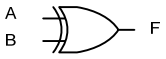
ΑΣΚΗΣΗ 4

Συνοδευτικά αρχεία : 4_1.v, 4_2.v, 4_3.v

Μέρος 1 : Εισαγωγή

Σκοπός της 4ης άσκησης είναι να δείτε εναλλακτικούς τρόπους για τη περιγραφή συνδυαστικών κυκλωμάτων χρησιμοποιώντας τη γλώσσα Verilog. Θα πρέπει τελειώνοντας αυτή την άσκηση να μπορείτε βλέποντας ένα κώδικα για συνδυαστικά κυκλώματα να καταλαβαίνετε αν αυτός είναι δομικός ή βάσει της περιγραφής.

Ας υποθέσουμε βλέποντας ένα πολύ απλό κύκλωμα, δηλαδή τη πύλη XOR του παρακάτω σχήματος.



Η πύλη παίρνει ως εισόδους τα A και B και βγάζει στην έξοδο F 1 αν ο αριθμός των εισόδων που είναι στο λογικό 1 είναι περιττός. Στην αντίθετη περίπτωση η έξοδος είναι στο 0.

Μέχρι ώρας έχετε δει μόνο το δομικό τρόπο για τη περιγραφή αυτού του κυκλώματος, δηλαδή κώδικα σα το παρακάτω :

```
module myxorgate1 (A, B, F);
  input  A, B;
  output F;

  xor i0 (F, A, B);
endmodule
```

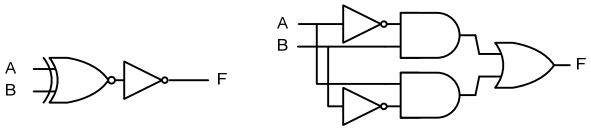
Ο κώδικας αυτός ονομάζεται δομικός καθώς περιγράφουμε το σχεδιασμό μας σα διασύνδεση ήδη υπαρχόντων δομικών στοιχείων (της πύλης XOR στο παράδειγμά μας). Δυστυχώς ο δομικός τρόπος περιγραφής παρότι είναι συνήθως ο πιο κατανοητός δεν είναι αποδοτικός για μεγάλα κυκλώματα. Επίσης έχει και πολλά άλλα μειονεκτήματα, τα οποία θα εξετάσουμε παρακάτω.

Μια εναλλακτική πρόταση είναι να χρησιμοποιήσουμε λογικές συναρτήσεις για τη περιγραφή ενός συνδυαστικού κυκλώματος. Η Verilog γι' αυτό διαθέτει τις εντολές continuous assignments και τις λογικές συναρτήσεις AND, OR, NOT, NAND, NOR, XOR, XNOR ή τις αντίστοιχες συντομογραφίες τους : &, |, ~, ~&, ~|, ^ και ~^. Η προτεραιότητα αυτών είναι η γνωστή σας από το μάθημα, δηλαδή πρώτα παρενθέσεις, μετά οι αρνήσεις, μετά το AND και την ελάχιστη προτεραιότητα έχει το OR. Για παράδειγμα ο παρακάτω κώδικας περιγράφει επιθυμητό κύκλωμα της εικόνας χρησιμοποιώντας τη λογική των συναρτήσεων :

```
module myxorgate2 (A, B, F);
  input  A, B;
  output F;

  assign F=A ^ B;
endmodule
```

Στο κώδικα χρησιμοποιούμε την continuous assignment εντολή (σε ελεύθερη μετάφραση : εντολή διαρκούς ανάθεσης) `assign F = A XOR B;` για να περιγράψουμε το κύκλωμά μας. Αξίζει να σημειώσουμε ότι αυτή η εντολή περιγράφει **έμμεσα και το κύκλωμά μας**. Αυτό σημαίνει ότι στη πραγματικότητα περιγράφει μια ολόκληρη ομάδα κυκλωμάτων όπως για παράδειγμα αυτά των παρακάτω σχημάτων, τα οποία έχουν την ίδια λογική εξίσωση με το επιθυμητό κύκλωμα, δηλαδή είναι κυκλώματα ισοδύναμων συναρτήσεων.



