Μάθημα:

Ψηφιακά Συστήματα HW σε Χαμηλά Επίπεδα Λογικής Ι

Τίτλος Εργασίας:

Υλοποίηση Συστήματος Αποστολέα-Δέκτη σε Verilog με χρήση Πρωτοκόλλου UART και προβολή σε Οθόνη Τεσσάρων LED 7-Τμημάτων

ΟΜΑΔΑ 9

Ιωάννης Σιακαβάρας - 10053

Αναστασία Στρίκου - 10052

Μέρος Α’

Υλοποίηση Αποκωδικοποιητή 7 Τμημάτων

Στόχος του module είναι η αποκωδικοποίηση του 4-bit διανύσματος που στέλνει ο αποστολέας στον δέκτη σε μορφή που να μπορεί να αναπαρασταθεί σε LED 7 τμημάτων. Η υλοποίηση του είναι αρκετά απλή. Το module παίρνει ως είσοδο ένα 4-bit vector (*char*) και δίνει ως έξοδο έναν 7-bit καταχωρητή (*LED*). Τα περιεχόμενα του char συνδέονται με τις εξόδους του LED με τη χρήση ενός case statement κατά τον εξής τρόπο:

char	LED	Αναπαράσταση
4'b0000	7'b0000001	0
4'b0001	7'b1001111	1
4'b0010	7'b0010010	2
4'b0011	7'b0000110	3
4'b0100	7'b1001100	4
4'b0101	7'b0100100	5
4'b0110	7'b0100000	6
4'b0111	7'b0001111	7
4'b1000	7'b0000000	8
4'b1001	7'b0000100	9
4'b1011	7'b1111110	–
4'b1100	7'b0111000	F
default	7'b1111111	Κενό

Όπως φαίνεται στον πίνακα, ο καταχωρητής παίρνει την τιμή 1 στα ψηφία που αντιστοιχούν σε σβηστά LED και την τιμή 0 στα ψηφία που αντιστοιχούν σε αναμμένα LED. Ο ειδικός χαρακτήρας «–» συνδυάστηκε με είσοδο *4'b1011*, ο χαρακτήρας σφάλματος «F» με είσοδο *4'b1100*, ενώ για οποιαδήποτε άλλη είσοδο επιλέχθηκε ο

καταχωρητής να παίρνει την τιμή *7'b1111111*, που αντιστοιχεί με το κενό (σβηστή οθόνη).

Υλοποίηση Οδήγησης Τεσσάρων Ψηφίων

Σε αυτό το module σκοπός μας είναι να υλοποιήσουμε την οδήγηση των τεσσάρων ψηφίων εναλλάξ. Αυτό πραγματοποιείται με την οδήγηση των ανόδων οι οποίες πέφτουν στο 0 η μία μετά την άλλη. Ως χρόνος παραμονής των ανόδων στο 0 επιλέχθηκαν τα 0,32 μs. Στο module μας έχουμε ως εισόδους το *reset* (χρησιμοποιείται για την επανεκκίνηση του συστήματος), το *clock* (θεωρούμε ότι έχει περίοδο 20 ns), το 4-bit vector *character* (με το οποίο εισάγονται τα δεδομένα που θα κωδικοποιηθούν στο *LEDdecoder*). Ως εξόδους έχουμε τις 4 ανόδους μας *an3*, *an2*, *an1*, *an0* και τα σήματα *a*, *b*, *c*, *d*, *e*, *f*, *g*, που αντιστοιχούν στα 7 τμήματα της κάθε οθόνης. Στη συνέχεια δηλώνεται ο καταχωρητής *state_clk* (ο οποίος χρησιμοποιείται για την καταμέτρηση του χρόνου στον οποίο κάθε άνοδος μένει στο 0), το 4-bit vector *char* (που κρατά προσωρινά τα δεδομένα προς κωδικοποίηση), το 4-bit vector *counter* (που μετράει 16 χτύπους του clock και βοηθάει στον συγχρονισμό της πτώσης των ανόδων και της εισαγωγής των δεδομένων στον αποκωδικοποιητή, υποδηλώνοντας την κατάσταση στην οποία βρίσκεται το FSM) και το 4-bit vector *state_counter* (που υπολογίζει τον χρόνο στον οποίο οι άνοδοι είναι ενεργές).

