ 2 : (χρησιμοποιείστε το αρχείο 6_2.v) Αφού εκτελέσετε εξομοίωση του παραπάνω κώδικα για 300 χρονικές στιγμές, δώστε ένα screenshot των κυματομορφών εξομοίωσης. Στο παράθυρο των κυματομορφών προσθέστε 2 δείκτες χρονικών στιγμών (cursors) και υποδείξτε με αυτούς τη χρονοκαθυστέρηση που μετράτε. Στο screenshot πρέπει να φαίνονται όλα τα σήματα του testdelay καθώς και οι cursors και η χρονική τους διαφορά.

Μέρος 2 : Ζητούμενα

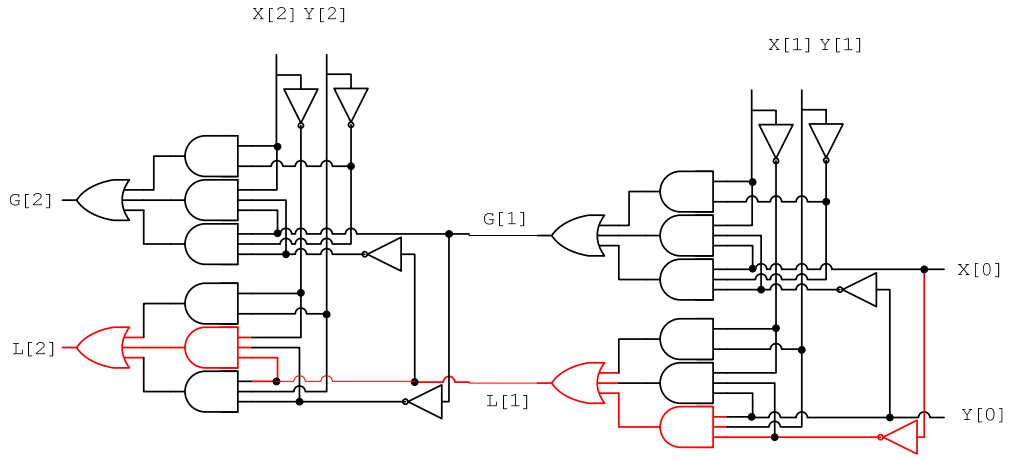
Υπολογίστε τη καθυστέρηση ενός σειριακού συγκριτή λ δυαδικών ψηφίων ως συνάρτηση των υποθέσεων που κάναμε για τη καθυστέρηση του συγκριτή των 2 δυαδικών ψηφίων. Προτείνετε κάποια γρηγορότερη λύση, χρησιμοποιώντας ως παράδειγμα το συγκριτή των 4 δυαδικών ψηφίων. Υπολογίστε τη καθυστέρηση ενός συγκριτή των 2^{κ} δυαδικών ψηφίων που φτιάχνεται σύμφωνα με τη γρήγορη λύση σας σα συνάρτηση των υποθέσεων που κάναμε για τη καθυστέρηση του συγκριτή των 2 δυαδικών ψηφίων.

Μέρος 3 : Ενδεικτική Λύση

Οπως είδαμε παραπάνω ο συγκριτής των 2 δυαδικών ψηφίων με τις αυθαίρετες παραδοχές μας εμφανίζει μια καθυστέρηση των 70 χρονικών μονάδων εξομοίωσης. Ας χρησιμοποιήσουμε 2 τέτοιους για να φτιάξουμε το σειριακό συγκριτή των 3 δυαδικών ψηφίων :