Περιγράφοντας το κύκλωμά μας με τη μέθοδο των λογικών εξισώσεων έχουμε περάσει σε ένα νέο επίπεδο αφαίρεσης. Στο επίπεδο αυτό δε μας ενδιαφέρει η ακριβής υλοποίηση του κυκλώματός μας από συγκεκριμένα δομικά κυκλώματα, παρά μόνο να επιτελεί τη σωστή λογική του λειτουργία. Στη πραγματικότητα έτσι έχουμε μια οικογένεια ισοδύναμων κυκλωμάτων και αργότερα θα επιλέξουμε ποια υλοποίηση αυτών ταιριάζει καλύτερα στην εφαρμογή μας.

Ας δούμε μερικά ακόμη παραδείγματα εντολών διαρκούς ανάθεσης πριν αναλύσουμε τις απαιτήσεις της εντολής :

- `assign X = A AND B OR C` ή ισοδύναμα `assign X = A & B | C` ή ισοδύναμα `assign X = (A AND B) OR C` που περιγράφει την οικογένεια των κυκλωμάτων που δίνουν 1 στην έξοδό τους όταν το C είναι 1 ή και το A και το B είναι 1.

- `assign X = ~((A ^ B) | C)` που περιγράφει την οικογένεια των κυκλωμάτων που δίνουν 0 στην έξοδό τους όταν το C είναι 1 ή όταν υπάρχει περιττός αριθμός από 1 στα σήματα A και B.
- `assign X = ~A | B | C` που περιγράφει την οικογένεια των κυκλωμάτων που δίνουν 1 στην έξοδό τους όταν το B ή το C είναι 1 ή όταν A =0.
- `assign X = ~(A | B | C)` που περιγράφει όλα τα κυκλώματα που είναι ισοδύναμα με μια πύλη NOR 3 εισόδων με εισόδους A, B και C.

Για τη σωστή σύνταξη της εντολής `assign` πρέπει η μεταβλητή που είναι στο αριστερό μέρος της ανάθεσης **να μην είναι δηλωμένη ως reg**. Αυτό σημαίνει ότι αν θέλετε να τη δηλώσετε πρέπει να τη δηλώσετε ως καλώδιο (`wire`). Μπορείτε ωστόσο να μη δηλώσετε καθόλου τη μεταβλητή και η Verilog θα τη θεωρήσει αυτόματα ως καλώδιο. Ας δούμε μερικά παραδείγματα στα οποία περιγράφουμε εκ νέου κύκλωμα ισοδύναμο με μια πύλη XOR :

```
module myxorgate3 (A, B, F);
  input  A, B;
  output F;
  wire   S1, S2;

  assign S1 = A & ~B;
  assign S2 = ~A & B;
  assign F = S1 | S2;
endmodule
```

Ο κώδικας αυτός είναι εντελώς ισοδύναμος με τον :

```
module myxorgate4 (A, B, F);
  input  A, B;
  output F;

  assign S1 = A & ~B;
  assign S2 = ~A & B;
  assign F = S1 | S2;
endmodule
```

όπου οι τοπικές μας μεταβλητές S1 και S2 δε δηλώνονται, αλλά η Verilog θα τις θεωρήσει ότι είναι καλώδια, σα να τις είχαμε δηλώσει. ***Επίσης, η σειρά αναγραφής αυτών των εντολών δεν έχει καμία σημασία. Θυμηθείτε ότι η Verilog περιγράφει υλικό και στο υλικό δεν ισχύει το ακολουθιακό μοντέλο εκτέλεσης.*** Αν και προφανές αξίζει να σημειωθεί ότι οποιαδήποτε μίξη μεταξύ δομικού και κώδικα λογικών εξισώσεων είναι δυνατή. Δείτε για παράδειγμα τον παρακάτω κώδικα :

```
module myxorgate5 (A, B, F);
  input  A, B;
  output F;

  not i1 (notB, B);
  not i2 (notA, A);
  assign S1 = A & notB;
  assign S2 = notA & B;
  or o1 (F, S1, S2);
endmodule
```