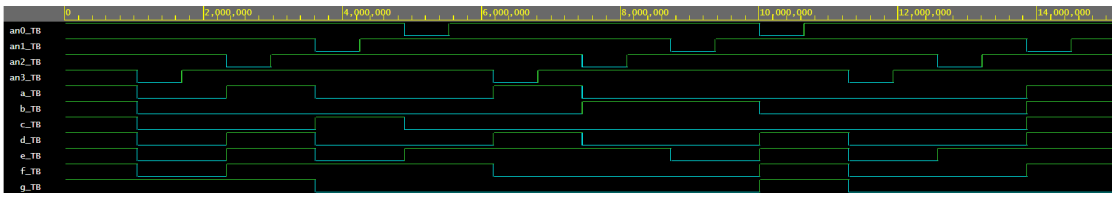
Αρχικά αρχικοποιούνται τα σήματα και οι άνοδοί μας με την τιμή 1, προκειμένου οι τέσσερις οθόνες να είναι σβηστές. Ο *state_counter* μέσω του reset αρχικοποιείται με την τιμή 0000 και σε κάθε χτύπο του *clk* αυξάνεται κατά 1. Το *state_clk* με το reset παίρνει την τιμή 0, και κάθε φορά που ο *state_counter* μας γίνεται ίσος με 1111 (15) (δηλαδή κάθε φορά που ολοκληρώνεται ένας κύκλος πτώσης των ανόδων), παίρνει την τιμή 1, ενώ καθ' όλη την υπόλοιπη διάρκεια βρίσκεται στο 0. Ο *counter* μας, λόγω του *reset*, ξεκινάει να μετράει από την τιμή 0001 και σε κάθε χτύπο του *state_clk* μειώνεται κατά 1. Οι άνοδοι πέφτουν διαδοχικά στις εξής τιμές του *counter*:

Άνοδος	Counter
an3	4'b1110
an2	4'b1010
an1	4'b0110
an0	4'b0010

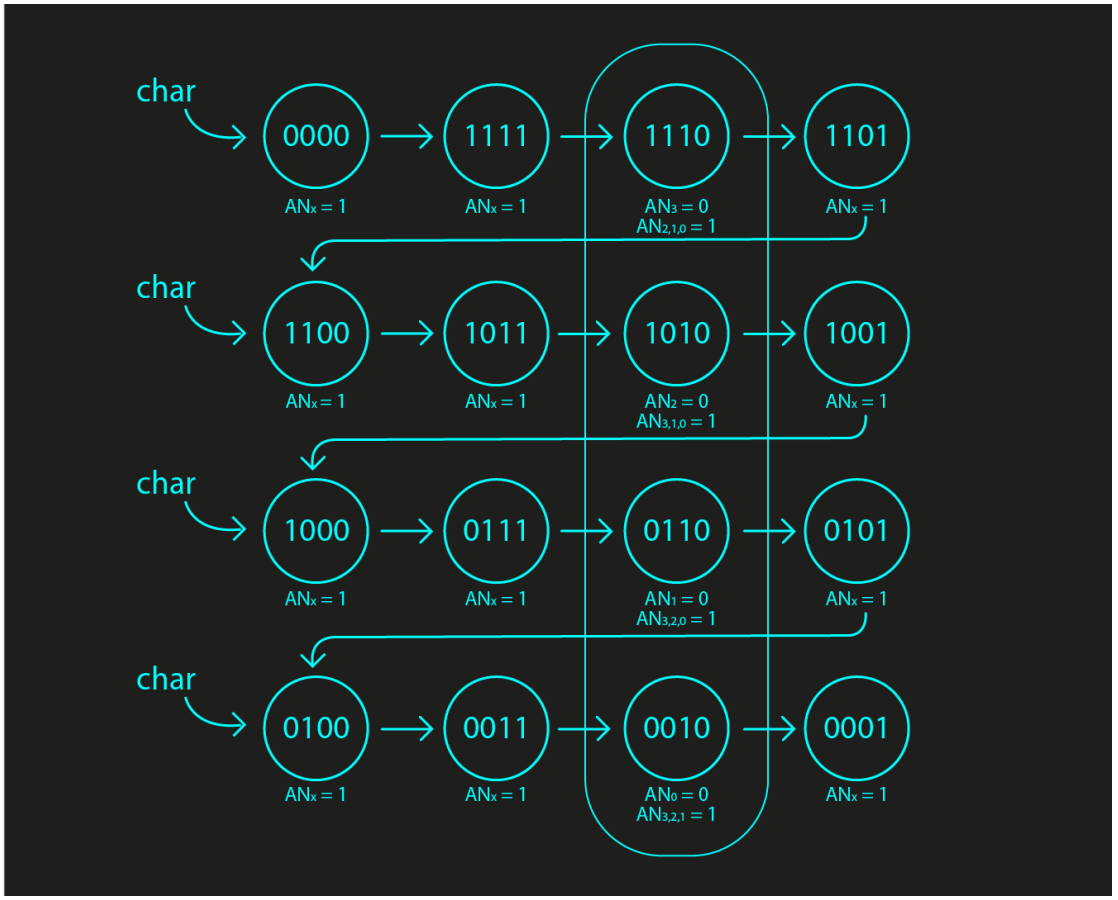
Ο καταχωρητής *char* αρχικοποιείται με την τιμή 1111, έτσι ώστε σε περίπτωση που μπούμε στον αποκωδικοποίητη πριν να πάρουμε τα δεδομένα, οι οθόνες να παραμείνουν σβηστές. Τα δεδομένα που εισάγονται στο module αποθηκεύονται προσωρινά στον *char* 2 χτύπους του *state_ clk* πριν την πτώση της κάθε ανόδου, όταν ο *counter* έχει τιμές 0000, 0100, 1000 και 1100. Με το *char* και το 7-bit wire *LED* που ορίσαμε γίνεται instantiate ο αποκωδικοποιητής μας. Τα σήματα παίρνουν τις τιμές μέσω του αποκωδικοποιητή μετά από κάθε πτώση των ανόδων, για counter ίσο με 0010, 0110, 1010 και 1110.