Παρατηρούμε πως και στο 2ο συγκριτή υπάρχει ένα μονοπάτι των 70 χρονικών μονάδων καθυστέρησης το οποίο οδηγείται από την έξοδο του μονοπατιού του 1ου συγκριτή (δεν είναι το μόνο μονοπάτι, απλά είναι ένα από αυτά). Συνεπώς αν μπορέσουμε να ενεργοποιήσουμε αυτό το μονοπάτι, θα πρέπει να μετρήσουμε τη σωστή έξοδό του μετά από 140 χρονικές στιγμές. Για την ενεργοποίηση αυτού του μονοπατιού θα πρέπει να βάλουμε μια μετάβαση στο `X[0]` και να την διαδώσουμε μέχρι το `G[2]`. Δυστυχώς όμως κάτι τέτοιο είναι αδύνατο, όπως θα διαπιστώσετε προσπαθώντας να βάλετε τιμές στην είσοδο ώστε να εγκαθιδρύεται διάδοση σήματος πάνω από αυτό το μονοπάτι ! Αρα παρότι αυτό το μονοπάτι υπάρχει στη φυσική υλοποίηση του σχεδιασμού, ποτέ δε πρόκειται να ενεργοποιηθεί (non sensitizable). Λόγω της συμμετρικότητας του σχήματος, το ίδιο ισχύει για κάθε μονοπάτι που ξεκινά από το `Y[0]` και καταλήγει μέσω του `G[1]` στο `L[2]`. Συμπεραίνουμε λοιπόν ότι ο σχεδιασμός μας θα έχει μικρότερη χρονοκαθυστέρηση από 140 χρονικές μονάδες. Το μονοπάτι που φαίνεται παρακάτω :



είναι από τα επόμενα υποψήφια με καθυστέρηση 130 χρονικών μονάδων. Το μονοπάτι αυτό μπορεί να εγκαθιδρυθεί βάζοντας `X[2]=Y[2]=0` και `X[1]=Y[1]=Y[0]=1`. Ο κώδικας Verilog που επιβεβαιώνει κάτι τέτοιο είναι ο ακόλουθος :

```
module three_bit_comp (X, Y, G, L);
  input  [2:0] X, Y;
  output G, L;

 two_bit_comp i0 (X[1], Y[1], X[0], Y[0], G1, L1);
 two_bit_comp i1 (X[2], Y[2],   G1,   L1,  G,  L);
endmodule
module testdelay3();
  reg  [2:0] X, Y;
  wire G, L;

three_bit_comp i0 (X, Y, G, L);
 initial
  begin
         X=3; Y=3;
    #500 X=2;
  end
endmodule
```

Παρατηρούμε συνεπώς πως ο σειριακός συγκριτής των 3 δυαδικών ψηφίων παρουσιάζει μια καθυστέρηση 60 χρονικών μονάδων μεγαλύτερη από αυτήν του συγκριτή των 2 δυαδικών ψηφίων. Είναι προφανές λόγω της συμμετρίας του σχήματος ότι όσους συγκριτές και αν προστεθούν επιπλέον, η έξοδος L του ενός δε μπορεί να οδηγήσει την έξοδο G του επόμενου λόγω του ότι δεν υπάρχει μονοπάτι που να μπορεί να εγκαθιδρυθεί ενώ μπορεί να οδηγήσει με 60 χρονικές μονάδες την έξοδο L του επόμενου. Αρα ο λ-bit σειριακός αθροιστής θα έχει μια μέγιστη καθυστέρηση 60 x (λ-2) + 70. Παρατηρούμε δηλαδή μια ***γραμμική*** αύξηση της καθυστέρησης σε σχέση με το μήκος του συγκριτή. Επίσης λόγω της συμμετρίας έχουμε και το διάνυσμα εισόδου το οποίο μας αναδεικνύει αυτή τη χρονοκαθυστέρηση : Y=000...011 και αρχικά X=000...011 και μετά X=000...010.

Προς επιβεβαίωση των παραπάνω ας γράψουμε το κώδικα για τον σειριακό συγκριτή των 8 δυαδικών ψηφίων :