Ενδεχόμενα εδώ και μερικά λεπτά καθώς διαβάζατε την άσκηση να αναρωτιέστε μήπως υπάρχει και επίπεδο ακόμα μεγαλύτερης αφαίρεσης, δηλαδή ένα επίπεδο στο οποίο μπορούμε να περιγράφουμε υλικό χωρίς να ξέρουμε ούτε καν τις λογικές συναρτήσεις που αυτό επιτελεί. Η Verilog υποστηρίζει άλλο ένα υψηλότερο επίπεδο αφαίρεσης, το επίπεδο περιγραφής κυκλώματος με βάση τη συμπεριφορά (behavioral). Στο επίπεδο αυτό περιγράφουμε το υλικό μας χρησιμοποιώντας ως μόνη γνώση το πως θέλουμε να συμπεριφέρεται. Για παράδειγμα θέλουμε στη περίπτωση του παραδείγματός μας να βγάζει λογικό 1 όταν ο αριθμός των εισόδων που είναι στο λογικό 1 είναι περιττός και λογικό 0 αλλιώς. Ας δούμε τον παρακάτω κώδικα γραμμένο για το σκοπό αυτό :

```
module myxorgate6 (A, B, F);
  input  A, B;
  output F;
  wire SUM;

  assign SUM = (A + B) % 2;
  assign F = (SUM==1)? 1 : 0;
endmodule
```

Η πρώτη γραμμή διαρκούς ανάθεσης αυτού του κώδικα προσθέτει τις τιμές των A και B, υπολογίζει το υπόλοιπο της διαίρεσης του αθροίσματος ως προς 2 (τελεστής %) και το αποθηκεύει στη μεταβλητή SUM. Η δεύτερη εντολή χρησιμοποιεί έναν ευρέως χρησιμοποιούμενο τελεστή (? :) γνωστό ως ternary operator. Ο τελεστής αυτός ελέγχει την αλήθεια μιας συνθήκης (SUM == 1 στο παράδειγμά μας). Αν αυτή είναι αληθής εκτελεί την ανάθεση με τιμή αυτή που ακολουθεί. Αν αυτή είναι ψευδής εκτελεί την ανάθεση με τη τιμή που ακολουθεί το :. Στη περίπτωσή μας αν η μεταβλητή SUM έχει τιμή 1 θα βάλει το 1 στην F, αλλιώς θα βάλει 0. Είναι προφανές ότι ο συγγραφέας του παραπάνω κώδικα δε χρειάζεται να γνωρίζει τίποτε περί λογικών συναρτήσεων ή πραγματικής υλοποίησης. Παρά ταύτα (όπως άλλωστε θα φανεί και στην εξομοίωση, ο κώδικας αυτός περιγράφει κύκλωμα αντίστοιχο με το ζητούμενο.

Ας δούμε και δύο τελευταίες μορφές behavioral κώδικα για το ίδιο κύκλωμα :

```
module myxorgate7 (A, B, F);
  input  A, B;
  output F;
  wire [1:0] AB;

  assign AB = {A, B};
  assign F = (AB == 0)? 0 :
             (AB == 3)? 0 : 1;
endmodule

module myxorgate8 (A, B, F);
  input  A, B;
  output F;
  reg F;

  always @(A or B)
    case ({A,B})
      0 : F=0;
      1 : F=1;
      2 : F=1;
      3 : F=0;
    endcase
endmodule
```

Στο πρώτο κώδικα με σκοπό να μπορούμε να εξετάζουμε ταυτόχρονα τη τιμή των σημάτων A και B, τα συνενώνουμε σε μια αρτηρία (περισσότερα για αρτηρίες θα δούμε στην άσκηση 6) η οποία ορίζεται με την εντολή `wire [1:0] AB;` Η συνένωση γίνεται με την εντολή `assign AB = {A, B};` όπου ορίζουμε ότι τα δύο σήματα της αρτηρίας θα παίρνουν τις τιμές των A και B. Στην επόμενη εντολή χρησιμοποιούμε επαναληπτικά τον ternary operator για να εξετάσουμε τις τιμές της αρτηρίας. Όταν η αρτηρία έχει τιμή 0 (A = B = 0) ή όταν έχει τιμή 3 (A = B = 1) αναθέτουμε στην έξοδο 0, αλλιώς της αναθέτουμε 1.

