Πανεπιστήμιο Ιωαννίνων

Πολυτεχνική Σχολή

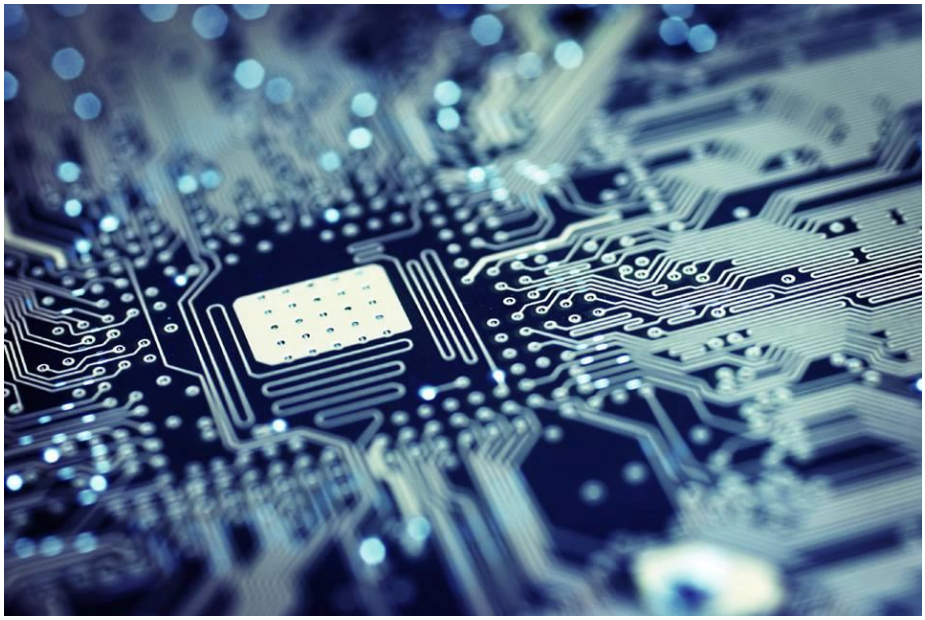
Τμήμα Μηχανικών Η/Υ και Πληροφορικής

Προπτυχιακό Μάθημα: «Δοκιμή και Αξιοπιστία Ηλεκτρονικών Συστημάτων»

Τρίτη Εργαστηριακή Άσκηση

Όνομα Φοιτητή – Α.Μ.:

Γεώργιος Κρομμύδας – 3260



ΙΩΑΝΝΙΝΑ,

2021

Πίνακας περιεχομένων

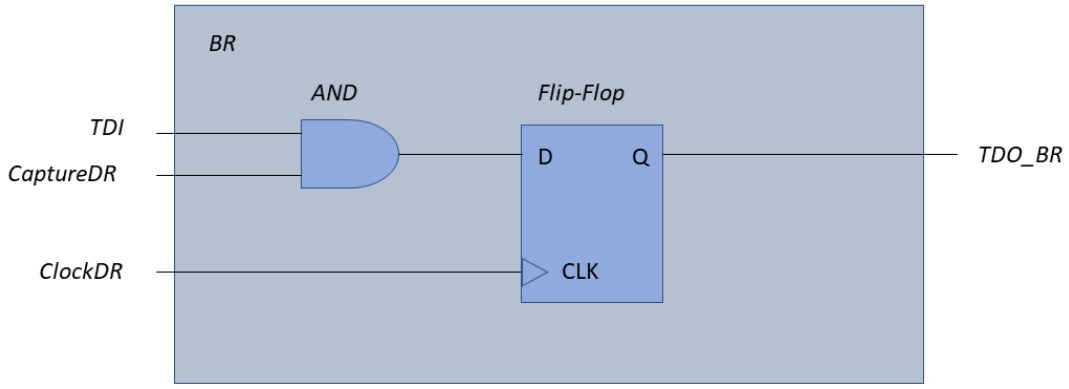
Μέρος – 1ο: Εισαγωγή

Σε αυτή την εργαστηριακή άσκηση είχαμε να υλοποιήσουμε τα δομικά κυκλώματα της JTAG αρχιτεκτονικής. Η JTAG αρχιτεκτονική είναι το πρωτόκολλο IEEE 1149.1 std. το οποίο βασίζεται στην Περιφερειακή Σάρωση(***Boundary Scan Testing***). Γενικά, το JTAG μας προσφέρει την δυνατότητα ελέγχου ολοκληρωμένων συστημάτων. Στην παρούσα εργασία έχουν σχεδιαστεί οι καταχωρητές της JTAG αρχιτεκτονικής (***BCS***, ***BR*** και ***IR***) και η μηχανή πεπερασμένων καταστάσεων (***Finite State Machine***) του ***TAP Controller***.

Μέρος – 2ο: Υλοποίηση Κυκλώματος

Άσκηση – 3.1:

Το πρώτο κύκλωμα που σχεδιάστηκε ήταν ο καταχωρητής παράκαμψης (***Bypass Register - BR***). Είναι ένας καταχωρητής ενός ***bit*** που αποτελείται από ένα flip-flop. Επίσης αποτελείται και από μία πύλη ***and***. Επιτρέπει να περάσει το σήμα ***TDI*** απευθείας από την γραμμή ***TDO***, εφόσον είναι ενεργοποιημένο το σήμα ελέγχου ***CaptureDR***, παρακάμπτοντας έτσι τον καταχωρητή ***BSR***. Το σχέδιο φαίνεται παρακάτω:



Εικόνα 1: Κύκλωμα BR

Ας δούμε αρχικά τα σήματα ελέγχου που έχει το παραπάνω κύκλωμα. Το σήμα ελέγχου ***CaptureDR*** επιτρέπει την είσοδο των δεδομένων σειριακά μέσα στον

συγκεκριμένο καταχωρητή. Το σήμα ελέγχου ***ClockDR*** λειτουργεί ως ρολόι στο flip-flop του κυκλώματος. Η έξοδος θα βγάζει τα δεδομένα σε κάθε θετική ακμή του ρολογιού. Η υλοποίηση βασίστηκε πάνω στον κώδικα **K1** του κεφαλαίου ***APPENDIX***.

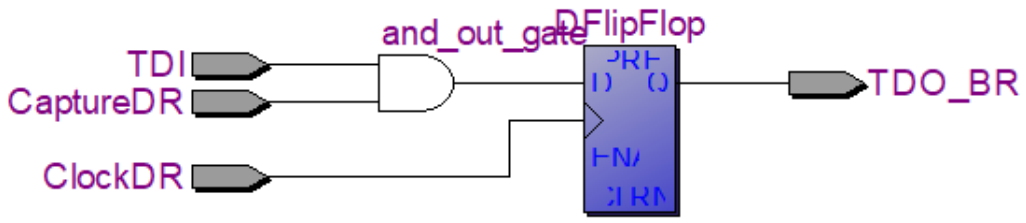
Αρχικά, ορίζουμε το μοντέλο μας ως ***module BR(TDI, CaptureDR, ClockDR, TDO_BR);*** με τις τρεις εισόδους να είναι τα σήματα ***TDI, CaptureDR, ClockDR*** και η έξοδος το σήμα ***TDO_BR***. Στη συνέχεια δημιουργείται η λογική πύλη με την ανάθεση