Testbench

Για το testbench του *FourDigitLEDdriver,* ορίζουμε τους καταχωρητές *clk_TB* και *reset_TB*, το 4-bit vector *character_TB* και τα wires *an3_TB*, *an2_TB*, *an1_TB*, *an0_TB*, *a_TB*, *b_TB*, *c_TB*, *d_TB*, *e_TB*, *f_TB*, *g_TB*, *LED_TB*. Το *character_TB* αρχικοποιείται με την τιμή 1111, το *clk_TB* αρχικά βρίσκεται στο 0 και το *reset_TB* στο 1. Το *clk_TB* ορίζεται να έχει περίοδο 20ns. Μετά από 100 timing units (ns), το *reset_TB* πέφτει και αρχίζουν να αλλάζουν οι τιμές του character_TB ανά 1280 ns. Τρέχοντας το testbench επιβεβαιώνεται η ομαλή λειτουργία του module FourDigitLEDdriver και του αποκωδικοποιητή.



FSM



EDA Playground link: https://edaplayground.com/x/uAg7

Προβλήματα που αντιμετωπίσαμε

Το πρώτο κομμάτι της εργασίας λειτούργησε για εμάς ως η πρώτη επαφή με την Verilog και τον προγραμματισμό hardware. Ως εκ

τούτου, τα προβλήματα που προέκυψαν ήταν κατά βάση λόγω συντακτικών λαθών και έλλειψης εξοικείωσης με το περιβάλλον του EDAPlayground, οπότε ξεπεράστηκαν σχετικά εύκολα και γρήγορα!

ΜΕΡΟΣ Β’

Baud Contoller:

Σε αυτό το module στόχος μας είναι να μετρήσουμε το T_{sc} = 1/(16*baud_rate). Είσοδο έχουμε το reset, το clock, το baud_select το οποίο μας υποδηλώνει ποιο baud_rate θα χρησιμοποιήσουμε και έξοδο έχουμε το sample_enable το οποίο γίνεται 1 όταν έχει μετρηθεί ο χρόνος T_{sc} επιτυχώς. Για να υπολογίσουμε σωστά τον συγκεκριμένο χρόνο, υπολογίζουμε το ελάχιστο κοινό πολλαπλάσιο των baud_rates, το οποίο είναι το Tsample = 1 /(16*115.200) = 542,5347 nsec.

Παρατηρούμε ότι ένας εύκολος τρόπος για να μετρήσουμε τον χρόνο T_{sc} είναι με έναν μετρητή που θα φτάνει ως το 27 (27*clock = 27*20 nsec = 540 nsec). Δηλαδή κάθε φορά που ο counter θα παίρνει την τιμή 27 θα έχω σφάλμα (542,534 – 540 = 2,5347 nsec). Στη συνέχεια ορίζουμε ακόμη έναν μετρητή τον count_reps, ο οποίος μετράει τις φορές που ο counter έγινε 27 και ανάλογα με το επιλεγμένο baud_rate από την είσοδο μας λέει πότε πρέπει το sample_enable να γίνει 1.