Στο τελευταίο κώδικα, χρησιμοποιούμε μια μόνο εντολή την `always`. Συνήθως αυτή χρησιμοποιείται για την επιβολή συνθήκης εκτέλεσης σε ένα block εντολών. Στο παράδειγμά μας όμως υπάρχει μόνο μια εντολή, η `case`. Η εντολή αυτή εκτελείται μόνο όταν υπάρξουν αλλαγές στη λίστα των σημάτων του `always` που αναγράφονται εντός του `@( )`. Συνεπώς η εντολή αυτή θα εκτελεστεί όταν υπάρχουν αλλαγές στα σήματα `A` ή `B`. Εντός του `case` ανάλογα με τη τιμή των A και B εξεταζομένων μαζί ως αρτηρία όπως προηγούμενα γίνεται η ανάθεση της κατάλληλης τιμής στην έξοδο. Παρατηρείστε επίσης ότι στο κώδικα αυτόν υπάρχει η δήλωση της εξόδου ως reg (register). ***Το συντακτικό της Verilog επιβάλλει ότι οποιαδήποτε μεταβλητή χρησιμοποιείται στο αριστερό μέρος αναθέσεων εντός ενός always πρέπει να δηλωθεί ως reg.***

Πριν δούμε κάποιο πιο πολύπλοκο παράδειγμα, ας κάνουμε κοινή εξομοίωση όλων των παραπάνω κωδίκων για να βεβαιωθούμε ότι περιγράφουν ισοδύναμο υλικό. Για την εξομοίωση θα χρησιμοποιήσουμε το παρακάτω κώδικα, που επιβάλλει σε όλες τις παραπάνω περιγραφές κοινές εισόδους. Επίσης ανά 100 χρονικές στιγμές βάζει και ένα νέο διάνυσμα εισόδων.

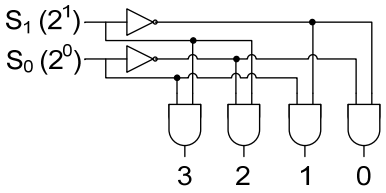
```
module test_all_xor ();
  reg I1, I2;
  wire F1, F2, F3, F4, F5, F6, F7, F8;

  myxorgate1 i1 (I1, I2, F1);
  myxorgate2 i2 (I1, I2, F2);
  myxorgate3 i3 (I1, I2, F3);
  myxorgate4 i4 (I1, I2, F4);
  myxorgate5 i5 (I1, I2, F5);
  myxorgate6 i6 (I1, I2, F6);
  myxorgate7 i7 (I1, I2, F7);
  myxorgate8 i8 (I1, I2, F8);
```

```
  initial begin I1=0; I2=0; end
  initial
    begin
      #100 I1 = 1;
      #100 I2 = 1;
      #100 I1 = 0;
    end
endmodule
```

ΠΑΡΑΔΟΤΕΟ 1 : (χρησιμοποιείστε το αρχείο 4_1.v) Αφού εκτελέσετε εξομοίωση του παραπάνω κώδικα για 500 χρονικές στιγμές, δώστε ένα screenshot των κυματομορφών εξομοίωσης. Θα πρέπει να συμπεριλάβετε όλα τα σήματα του module test_all_xor.