$$\boldsymbol{assign\ and\_out\_gate\ =\ TDI\ \&\ CaptureDR};$$

Στη συνέχεια, δημιουργούμε έναν always βρόχο για το flip-flop και σε κάθε θετική ακμή του σήματος ***ClockDR***, εισάγεται η τιμή από την έξοδο της λογικής πύλης, αφού έχει ανατεθεί και η έξοδος ως

$$\boldsymbol{assign\ TDO\_BR\ =\ DFlipFlop};$$

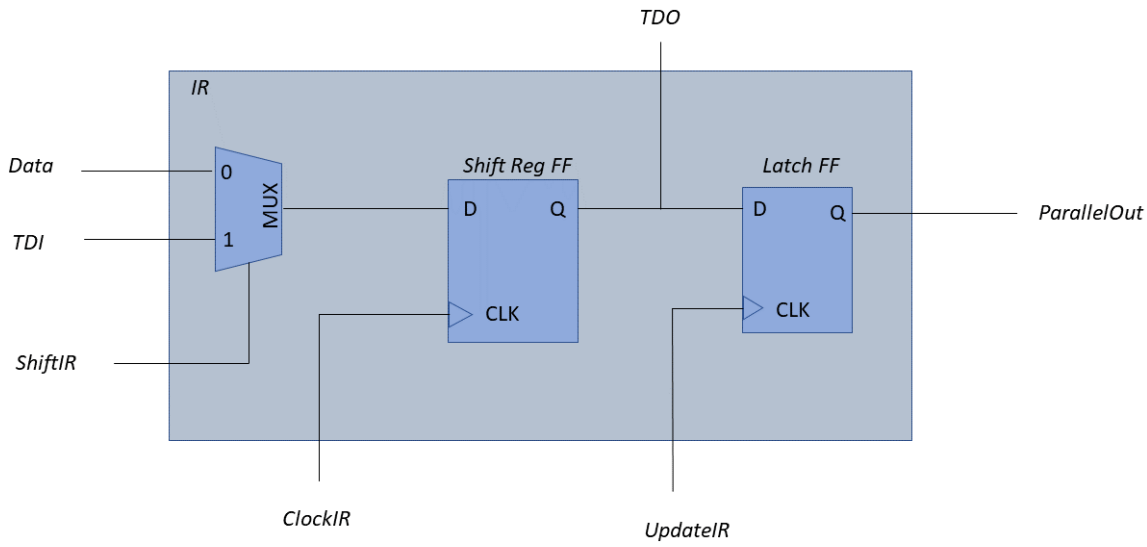
Έτσι η επαλήθευση από το quartus θα είναι η εξής:



Εικόνα 2: Κύκλωμα BR Quartus

Το επόμενο κύκλωμα που σχεδιάστηκε είναι ο καταχωρητής εντολών (***Instruction Register - IR***). Είναι ένας καταχωρητής σειριακής/παράλληλης εισόδου και εξόδου. Κάθε flip-flop του καταχωρητή οδηγεί ένα latch. Το latch αυτό κατακρατεί την τρέχουσα εντολή όταν ο καταχωρητής ενημερώνεται με νέα δεδομένα (εντολές). Ο καταχωρητής αυτό θα πρέπει να έχει τουλάχιστον δύο ***bit***. Επίσης, αποτελείται και από έναν πολυπλέκτη ο οποίος δέχεται τα δεδομένα εισόδου. Τα δεδομένα μπορεί να είναι είτε από παράλληλο φόρτωμα ων δεδομένων είτε από τα σειριακά δεδομένα του προηγούμενου κελιού. Η απόφαση αυτή θα παρθεί από το σήμα ελέγχου ***ShiftIR*** για να περάσουν τα δεδομένα μέσω της εισόδου ***TDI*** σειριακά μέσα στο κύκλωμα και να

ολισθήσουν στα υπόλοιπα κελιά σε περίπτωση που έχουμε πάνω από δύο. Επιπλέον, ο καταχωρητής ολίσθησης που βρίσκεται στο κύκλωμα δέχεται ως είσοδο τα δεδομένα του πολυπλέκτη και τα οδηγεί στο latch σε κάθε θετική ακμή του σήματος ***ClockIR(CaptureIR)***. Στη συνέχεια η έξοδος του καταχωρητή αποτελεί την σειριακή έξοδο του κυκλώματος, η οποία μπορεί να συνδεθεί με το επόμενη κελί. Τέλος, η έξοδος του latch θα βγάλει την εντολή που έχει δώσει ο κεντρικός ελεγκτής. Παρακάτω φαίνεται το σχέδιο του κυκλώματος:



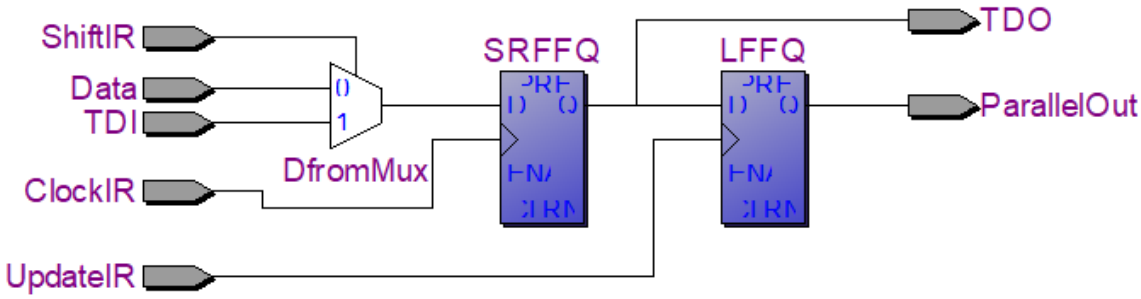
Εικόνα 3: Κύκλωμα IR

Ας δούμε αρχικά τα σήματα ελέγχου που έχει το παραπάνω κύκλωμα. Το σήμα ελέγχου ***ShiftIR*** επιτρέπει την είσοδο των δεδομένων είτε σειριακά από κάποιο προηγούμενο κελί είτε παράλληλα από κάποια μνήμη. Το σήμα ελέγχου ***ClockIR*** λειτουργεί ως ρολόι στον καταχωρητή ολίσθησης του κυκλώματος. Το σήμα ελέγχου ***UpdateIR*** λειτουργεί και αυτό ως σήμα ενεργοποίησης του latch για να βγάλουμε την εντολή από το κύκλωμα. Η υλοποίηση βασίστηκε πάνω στον κώδικα **K2** του κεφαλαίου ***APPENDIX***.