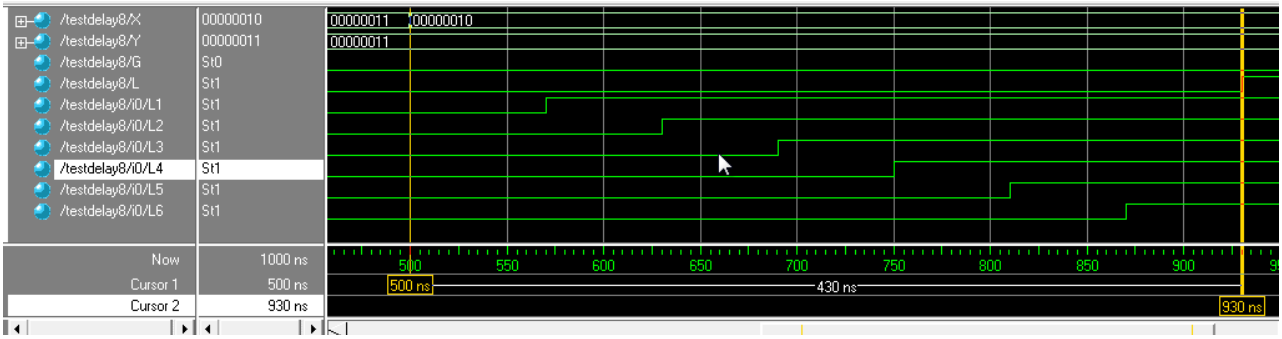
```
module eight_bit_comp (X, Y, G, L);
  input  [7:0] X, Y;
  output G, L;

  two_bit_comp i0 (X[1], Y[1], X[0], Y[0], G1, L1);
  two_bit_comp i1 (X[2], Y[2],   G1,   L1, G2, L2);
  two_bit_comp i2 (X[3], Y[3],   G2,   L2, G3, L3);
  two_bit_comp i3 (X[4], Y[4],   G3,   L3, G4, L4);
  two_bit_comp i4 (X[5], Y[5],   G4,   L4, G5, L5);
  two_bit_comp i5 (X[6], Y[6],   G5,   L5, G6, L6);
```

```
  two_bit_comp i6 (X[7], Y[7],   G6,   L6,  G,  L);
endmodule
```

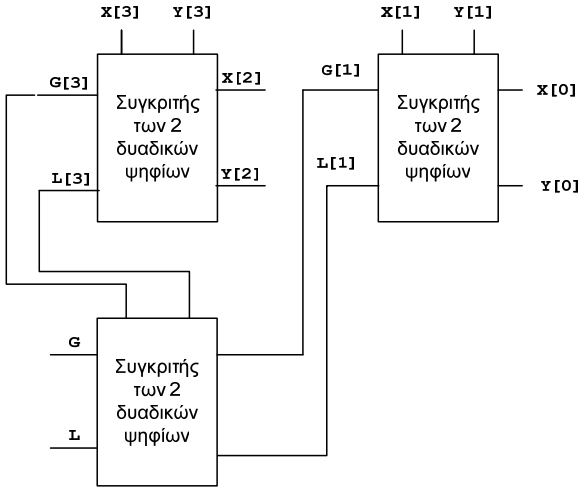
και το testbench :

```
module testdelay8();
  reg  [7:0] X, Y;
  wire G, L;

eight_bit_comp i0 (X, Y, G, L);
 initial
  begin
         X=3; Y=3;
    #500 X=2;
  end
endmodule
```

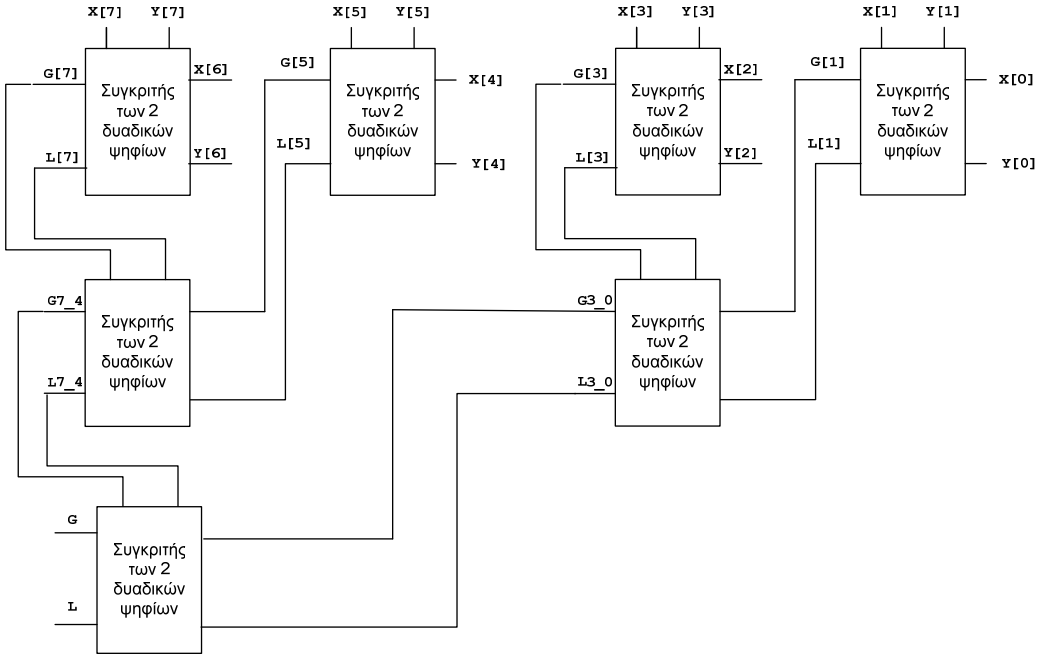
Στο παρακάτω παράθυρο κυματομορφών μετράμε χρονική καθυστέρηση 430 χρονικών μονάδων, σύμφωνη με τη θεωρητικά αναμενόμενη τιμή. Στο παράθυρο αυτό έχουμε συμπεριλάβει και τα σήματα εξόδου των συγκριτών `L1` έως `L6`. Η αλλαγή του πρώτου μετά από 70 χρονικές στιγμές προκαλεί την διαδοχική αλλαγή κάθε επόμενου κάθε επιπλέον 60 χρονικές στιγμές, όπως ακριβώς προβλέψαμε θεωρητικά.