Σα δεύτερο παράδειγμα, ας προσπαθήσουμε να γράψουμε 3 διαφορετικές εκδόσεις κώδικα για έναν αποκωδικοποιητή 2 σε 4 χωρίς είσοδο επίτρεψης (στο μάθημα χρησιμοποιούμε την ορολογία αποκωδικοποιητής και αποπλέκτης για κυκλώματα χωρίς και με είσοδο επίτρεψης αντίστοιχα). Ο αποκωδικοποιητής παίρνει μια δυαδική τιμή στις εισόδους του και ανάλογα με αυτήν ενεργοποιεί μία από τις εξόδους του. Για παράδειγμα όταν η δυαδική τιμή 11 (3_{10}) εμφανιστεί στις εισόδους του η έξοδος 3 ενεργοποιείται. Οι είσοδες του αποκωδικοποιητή έχουν δυαδικά βάρη και οι έξοδοί του είναι αριθμημένες. Στο παρακάτω σχήμα παρουσιάζεται ο αποκωδικοποιητής 2 σε 4 :



Μπορούμε βάσει του σχήματος πολύ εύκολα να γράψουμε δομικό κώδικα :

```
module decoder2_2_4_struct (S1, S0, O0, O1, O2, O3);
  input S1, S0;
  output O0, O1, O2, O3;

  not n0 (S0n, S0);
  not n1 (S1n, S1);
  and a0 (O0, S1n, S0n);
  and a1 (O1, S1n, S0);
  and a2 (O2,  S1, S0n);
  and a3 (O3,  S1, S0);
endmodule
```

Επίσης είναι εξαιρετικά εύκολο να περιγράψουμε τον αποκωδικοποιητή βάσει των λογικών εξισώσεών του.

```
module decoder2_2_4_equat (S1, S0, O0, O1, O2, O3);
  input S1, S0;
  output O0, O1, O2, O3;

  assign O0 = ~S1 & ~S0;
  assign O1 = ~S1 & S0;
  assign O2 = S1 & ~S0;
  assign O3 = S1 & S0;
endmodule
```

Αν θέλουμε μια περιγραφή βάσει της συμπεριφοράς μπορούμε να χρησιμοποιήσουμε εμφωλευμένους ternary operators:

```
module decoder2_2_4_ternary (S1, S0, O0, O1, O2, O3);
  input S1, S0;
  output O0, O1, O2, O3;

  assign {O3, O2, O1, O0} = S1 ? S0 ? 8 : 4 : S0 ? 2 : 1;
endmodule
```

Προσέξτε ότι βάζοντας στην αρτηρία που δημιουργείται από τη συνένωση των O3, O2, O1 και O0 τη τιμή $8_{10} = 1000_2$ στην ουσία ενεργοποιούμε την O3, με 4_{10} την O2 κ.ο.κ.

```
module decoder2_2_4_beh (S1, S0, O0, O1, O2, O3);
  input S1, S0;
  output O0, O1, O2, O3;
  reg O0, O1, O2, O3;
```

```
  always@(S1 or S0)
  case ({S1, S0})
    0 : {O3, O2, O1, O0} = 1;
    1 : {O3, O2, O1, O0} = 2;
    2 : {O3, O2, O1, O0} = 4;
    3 : {O3, O2, O1, O0} = 8;
  endcase
endmodule
```

Τέλος μπορούμε να εξομοιώσουμε όλες αυτές τις περιγραφές βάζοντάς τους κοινά διανύσματα εισόδου :

```
module decoder_test();
  reg S1, S0;
  wire O0_1, O1_1, O2_1, O3_1;
  wire O0_2, O1_2, O2_2, O3_2;
  wire O0_3, O1_3, O2_3, O3_3;
  wire O0_4, O1_4, O2_4, O3_4;

decoder2_2_4_struct  m0 (S1, S0, O0_1, O1_1, O2_1, O3_1);
decoder2_2_4_equat   m1 (S1, S0, O0_2, O1_2, O2_2, O3_2);
decoder2_2_4_ternary m2 (S1, S0, O0_3, O1_3, O2_3, O3_3);
decoder2_2_4_beh     m3 (S1, S0, O0_4, O1_4, O2_4, O3_4);

initial begin S1=0; S0=0; end
initial
  begin
    #100 S0 = 1;
    #100 S1 = 1;
    #100 S0 = 0;
  end
endmodule
```