Πρώτα από όλα, έχουμε το module μας με τις εισόδους που είναι τα ***Data, TDI, ShiftIR, ClockIR, UpdateIR*** και οι έξοδοι είναι ***ParallelOut, TDO***. Στη συνέχεια φτιάχνουμε τον πολυπλέκτη με την εντολή

$$\boldsymbol{assign\ DfromMux} = (\boldsymbol{ShiftIR})?\boldsymbol{TDI}:\boldsymbol{Data};$$

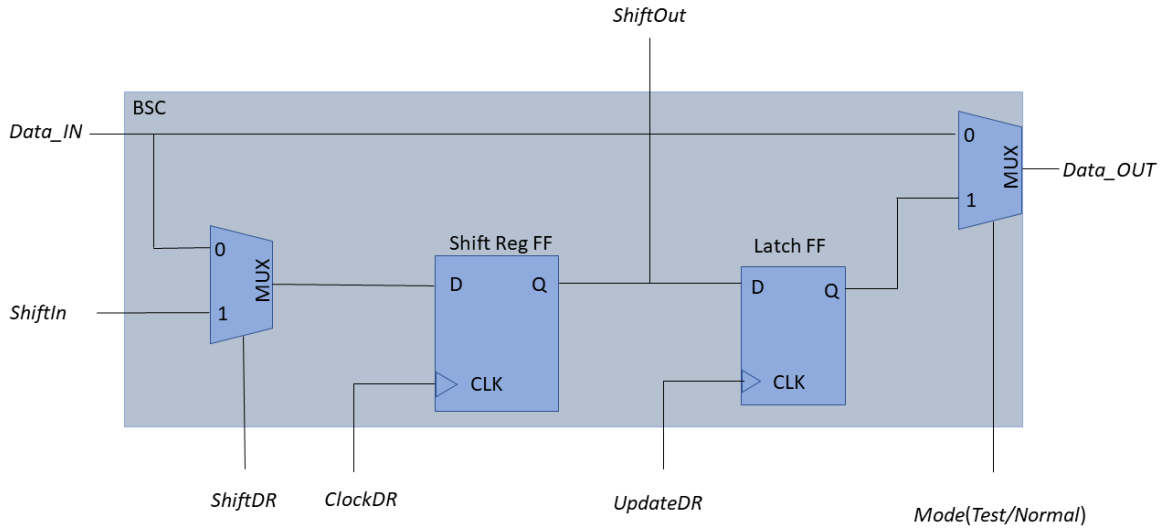
όπου βάσει του σήματος ***ShiftIR*** θα βγάζει ως έξοδο είτε τα δεδομένα σειριακής εισόδου, είτε τα δεδομένα παράλληλης εισόδου. Στη συνέχεια αναθέτουμε τις εξόδους στην έξοδο του flip-flop και του latch με την εντολή ***assign***. Τέλος, δημιουργούμε δύο ***always*** βρόχους για να φτιάξουμε τα παραπάνω. Πρώτα φτιάχνουμε τον καταχωρητή ολίσθησης όπου σε κάθε θετική ακμή του σήματος ***ClockIR (CaptureIR)*** θα δέχεται ως είσοδο τα δεδομένα από τον πολυπλέκτη. Έπειτα τα δεδομένα εξόδου από τον καταχωρητή ολίσθησης, θα είναι τα δεδομένα εισόδου στο latch. Στην έξοδο θα παίρνουμε αυτά τα δεδομένα σε κάθε θετική ακμή του σήματος ***UpdateIR***. Έτσι η επαλήθευση από το quartus θα είναι η εξής:



Εικόνα 4: Κύκλωμα IR Quartus

Το επόμενο κύκλωμα που σχεδιάστηκε είναι ο καταχωρητής περιφερειακής σάρωσης (***Boundary Scan Register – BSR*** ή ***BSC – Boundary Scan Cell***). Ο καταχωρητής αυτός αποτελεί στην ουσία μια σειριακή αλυσίδα για το ολοκληρωμένο κύκλωμα που είναι υπό δοκιμή. Η σχεδίαση του συγκεκριμένου καταχωρητή είναι παρόμοια με αυτή του καταχωρητή εντολών, με την μόνη διαφορά πως η έξοδος του latch αποτελεί ως είσοδο σε έναν πολυπλέκτη. Αρχικά, έχουμε τον πρώτο πολυπλέκτη ο οποίος δέχεται ως είσοδο τα δεδομένα παράλληλης εισόδου και επιπλέον δέχεται σειριακά δεδομένα είτε από κάποιο προηγούμενο cell είτε από την είσοδο ***TDI*** με το όνομα της να είναι ***ShiftIn***. Η επιλογή των εισόδων θα γίνεται από το σήμα ελέγχου ***ShiftDR***. Η έξοδος του πολυπλέκτη θα αποτελεί την είσοδο στον καταχωρητή ολίσθησης ***CAP***, ο οποίος ελέγχεται από το σήμα ελέγχου ***ClockDR (CaptureDR)***. Στη συνέχεια, η έξοδος του καταχωρητή θα αποτελεί την σειριακή έξοδο ***ShiftOut*** του κυκλώματος και θα ολισθαίνει τα δεδομένα στο επόμενο κελί. Επίσης, τα δεδομένα αυτά θα αποτελούν είσοδο του latch ***UPD***. Το latch ελέγχεται από το σήμα ***UpdateDR*** και σε κάθε θετική ακμή θα το ενεργοποιεί. Η έξοδος του, θα αποτελεί είσοδο στον

πολυπλέκτη. Επίσης, αυτός ο πολυπλέκτης θα δέχεται ως δεύτερη είσοδο τα δεδομένα εισόδου ***Data IN***. Τέλος, έχουμε και το σήμα ελέγχου ***Mode(Test/Normal)***, το οποίο καθορίζει την λειτουργία του κυκλώματος, Δηλαδή, εάν έχουμε παράλληλη φόρτωση των δεδομένων στο κύκλωμα επί δοκιμή (***Normal Mode***) ή έχουμε σειριακή φόρτωση των δεδομένων (***Test Mode***). Παρακάτω φαίνεται το σχέδιο του κυκλώματος:



Εικόνα 5: Κύκλωμα BSC

Η υλοποίηση βασίστηκε πάνω στον κώδικα **K3** του κεφαλαίου ***APPENDIX***. Αρχικά, ορίζονται αι είσοδοι και οι έξοδοι του κυκλώματος. Ως εισόδους, έχουμε τα σήματα ***Data_IN, ShiftIn, ShiftDR, ClockDR, UpdateDR, Mode*** και ως εξόδους τα σήματα ***ShiftOut, Data_OUT***. Στη συνέχεια, αρχικοποιούμε τα εσωτερικά υποκυκλώματα ως *wires* και *regs*.

Στην αρχή αναθέτουμε τον πρώτο πολυπλέκτη με την εντολή

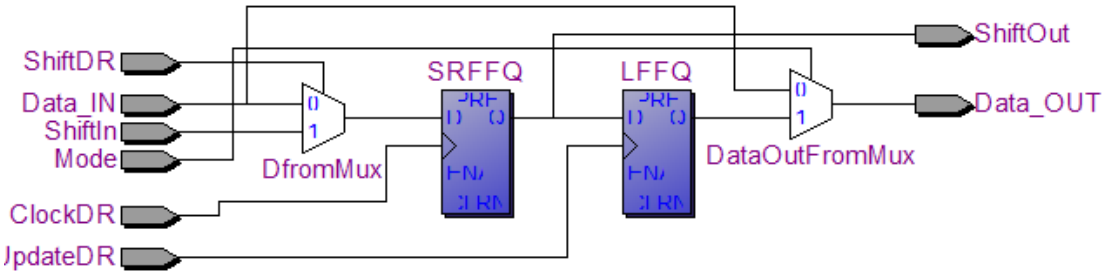
$$\boldsymbol{assign\ DfromMux\ =\ (ShiftDR)?ShiftIn:Data\_IN}$$