Ας δούμε τώρα πως μπορούμε να φτιάξουμε ένα πιο γρήγορο συγκριτή. Στο παρακάτω σχήμα φαίνεται μια διαφορετική ***αρχιτεκτονική*** για το συγκριτή των 4 δυαδικών ψηφίων. (Ο όρος αρχιτεκτονική χρησιμοποιείται στην επιστήμη μας σε διάφορα επίπεδα των υπολογιστικών συστημάτων. Εδώ τον χρησιμοποιούμε για αρχιτεκτονική κυκλωμάτων).



Παρατηρείστε ότι σε σχέση με το σειριακό συγκριτή δεν έχουμε καθόλου επιπλέον υλικό. Και οι 2 συγκριτές φτιάχνονται από 3 συγκριτές των 2 δυαδικών ψηφίων. Η διαφορά του παραπάνω συγκριτή με το σειριακό έγκειται στο ότι οι 2 πάνω συγκριτές των 2 δυαδικών ψηφίων λειτουργούν ***παράλληλα*** ! Κανείς τους δε περιμένει το αποτέλεσμα του άλλου. Τα αποτελέσματα αυτών συγκεντρώνονται στον κάτω συγκριτή 2 δυαδικών ψηφίων, ο οποίος παράγει το τελικό αποτέλεσμα. Ας προσπαθήσουμε να κάνουμε μια εκτίμηση της καθυστέρησης αυτού του νέου σχήματος. Αφού οι δύο πάνω συγκριτές λειτουργούν παράλληλα, τα αποτελέσματά τους `G[3]`, `L[3]`, `G[1]` και `L[1]` θα είναι διαθέσιμα μετά το πολύ 70 χρονικές στιγμές. Ο συγκεντρωτικός συγριτής θα βγάλει τα αποτελέσματά του μετά από 60 επιπλέον χρονικές στιγμές (και πάλι δε μπορεί να εγκαθιδρυθούν τα μονοπάτια των 70 χρονικών στιγμών καθυστέρησης). Συνεπώς η νέα μας αρχιτεκτονική χρειάζεται 130 χρονικές στιγμές για να βγάλει το αποτέλεσμά της έναντι 190 της σειριακής λύσης, δηλαδή προσφέρει μια αύξηση της απόδοσης κατά 31,5% (!) στη περίπτωση συγκριτή των 4 δυαδικών ψηφίων χωρίς να απαιτεί καθόλου επιπλέον υλικό ! Καθόλου άσχημα !