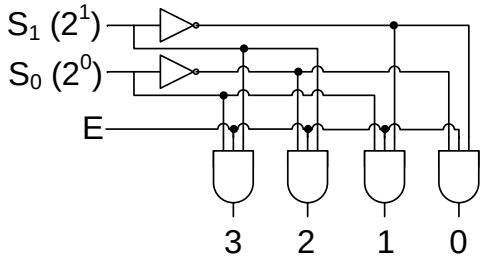
ΠΑΡΑΔΟΤΕΟ 2 : (χρησιμοποιείστε το αρχείο 4_2.v) Αφού εκτελέσετε εξομοίωση του παραπάνω κώδικα για 500 χρονικές στιγμές, δώστε ένα screenshot των κυματομορφών εξομοίωσης. Θα πρέπει να συμπεριλάβετε όλα τα σήματα του module decoder_test και να τα αναδιατάξετε πριν το screenshot έτσι ώστε τα σήματα εξόδου O0 να είναι όλα μαζί, τα σήματα O1 όλα μαζί κ.ο.κ ώστε να διευκολύνετε τη σύγκριση.

Μέρος 2 : Ζητούμενα

Ζητείται να περιγράψετε έναν αποπλέκτη (αποκωδικοποιητή με είσοδο επίτρεψης) 2 σε 4 με δομικό τρόπο, βάσει των λογικών εξισώσεών του και βάσει της συμπεριφοράς. Εκτός των σημάτων επιλογής της εξόδου, ο αποπλέκτης διαθέτει την είσοδο επίτρεψης E. Όταν E=0 όλες οι έξοδες είναι στο 0. Όταν E=1 συμπεριφέρεται όπως ο αποκωδικοποιητής. Εξομοιώστε τις διάφορες περιγραφές και δείξτε ότι λειτουργούν ορθά.

Μέρος 3 : Ενδεικτική Λύση

Το λογικό διάγραμμα του αποπλέκτη φαίνεται στο πιο κάτω σχήμα :



απ' όπου μπορούμε πολύ εύκολα να πάρουμε τη δομική περιγραφή :

```
module dec2_2_4_struct (E, S1, S0, O0, O1, O2, O3);
  input E, S1, S0;
  output O0, O1, O2, O3;

  not n0 (S0n, S0);
  not n1 (S1n, S1);
  and a0 (O0, S1n, S0n, E);
  and a1 (O1, S1n, S0,  E);
  and a2 (O2,  S1, S0n, E);
```

```
  and a3 (O3,  S1, S0,  E);
endmodule
```

και τη περιγραφή βάσει των λογικών εξισώσεων :

```
module dec2_2_4_equat (E, S1, S0, O0, O1, O2, O3);
  input E, S1, S0;
  output O0, O1, O2, O3;

  assign O0 = E & ~S1 & ~S0;
  assign O1 = E & ~S1 & S0;
  assign O2 = E &  S1 & ~S0;
  assign O3 = E &  S1 & S0;
endmodule
```

Για τη περιγραφή βάσει της συμπεριφοράς, θα χρησιμοποιήσουμε τη δομή always :

```
module dec2_2_4_beh (E, S1, S0, O0, O1, O2, O3);
  input E, S1, S0;
  output O0, O1, O2, O3;
  reg O0, O1, O2, O3;

  always@(E or S1 or S0)
    if (E)
      case ({S1, S0})
        0 : {O3, O2, O1, O0} = 1;
        1 : {O3, O2, O1, O0} = 2;
        2 : {O3, O2, O1, O0} = 4;
        3 : {O3, O2, O1, O0} = 8;
      endcase
     else {O3, O2, O1, O0} = 0;
endmodule
```

Ο κώδικας αυτός έχει εμπλουτισμένη συνθήκη εκτέλεσης του always : ο κώδικας εντός του always θα εκτελεστεί και με κάθε αλλαγή του σήματος Ε. Επίσης η μοναδική εντολή του always είναι μια εντολή if. Αν λοιπόν το Ε είναι 1 [if (E)] θα εκτελεστεί ο κώδικας του case που είναι ίδιος με αυτόν του αποκωδικοποιητή. Αλλιώς όλοι οι έξοδοι θα είναι 0.