όπου στην έξοδο του πολυπλέκτη θα παίρνουμε είτε την είσοδο ***Data_IN*** είτε την είσοδο ***ShiftIn***, ανάλογα με την τιμή του σήματος ***ShiftDR***. Στη συνέχεια, αναθέτουμε ως έξοδο του flip-flop το σήμα ***ShiftOut*** και η έξοδος του latch θα ανατίθεται στον δεύτερο πολυπλέκτη. Το επόμενο βήμα, είναι να σχεδιάσουμε το flip-flop και το latch. Σε κάθε θετική ακμή του ρολογιού ***ClockDR***, ο reg SRFFQ παίρνει την τιμή του πολυπλέκτη DfromMux. Στη συνέχεια, σχεδιάζεται το latch του κυκλώματος. Σε κάθε θετική ακμή του σήματος ***UpdateDR***, ο reg LFFQ παίρνει την

έξοδο του flip-flop. Τέλος, σχεδιάζεται και ο δεύτερος πολυπλέκτης που αποτελεί την δεύτερη έξοδο του κυκλώματος. Η ανάθεση γίνεται με την εντολή:

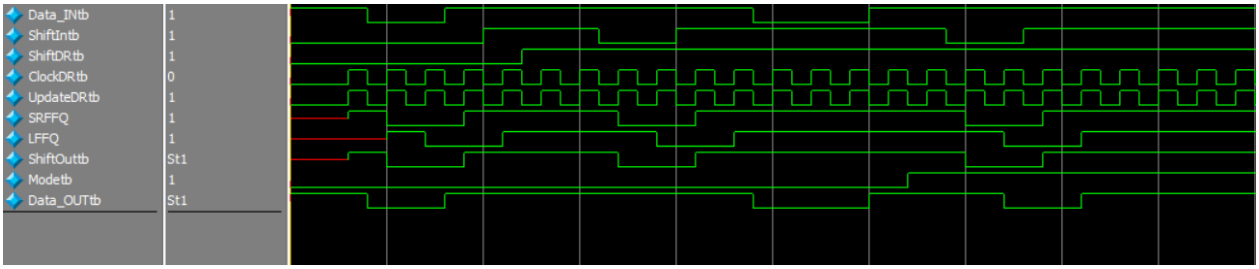
$$\boldsymbol{DataOutFromMux} = (\boldsymbol{Mode})?\,\boldsymbol{DataOutfromLatch}:\boldsymbol{Data\_IN};$$

όπου στην έξοδο του πολυπλέκτη θα παίρνουμε είτε την είσοδο ***Data_IN*** είτε την είσοδο ***DataOutfromLatch***ανάλογα με την τιμή του σήματος ***Mode***. Αν το Mode είναι μονάδα, τότε ως έξοδο θα πάρουμε το περιεχόμενο του latch. Στην αντίθετη περίπτωση θα πάρουμε τα δεδομένα εισόδου. Τέλος, αναθέτουμε ως έξοδο το ***Data_OUT***, που αποτελεί την έξοδο του πολυπλέκτη. Το αποτέλεσμα από το quartus του κυκλώματος, φαίνεται παρακάτω:



Εικόνα 6: Κύκλωμα BSC Quartus

Το επόμενο βήμα είναι να προσομοιώσουμε το κύκλωμα, για να δούμε τις διάφορες λειτουργίες που έχει. Το testbench του κυκλώματος αποτελεί τον κώδικα **K4** του κεφαλαίου **APPENDIX**. Αρχικά, αναθέτουμε τα σήματα εισόδου ως regs για να τους αναθέτουμε τιμές και ως wires τις εξόδους για να παρατηρήσουμε την λειτουργία του. Πρώτα φτιάχνουμε τα ρολόγια μέσο των clock generator blocks. Και τα δύο θα έχουν περίοδο 20 μονάδες χρόνου. Για το flip-flop έχουμε το σήμα ***ClockDRtb*** και για το latch έχουμε το σήμα ***UpdateDRtb***. Στη συνέχεια ξεκινάει ο βρόχος με τον έλεγχο διάφορων λειτουργιών. Στην κυματομορφή που ακολουθεί, φαίνονται οι διάφορες λειτουργίες του κυκλώματος. Η παραγόμενη κυματομορφή είναι η εξής:



Εικόνα 7: Κυματομορφή Κυκλώματος BSC

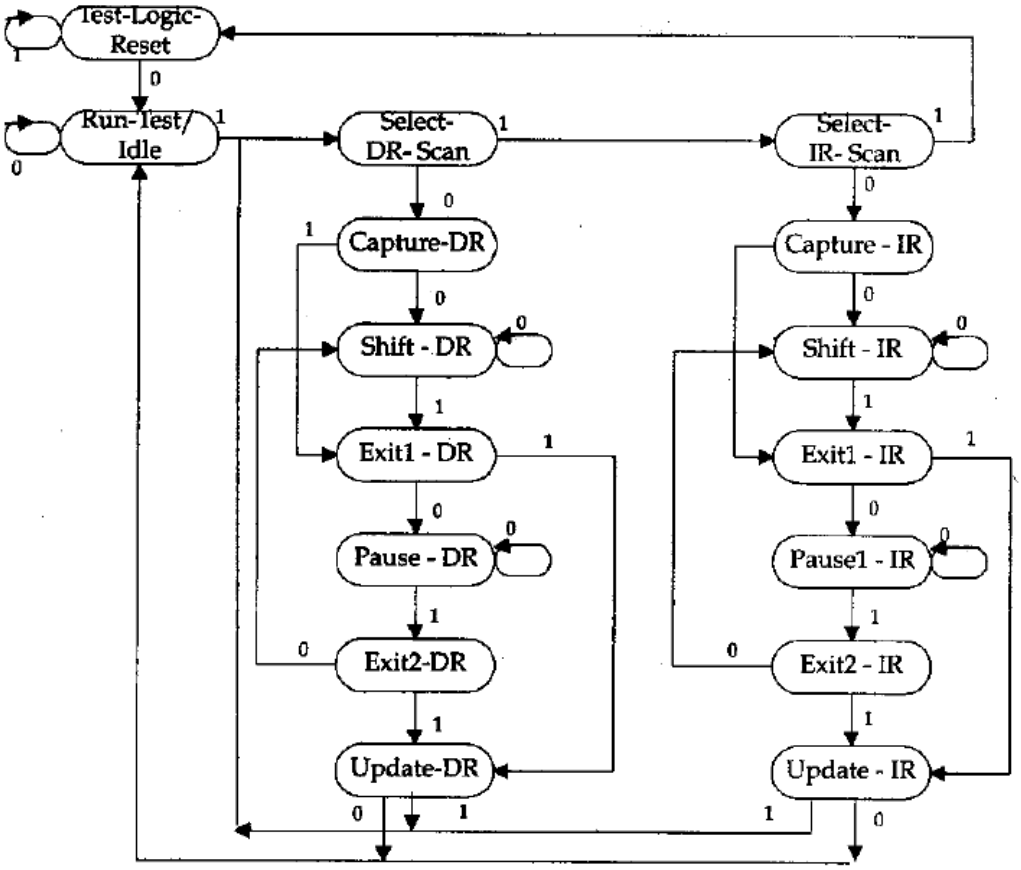
Το πρώτο βήμα που κάνουμε είναι να αρχικοποιήσουμε τα σήματα ***Modetb*** και ***ShiftDRtb*** στο 0 για να ξεκινήσει ομαλά η λειτουργία του κυκλώματος, χωρίς καθυστερήσεις. Αρχικά, ελέγχουμε τις τιμές που θα δεχθεί το flip-flop από τον πολυπλέκτη. Στο πρώτο ερώτημα θέτουμε το σήμα ***ShiftIntb = 0*** και θέτουμε ως διάνυσμα εισόδου το <101> από το ***Data_IN***. Στη συνέχεια, παρατηρούμε πως και οι τρεις τιμές εμφανίζονται στο flip-flop. Στο δεύτερο ερώτημα, αλλάζουμε το σήμα ***ShiftDRtb*** σε 1, για να δεχθεί από την σειριακή είσοδο δεδομένα και να διαδοθούν στο κύκλωμα. Έτσι, εισάγουμε σειριακά το διάνυσμα <101> με καθυστέρηση ενός κύκλου το καθένα, και παρατηρούμε μετά από κάθε κύκλο, πως οι τιμές εμφανίζονται στο flip-flop. Στο τρίτο ερώτημα απλά παρατηρούμε τις τιμές που περνάνε από το flip-flop στο latch. Αυτό συμβαίνει σε κάθε θετική ακμή του σήματος ***UpdateDRtb***. Δηλαδή, το SRFFQ μεταδίδει την τιμή του στο LFFQ με καθυστέρηση ενός κύκλου. Στο τέταρτο και πέμπτο ερώτημα ελέγχουμε την έξοδο του κυκλώματος. Δηλαδή, ανάλογα με την τιμή του ***Modetb***, το κύκλωμα είτε βρίσκεται σε κατάσταση testing, είτε βρίσκεται σε normal κατάσταση. Αρχικά, έχουμε το σήμα ***Modetb*** στο 0. Παρατηρούμε πως τα δεδομένα <10> που εισάγουμε από την ***Data_IN*** εμφανίζονται κανονικά στην ***Data_OUT***, όπως περιμέναμε. Τέλος, αλλάζουμε το σήμα ***Modetb*** σε 1 και ξεκινάει να τροφοδοτείται η έξοδος του κυκλώματος, από το latch. Εισάγουμε το διάνυσμα <10> σειριακά από την ***ShifIntb***, και παρατηρούμε πως μετά από δύο κύκλους ρολογιού μεταδόθηκε η πρώτη τιμή και μετά από μία η επόμενη. Έτσι, μπορούμε να διαπιστώσουμε την ορθή λειτουργία του παραπάνω κυκλώματος.