Η ίδια μεθοδολογία μπορεί να εφαρμοστεί και σε μεγαλύτερους συγκριτές. Αυτός των 8 δυαδικών ψηφίων παρουσιάζεται στο παραπάνω σχήμα. Αποτελείται από 4 συγκριτές που συγκρίνουν δυαδικά ψηφία των εντέλων, 2 συγκεντρωτικούς ($2^{ου}$ επιπέδου) που αποφαίνονται για το τι ισχύει στη κάθε τετράδα δυαδικών ψηφίων (`G7_4, L7_4, G3_0, L3_0`) και τον τελικό που από τις αποκρίσεις των 4άδων μας δίνει το τελικό αποτέλεσμα. Η παραλληλία της εκτέλεσης συνεπώς επεκτείνεται και στο $2^{ο}$ επίπεδο πέρα από το $1^{ο}$. Ο συγκριτής αυτός αναμένουμε να έχει μια καθυστέρηση 70 χρονικών μονάδων από το $1^{ο}$ επίπεδο και 60 χρονικών μονάδων από καθένα από τα $2^{ο}$ και $3^{ο}$ επίπεδο, δηλαδή συνολικά 190 μονάδων, έναντι 430 της σειριακής λύσης ! Επίσης και σε αυτή τη περίπτωση δε παρατηρείται καμμία αύξηση στο υλικό που χρειαζόμαστε (7 συνολικά συγκριτές των 2 δυαδικών ψηφίων). Ο συγκριτής αυτός περιγράφεται από τον ακόλουθο κώδικα Verilog. Μαζί παρέχεται και ένα testbench που αναδεικνύει τη μέγιστη καθυστέρησή του.

```
module eight_bit_parallel (X, Y, G, L);
  input  [7:0] X, Y;
  output G, L;

// 1ο επίπεδο
  two_bit_comp i0 (X[1], Y[1], X[0], Y[0], G1, L1);
  two_bit_comp i1 (X[3], Y[3], X[2], Y[2], G3, L3);
  two_bit_comp i2 (X[5], Y[5], X[4], Y[4], G5, L5);
  two_bit_comp i3 (X[7], Y[7], X[6], Y[6], G7, L7);
// 2ο επίπεδο
  two_bit_comp i4 (G7, L7, G5, L5, G7_4, L7_4);
  two_bit_comp i5 (G3, L3, G1, L1, G3_0, L3_0);
//3ο επίπεδο
  two_bit_comp i6 (G7_4, L7_4, G3_0, L3_0, G,  L);
endmodule

module test_parallel_8();
  reg  [7:0] X, Y;
  wire G, L;

eight_bit_parallel i0 (X, Y, G, L);
 initial
  begin
         X=3; Y=3;
    #500 X=2;
  end
endmodule
```

Γενικεύοντας τα παραπάνω, μπορούμε να πούμε ότι αυτή η "παράλληλη" αρχιτεκτονική, απαιτεί για ένα συγκριτή 2^{κ} δυαδικών ψηφίων 2^{κ}-1 συγκριτές των 2 δυαδικών ψηφίων, οργανωμένους σε κ επίπεδα όπου στο επίπεδο i, με $0 \le i \le$ κ-1 υπάρχουν $2^{\kappa}/2^{i+1}$ συγκριτές των 2 δυαδικών ψηφίων. Η καθυστέρηση αυτού του συγκριτή είναι δε 70+(κ-1)*60 και μιας εδώ το μέγεθος του συγκριτή είναι 2^{κ}, δηλαδή το $\kappa=\log_2$*μεγέθους*, συμπεραίνουμε ότι αντίθετα με τη προηγούμενη περίπτωση η καθυστέρηση αυξάνεται ***λογαριθμικά*** με το μέγεθος του συγκριτή.

ΠΑΡΑΔΟΤΕΟ 3 : ***(χρησιμοποιείστε το αρχείο 6_3.v)***

- ***Ο κώδικας του αρχείου 6_3.v χρησιμοποιεί τυχαία κοινά διανύσματα στις εισόδους του σειριακού και του παράλληλου συγκριτή των 8 δυαδικών ψηφίων. Κάθε νέο διάνυσμα επιβάλλεται κάθε 1000 χρονικές στιγμές εξομοίωσης. Εκτελέσετε εξομοίωση του παραπάνω κώδικα για 40000 χρονικές στιγμές και επιλέξτε ένα παράθυρο χρόνου της αρεσκείας σας στο οποίο να φαίνονται οι διαφορές του χρόνου εκτέλεσης της σύγκρισης από τις 2 υλοποιήσεις.***
- ***Υπάρχει περίπτωση για κάποιο διάνυσμα εισόδου ο σειριακός συγκριτής να βγάλει πιο γρήγορα το αποτέλεσμά του ? Αν ναι, αλλάξτε το testbench του 6_3.v, ώστε να επιβάλλει το επιθυμητό διάνυσμα, εξομοιώστε από κοινού και τους 2 συγκριτές και υποδείξτε την ορθότητα του ισχυρισμού σας μέσω screenshot των κυματομορφών. Επισυνάψτε το νέο testbench.***