Για την εξομοίωση χρησιμοποιούμε μια προσέγγιση αντίστοιχη με αυτήν του αποκωδικοποιητή. Ο κώδικας που χρειαζόμαστε θα είναι σα κι αυτόν που ακολουθεί :

```
module test_all_dec();
  reg S1, S0, E;
  wire O0_1, O1_1, O2_1, O3_1;
  wire O0_2, O1_2, O2_2, O3_2;
  wire O0_3, O1_3, O2_3, O3_3;


dec2_2_4_struct m0 (E, S1, S0, O0_1, O1_1, O2_1, O3_1);
dec2_2_4_equat  m1 (E, S1, S0, O0_2, O1_2, O2_2, O3_2);
dec2_2_4_beh    m2 (E, S1, S0, O0_3, O1_3, O2_3, O3_3);

initial begin E=0; S1=0; S0=0; end
initial
  begin
    #100 S0 = 1;
    #100 S1 = 1;
    #100 S0 = 0;
    #100  E = 1;
    #100 S1 = 0;
    #100 S0 = 1;
    #100 S1 = 1;
  end
endmodule
```

ΠΑΡΑΔΟΤΕΟ 3 : (χρησιμοποιείστε το αρχείο 4_3.v) Αφού εκτελέσετε εξομοίωση του παραπάνω κώδικα για 1000 χρονικές στιγμές, δώστε ένα screenshot των κυματομορφών εξομοίωσης. Θα πρέπει να συμπεριλάβετε όλα τα σήματα του module test_all_dec και να τα αναδιατάξετε πριν το screenshot έτσι ώστε τα σήματα εξόδου O0 να είναι όλα μαζί, τα σήματα O1 όλα μαζί κ.ο.κ ώστε να διευκολύνετε τη σύγκριση.

Εχοντας ολοκληρώσει την άσκηση, μπορούμε να σταθούμε λίγο στο τι προσφέρει η περιγραφή υλικού σε υψηλό επίπεδο αφαίρεσης, με τη χρήση δηλαδή λογικών εξισώσεων ή κώδικα που περιγράφει τη συμπεριφορά του

κυκλώματος. Οι περιγραφές αυτές όντας απαλλαγμένες από άχρηστες λεπτομέρειες εξομοιώνονται πολύ πιο γρήγορα (μπορείτε να συγκρίνετε το πόσες γραμμές κώδικα χρειάζεται να εκτελεστούν στο δομικό τρόπο περιγραφής του αποκωδικοποιητή και πόσες με τη χρήση του ternary operator) αλλά ταυτόχρονα μπορούν να υλοποιηθούν από "έξυπνα" εργαλεία με διαφορετικούς τρόπους που ο καθένας προσφέρει τα δικά του χαρακτηριστικά. Άλλη υλοποίηση είναι η γρηγορότερη, άλλη η πιο μικρή και άλλη αυτή που καταναλώνει τη λιγότερη ισχύ. Ο σχεδιαστής είναι απαλλαγμένος από αυτό το φορτίο. Γράφει τον ίδιο κώδικα και αφήνει τα εργαλεία να βρουν την υλοποίηση που ταιριάζει πιο καλά στην εφαρμογή του. Αντίθετα με το δομικό τρόπο θα έπρεπε να υπεισέρχεται σε λεπτομέρειες του στυλ πόσο χώρο καταλαμβάνει μια πύλη AND, πόσο γρήγορη ή αργή είναι κ.ο.κ.

Στη σημερινή βιομηχανία ο δομικός τρόπος περιγραφής χρησιμοποιείται μόνο από εξαιρετικά πεπειραμένους σχεδιαστές και μόνο για εκείνα τα τμήματα υλικού που θέλουμε να υλοποιηθούν με πολύ συγκεκριμένο, προδιαγεγραμμένο τρόπο. Όλα τα υπόλοιπα σχεδιαστικά κομμάτια περιγράφονται βάσει της συμπεριφοράς τους.