Άσκηση – 3.2:

Το επόμενο κυκλωματικό στοιχείο της *JTAG* αρχιτεκτονικής είναι ο ελεγκτής ***TAP Controller***. Η λειτουργία του βασίζεται σε ένα πεπερασμένο αυτόματο (*FSM- Finite State Machine*) το οποίο φαίνεται παρακάτω:

Εικόνα 8: FSM Tap Controller

Έχει ως εισόδους τα σήματα ***TCK, TMS*** και ***TRST***, όπου το πρώτο αποτελεί το ρολόι του ελεγκτή και της αρχιτεκτονικής γενικά. Το δεύτερο σήμα δέχεται τιμές και αλλάζει σε κάθε θετικό κύκλο του ρολογιού την κατάσταση της μηχανής. Τέλος, το τελευταίο σήμα ενεργοποιεί το κύκλωμα στην αρχική του κατάσταση. Τώρα, η υλοποίηση του παραπάνω βασίστηκε το διάγραμμα ροής το οποίο φαίνεται παρακάτω:



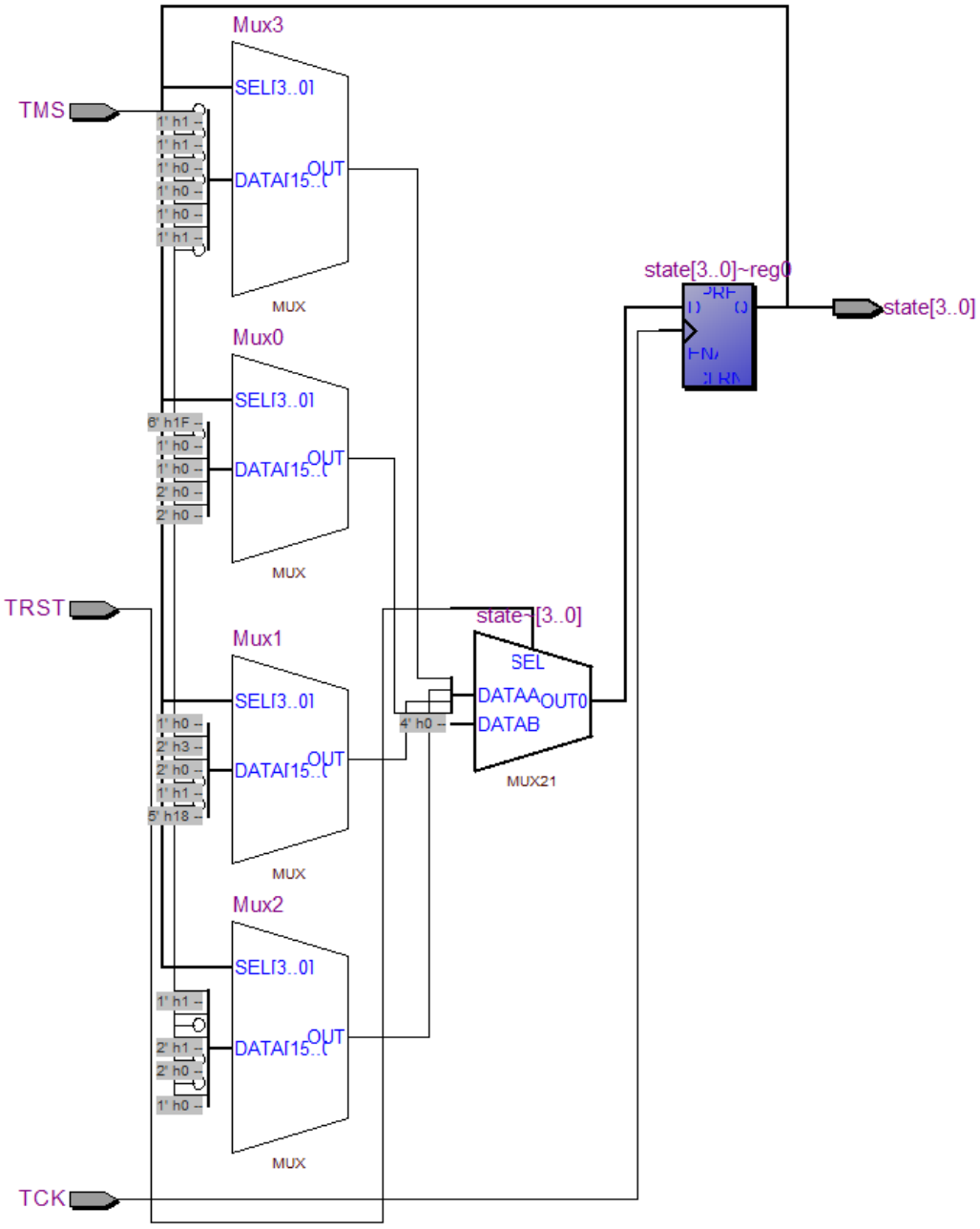
Εικόνα 9: Διάγραμμα Καταστάσεων FSM Tap Controller

το οποίο έχει υλοποιηθεί από τον κώδικα **K5** του κεφαλαίου **APPENDIX.** Αρχικά, το αυτόματο περιέχει 16 καταστάσεις, οπότε ορίζουμε ως παράμετρο το ***FSM_SIZE*** με 4, καθώς έχουμε 16 τιμές. Κάθε κατάσταση αποτελεί και μία τιμή. Οι καταστάσεις ορίζονται ως παράμετροι από το 0 έως το 15. Από το 2 έως το 8 αφορούν τις καταστάσεις για τους ***Data Registers***. Από το 9 μέχρι το 15 αφορούν καταστάσεις για τους ***Instruction Registers***. Το πρώτο βήμα που κάνουμε είναι να θέσουμε ένα καταχωρητή ***next_state*** στο 0, για να ξεκινήσει η λειτουργία από το ***TEST_LOGIC_RESET***. Στη συνέχεια φτιάχνουμε ένα case loop μέσα στο always για

να ορίσουμε κάθε κατάσταση σε χαρακτήρες ASCII (Strings). Ορίζουμε τον καταχωρητή ***STATE_AS_STR*** που έχει μέγεθος 256, για να μπορούν να φαίνονται όλες οι καταστάσεις που βρίσκεται στο αυτόματο. Τώρα, μέσα στο case loop για κάθε κατάσταση που βρίσκεται το αυτόματο αποτυπώνεται σε ένα string, όπου το όνομα αντιστοιχεί στο όνομα της κατάστασης.

Το επόμενο βήμα του προγράμματος είναι να σχεδιάζουμε την λειτουργία του αυτόματου. Δημιουργούμε ένα ασύγχρονο FSM, το οποίο βασίζεται στα σήματα ***state*** και ***TMS***. Ξεκινάμε μέσα στο always loop με ένα case loop για κάθε κατάσταση. Η μετάβαση μεταξύ των καταστάσεων γίνεται σύμφωνα με την ακολουθία τιμών (διάνυσμα) του σήματος ***TMS***. Η μετάβαση από μία κατάσταση στην άλλη γίνεται σύμφωνα με το παραπάνω διάγραμμα. Κάθε φορά που γίνεται μία μετάβαση, αποθηκεύουμε την τωρινή που βρίσκεται το αυτόματο στην ***next_state***. Αυτή η μεθοδολογία γίνεται σε κάθε κατάσταση που βρίσκεται ο ελεγκτής, μέχρι να τερματίσει η λειτουργία του.

Τέλος, έχουμε το σύγχρονο FSM του κυκλώματος. Καθώς το κύκλωμα του ***Tap Controller*** είναι ακολουθιακό, τότε χρησιμοποιούμε το σήμα ***TCK*** το οποίο είναι το ρολόι. Σε κάθε θετική ακμή του ρολογιού, ελέγχουμε την τιμή του σήματος ***TRST***, που είναι το reset και επαναφέρει το κύκλωμα στην αρχική κατάσταση που είναι ***η TEST_LOGIC_RESET*** με καθυστέρηση μιας μονάδας χρόνου. Αυτό συμβαίνει στην περίπτωση που είναι μονάδα το σήμα. Σε διαφορετική περίπτωση, ανάλογα με την τιμή του σήματος ***TMS*** αλλάζει κατάσταση και επιστρέφει την τωρινή την οποία βρίσκεται με καθυστέρηση μιας μονάδας χρόνου. Η παραπάνω ανάλυση έχει την εξής επαλήθευση από το quartus:



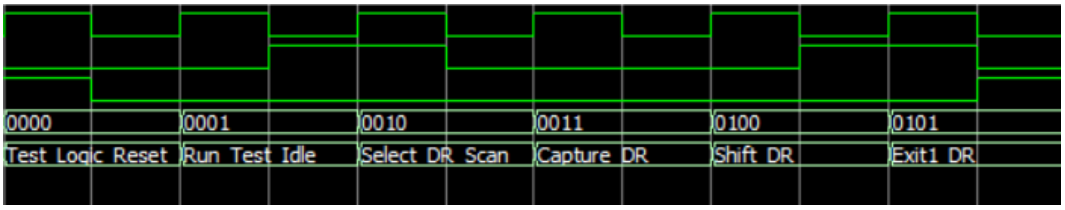
Εικόνα 10: Κύκλωμα FSM Tap Controller Quartus

Το επόμενο βήμα είναι να προσομοιώσουμε το πεπερασμένο αυτόματο για να δούμε εάν όντως λειτουργεί σωστά. Το testbench που δημιουργήθηκε είναι ο κώδικας Κ6 του κεφαλαίου **APPENDIX**. Αρχικά, ορίζουμε και πάλι το μέγεθος του αυτομάτου ως παράμετρο FSM_SIZE ίσο με 4. Στη συνέχεια φτιάχνουμε τους καταχωρητές ***TCKtb***, ***TMStb*** και ***TRSTtb***, που αποτελούν τα σήματα εισόδου του κυκλώματος. Στη συνέχεια φτιάχνουμε το wire ***statetb***, το οποίο θα είναι ένας δίαυλος μεγέθους ***FSM_SIZE***. Το επόμενο βήμα είναι να καλέσουμε το κύκλωμα ως instance με την εξής εντολή:

$$\boldsymbol{TapControllerFSM\ FSMInstance(TCKtb, TMStb, TRSTtb, statetb);}$$

Στη συνέχεια, φτιάχνουμε το ρολόι του κυκλώματος με περίοδο τα 100 *ps*, με την αλλαγή της ακμής να γίνεται στα 50 *ps*. Το τελευταίο βήμα είναι να προσομοιώσουμε την λειτουργία του κυκλώματος, βάσει των άλλων δύο σημάτων. Σε κάθε θετική ακμή του ρολογιού το κύκλωμα θα παραμένει είτε στην αρχική κατάσταση ή θα πηγαίνει στην επόμενη ανάλογα με τις τιμές των σημάτων. Αρχικά, θέτουμε το κύκλωμα στην αρχική κατάσταση με το σήμα ***TRSTtb = 1***. Μέτα από έναν κύκλο ρολογιού, μηδενίζουμε το σήμα ***TRSTtb*** και εισάγουμε σειριακά μία ακολουθία από bit ανά κύκλο, από το σήμα ***TMStb***.

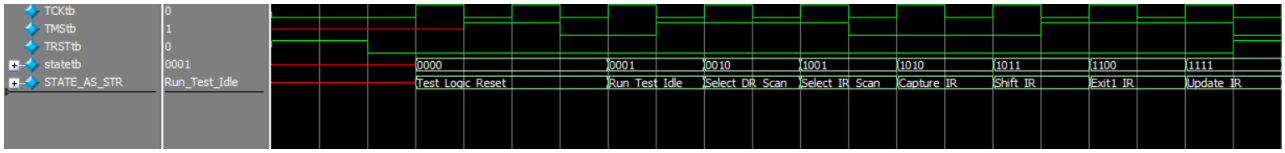
Έστω ότι έχουμε την ακολουθία ***TMS = <010010>***. Ο ελεγκτής διαβάζει ένα ένα τα bit από τα δεξιά προς τα αριστερά και αλλάζει κατάσταση. Με το πρώτο bit που είναι 0, αλλάζει κατάσταση από την αρχική που είναι η ***Test_Logic_Reset*** στην κατάσταση ***Run_Test_Idle***. Το επόμενο bit που βλέπει είναι το 1. Οπότε, η κατάσταση που θα βρεθεί θα είναι η ***Select_DR_Scan.*** Το επόμενο bit που βλέπει είναι το 0. Άρα, θα βρεθεί στην κατάσταση ***Capture_DR***. Το επόμενο bit που βλέπει είναι το 0. Άρα, θα βρεθεί στην κατάσταση ***Shift_DR***. Το επόμενο bit που βλέπει είναι το 1. Συνεπώς, θα βρεθεί στην κατάσταση ***Exit1_DR***. Καθώς, δεν βλέπει άλλη είσοδο από το ***TMS***, τότε το αυτόματο επιστρέφει στην αρχική κατάσταση την ***Test_Logic_Reset***, διότι από default case του αυτομάτου είναι η συγκεκριμένη. Η κυματομορφή είναι η εξής:



Εικόνα 11: Κυματομορφή 1 Tap Controller FSM

Ας δούμε τώρα ένα ακόμα παράδειγμα. Έστω ότι έχουμε την ακολουθία ***TMS = <11001101>***. Ο ελεγκτής διαβάζει ένα ένα τα bit από τα δεξιά προς τα αριστερά και αλλάζει κατάσταση. Με το πρώτο bit που είναι 0, αλλάζει κατάσταση από την αρχική που είναι η ***Test_Logic_Reset*** στην κατάσταση ***Run_Test_Idle***. Το επόμενο bit που βλέπει είναι το 1. Οπότε, η κατάσταση που θα βρεθεί θα είναι η ***Select_DR_Scan.*** Το

επόμενο bit που βλέπει είναι το 1. Άρα, θα βρεθεί στην κατάσταση ***Select_IR_Scan***. Το επόμενο bit που βλέπει είναι το 0. Άρα, θα βρεθεί στην κατάσταση ***Capture _IR***. Το επόμενο bit που βλέπει είναι το 0. Άρα, θα βρεθεί στην κατάσταση ***Shift_IR***. Το επόμενο bit που βλέπει είναι το 1. Άρα, θα βρεθεί στην κατάσταση ***Exit1_IR***. Το επόμενο bit που βλέπει είναι το 1. Άρα, θα βρεθεί στην κατάσταση ***Update_IR***. Καθώς, δεν βλέπει άλλη είσοδο από το ***TMS***, τότε το αυτόματο επιστρέφει στην αρχική κατάσταση την ***Test_Logic_Reset***, διότι από default case του αυτομάτου είναι η συγκεκριμένη. Η κυματομορφή είναι η εξής:



Εικόνα 12: Κυματομορφή 2 Tap Controller FSM

APPENDIX:

Παρακάτω φαίνονται οι RTL περιγραφές των κυκλωμάτων με τα αντίστοιχα testbench:

Άσκηση-3.1:

K1: Κώδικας κυκλώματος καταχωρητή BR

```
`timescale 1ns/1ps

module BR(TDI, CaptureDR, ClockDR, TDO_BR);

    input TDI, CaptureDR, ClockDR;

    output TDO_BR;

    wire TDI, CaptureDR, ClockDR, TDO_BR;

    reg DFlipFlop;

    wire and_out_gate;

    // Here we create the and gate

    assign and_out_gate = TDI & CaptureDR;

    //Here we create the flip-flop

    assign TDO_BR = DFlipFlop;
```

```
        always @  (posedge ClockDR)
        begin
                DFlipFlop<=and_out_gate;
        end
    endmodule
```

K2: Κώδικας κυκλώματος καταχωρητή IR

```
`timescale 1ns/1ps
module IR(Data, TDI, ShiftIR, ClockIR, UpdateIR, ParallelOut, TDO);
  input Data, TDI, ShiftIR, ClockIR, UpdateIR;
  output ParallelOut, TDO;
  wire Data, TDI, ShiftIR, ClockIR, UpdateIR, ParallelOut,TDO;
  reg SRFFQ, LFFQ;
  wire DfromMux;
  //here we create the Mux
  assign DfromMux=(ShiftIR)?TDI:Data;
  assign TDO=SRFFQ;
  assign ParallelOut=LFFQ;
  //here we create the SRFF
  always @ (posedge ClockIR)
  begin
    SRFFQ<=DfromMux;
  end
  //here we create the LFF
  always @ (posedge UpdateIR)
```

```
 begin
   LFFQ<=SRFFQ;
 end
endmodule
```

K3: Κώδικας κυκλώματος καταχωρητή BSC:

```
`timescale 1ns/1ps
module BSC(Data_IN, ShiftIn, ShiftDR, ClockDR, UpdateDR, Mode, ShiftOut, Data_OUT);
        input Data_IN, ShiftIn, ShiftDR, ClockDR, UpdateDR, Mode;
        output ShiftOut, Data_OUT;
        wire Data_IN, ShiftIn, ShiftDR, ClockDR, UpdateDR, Mode, ShiftOut, Data_OUT;
        reg SRFFQ, LFFQ;
  wire DfromMux, DataOutFromMux;
        //Here we create the first multiplexer
        assign DfromMux = (ShiftDR)?ShiftIn:Data_IN;
        assign ShiftOut = SRFFQ;
        assign DataOutfromLatch = LFFQ;
        //here we create the SRFF
        always @ (posedge ClockDR)
        begin
                SRFFQ<=DfromMux;
        end
        //here we create the LFF
        always @ (posedge UpdateDR)
```

```
        begin
            LFFQ<=SRFFQ;
        end
        //Here we create the second multiplexer
        assign DataOutFromMux = (Mode)?DataOutfromLatch:Data_IN;
        assign Data_OUT = DataOutFromMux;
endmodule
```

K4: Κώδικας testbench κυκλώματος καταχωρητή BSC:

```
`timescale 1ns/1ps
module BSCtb();
        reg Data_INtb, ShiftIntb, ShiftDRtb, ClockDRtb, UpdateDRtb, Modetb;
        wire ShiftOuttb, Data_OUTtb;
        // BSC circuit instance for the testbench
        BSC bscInstance(Data_INtb, ShiftIntb, ShiftDRtb, ClockDRtb, UpdateDRtb,
Modetb, ShiftOuttb, Data_OUTtb);
        //testbench starts here
        //Clock Generation for D flip-flop component
        initial begin
            ClockDRtb = 0;
            #20
            forever begin
                #10 ClockDRtb =! ClockDRtb;
            end
        end
        //Clock Generation for Latch component
```

```
        initial begin
            UpdateDRtb = 0;
            #20
            forever begin
                #10 UpdateDRtb =! UpdateDRtb;
            end
        end
        initial begin
            //initiallazation of multiplexers
            Modetb = 0;
            ShiftDRtb = 0;
            //Bellow begins the checking of every functionality of the circuit and its components
            //A
            ShiftIntb = 0;
            Data_INtb = 1;
            #40 Data_INtb = 0;
            #40 Data_INtb = 1;
            //B
            #20 ShiftIntb = 1;
            #20 ShiftDRtb = 1;
            #40 ShiftIntb = 0;
            #40 ShiftIntb = 1;
            //C
```

```
            // Every positive edge of the clock a bit passes from a bit passes from
SRFF to LFF
            //D
            // Normal mode, from internal logic input to internal logic output
            #20 Modetb = 0;
            #20 Data_INtb = 0;
            #60 Data_INtb = 1;
            //E
            // Test mode, data from LFF to internal logic outpu
            #20 Modetb = 1;
            #20 ShiftIntb = 0;
            #40 ShiftIntb = 1;
        end
        //end of testbench
endmodule
```

Άσκηση-3.2:

K5: Κώδικας μηχανής πεπερασμένων καταστάσεων TAP controller:

```
`timescale 1ns/1ps
module TapControllerFSM(TCK, TMS, TRST, state);
    //--------FSM size--------//
    parameter FSM_SIZE = 4;
    //--------Inputs--------//
    input TCK, TMS, TRST;
    //--------Output--------//
```

```
output reg [FSM_SIZE-1:0] state;
//--------Wires and Regs--------//
wire TCK, TMS, TRST;
reg [FSM_SIZE-1:0] next_state = 4'b0000;
//--------FSM States--------//
parameter TEST_LOGIC_RESET = 0;
parameter RUN_TEST_IDLE = 1;
parameter SELECT_DR_SCAN = 2;
parameter CAPTURE_DR = 3;
parameter SHIFT_DR = 4;
parameter EXIT1_DR = 5;
parameter PAUSE_DR = 6;
parameter EXIT2_DR = 7;
parameter UPDATE_DR = 8;
parameter SELECT_IR_SCAN = 9;
parameter CAPTURE_IR = 10;
parameter SHIFT_IR = 11;
parameter EXIT1_IR = 12;
parameter PAUSE_IR = 13;
parameter EXIT2_IR = 14;
parameter UPDATE_IR = 15;
//--------FSM Code Starts Here--------//
//--------For debugging purposes--------//
reg [255:0] STATE_AS_STR;
```

```
always @ (state)
begin
    case(state)
        TEST_LOGIC_RESET:
            begin
                STATE_AS_STR<="Test_Logic_Reset";
            end
        RUN_TEST_IDLE:
            begin
                STATE_AS_STR<="Run_Test_Idle";
            end
        SELECT_DR_SCAN:
            begin
                STATE_AS_STR<="Select_DR_Scan";
            end
        CAPTURE_DR:
            begin
                STATE_AS_STR<="Capture_DR";
            end
        SHIFT_DR:
            begin
                STATE_AS_STR<="Shift_DR";
            end
        EXIT1_DR:
```

```
            begin
                STATE_AS_STR<="Exit1_DR";
            end
        PAUSE_DR:
            begin
                STATE_AS_STR<="Pause_DR";
            end
        EXIT2_DR:
            begin
                STATE_AS_STR<="Exit2_DR";
            end
        UPDATE_DR:
            begin
                STATE_AS_STR<="Update_DR";
            end
        SELECT_IR_SCAN:
            begin
                STATE_AS_STR<="Select_IR_Scan";
            end
        CAPTURE_IR:
            begin
                STATE_AS_STR<="Capture_IR";
            end
        SHIFT_IR:
```

```
                begin
                    STATE_AS_STR<="Shift_IR";
                end
            EXIT1_IR:
                begin
                    STATE_AS_STR<="Exit1_IR";
                end
            PAUSE_IR:
                begin
                    STATE_AS_STR<="Pause_IR";
                end
            EXIT2_IR:
                begin
                    STATE_AS_STR<="Exit2_IR";
                end
            UPDATE_IR:
                begin
                    STATE_AS_STR<="Update_IR";
                end
        endcase
    end
    //
    //--------Asychronous FSM--------//
    //--------For every value of TMS signal change state--------//
```

```
always @ (state, TMS)
begin: MAIN_FSM_ASYCHRONOUS
    case(state)
        TEST_LOGIC_RESET:
            if(TMS == 1'b1) begin
                next_state = TEST_LOGIC_RESET;
            end else begin
                next_state = RUN_TEST_IDLE;
            end
        RUN_TEST_IDLE:
            if(TMS == 1'b1) begin
                next_state = SELECT_DR_SCAN;
            end else begin
                next_state = RUN_TEST_IDLE;
            end
        SELECT_DR_SCAN:
            if(TMS == 1'b1) begin
                next_state = SELECT_IR_SCAN;
            end else begin
                next_state = CAPTURE_DR;
            end
        CAPTURE_DR:
            if(TMS == 1'b1) begin
                next_state = EXIT1_DR;
```

```
            end else begin
                next_state = SHIFT_DR;
            end
        SHIFT_DR:
            if(TMS == 1'b1) begin
                next_state = EXIT1_DR;
            end else begin
                next_state = SHIFT_DR;
            end
        EXIT1_DR:
            if(TMS == 1'b1) begin
                next_state = UPDATE_DR;
            end else begin
                next_state = PAUSE_DR;
            end
        PAUSE_DR:
            if(TMS == 1'b1) begin
                next_state = EXIT2_DR;
            end else begin
                next_state = PAUSE_DR;
            end
        EXIT2_DR:
            if(TMS == 1'b1) begin
                next_state = UPDATE_DR;
```

```
            end else begin
                next_state = SHIFT_DR;
            end
        UPDATE_DR:
            if(TMS == 1'b1) begin
                next_state = SELECT_DR_SCAN;
            end else begin
                next_state = RUN_TEST_IDLE;
            end
        SELECT_IR_SCAN:
            if(TMS == 1'b1) begin
                next_state = TEST_LOGIC_RESET;
            end else begin
                next_state = CAPTURE_IR;
            end
        CAPTURE_IR:
            if(TMS == 1'b1) begin
                next_state = EXIT1_IR;
            end else begin
                next_state = SHIFT_IR;
            end
        SHIFT_IR:
            if(TMS == 1'b1) begin
                next_state = EXIT1_IR;
```

```
            end else begin
                next_state = SHIFT_IR;
            end
        EXIT1_IR:
            if(TMS == 1'b1) begin
                next_state = UPDATE_IR;
            end else begin
                next_state = PAUSE_IR;
            end
        PAUSE_IR:
            if(TMS == 1'b1) begin
                next_state = EXIT2_IR;
            end else begin
                next_state = PAUSE_IR;
            end
        EXIT2_IR:
            if(TMS == 1'b1) begin
                next_state = UPDATE_IR;
            end else begin
                next_state = SHIFT_IR;
            end
        UPDATE_IR:
            if(TMS == 1'b1) begin
                next_state = SELECT_DR_SCAN;
```

```
                                end else begin
                                        next_state = RUN_TEST_IDLE;
                                end
                        default: next_state = TEST_LOGIC_RESET;
                endcase
        end
        //
  //--------Sychronous FSM--------//
        always @ (posedge TCK)
        begin
                if(TRST == 1'b1) begin
                        state <= #1 TEST_LOGIC_RESET;
                end else begin
                        state <= #1 next_state;
                end
        end
endmodule
```

K6: Κώδικας testbench μηχανής πεπερασμένων καταστάσεων TAP controller:

```
`timescale 1ns/1ps
module TapControllerFSMtb();
        parameter FSM_SIZE = 4;
        reg TCKtb, TMStb, TRSTtb;
        wire [FSM_SIZE-1:0] statetb;
        TapControllerFSM FSMInstance(TCKtb, TMStb, TRSTtb, statetb);
```

```
        //Block for clock generation
        initial begin
                TCKtb = 0;
                #100
                forever begin
                        #50 TCKtb =! TCKtb;
                end
        end
        always begin
                TRSTtb = 1;
                #100 TRSTtb = 0;
                #100 TMStb = 1;
                #100 TMStb = 0;
                #100 TMStb = 0;
                #100 TMStb = 1;
                #100 TMStb = 0;
        end
endmodule
```