Η υλοποίηση γίνεται με μια case σε ένα always block (σε κάθε χτύπο ρολογιού γίνεται έλεγχος). Στο τέλος χρησιμοποιούμε ένα ακόμη always block για τον μηδενισμό του sample_enable και του count_reps όταν το sample_enable γίνεται 1 καθώς θέλουμε να επαναλαμβάνεται η διαδικασία.

Παρακάτω δίνεται ο πίνακας με τα σφάλματα για όλες τις τιμές του baud_rate.

Baud_Rate σε bits/sec	Count_Reps	Σφάλμα σε nsec
300	384	384*2,5347 = 973
1200	96	96*2,5347 = 243,26
4800	24	24*2,5347 = 60,81
9600	12	12*2,5347 = 30,416
19200	6	6*2,5347 = 15,2
38400	3	3*2,5347 = 7,6
57600	2	2*2,5347 = 5,069
115200	1	1*2,5347 = 2,534

Υλοποίηση UART Αποστολέα

Στο module αυτό υλοποιείται ο αποστολέας του συστήματός μας. Οι είσοδοί μας είναι το *reset* και το *clk*, καθώς επίσης το 8-bit vector *Tx_DATA* (όπου βρίσκονται τα δεδομένα του συμβόλου προς μεταφορά), το 3-bit vector *baud select* (ορίζει την ταχύτητα του baud controller) και τα σήματα *Tx_EN* (σήμα ενεργοποίησης του αποστολέα) και *Tx_WR* (επισημαίνει ότι ο αποστολέας μπορεί να παραλάβει δεδομένα). Οι έξοδοι του module είναι το σήμα *TxD*, μέσω του οποίου τα δεδομένα αποστέλλονται στον δέκτη bit προς bit, και το σήμα *Tx_BUSY* (βρίσκεται στο 1 όσο ο αποστολέας λαμβάνει και αποστέλλει δεδομένα). Ορίσαμε επιπλέον τα vector *tx_data_reg* (8-bit, εδώ αποθηκεύονται προσωρινά τα δεδομένα προς μεταβίβαση ώστε να μη χαθούν ακόμα κι αν το Tx_DATA αλλάξει), *counter_sample* (4-bit, μετράει τις φορές που ο καταχωρητής *tx_sample_enable* που θα οριστεί παρακάτω γίνεται 1) και *counter* (4-bit, μετράει τις φορές που ο counter_sample γίνεται 1111), τους καταχωρητές *tx_enable* (ενεργοποιεί το Tx_EN), *parity_bit* (κρατάει το parity bit μέχρι να αποσταλεί μέσω του TxD), *reset_baud* (στέλνει σήμα ώστε να γίνει επανεκκίνηση του baud controller και να είναι

σωστά συγχρονισμένο το σύστημα) και το wire *tx_sample_enable* (αποτελεί την έξοδο του baud controller).

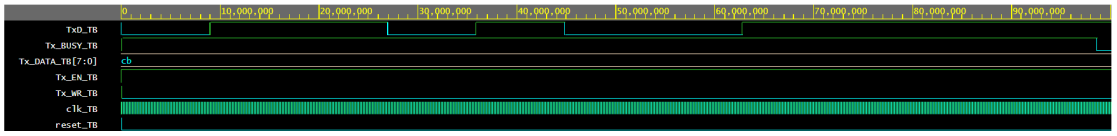
Ο *counter_sample*, ο *counter* και το *reset_baud* αρχικοποιούνται μέσω του *reset* και παίρνουν τις τιμές 0000, 0000 και 1 αντίστοιχα. Με το reset αρχικοποιούνται επίσης και ο *tx_data_reg* (0000_0000), το *tx_enable* (0), το *Tx_BUSY* (0) και το *TxD* (1, βρίσκεται στο stop bit). Όταν το *Tx_EN* είναι ίσο με ένα (δηλαδή όταν ο αποστολέας είναι ενεργός), αν ταυτόχρονα στέλνεται σήμα από τον baud controller μέσω του *tx_sample_enable*, τότε μαζί με τον χτύπο του *clk* ο *counter_sample* μεγαλώνει κατά 1. Αν φτάσει στην τιμή 1111, την επόμενη φορά που θα πληρούνται όλες οι συνθήκες θα αυξηθεί και ο *counter* κατά 1. Στη συνέχεια, ο *counter* χρησιμοποιείται σε case statement, όπου ανάλογα την τιμή του στέλνεται και το αντίστοιχο bit δεδομένων μέσω του *TxD*. Έτσι καταφέρνουμε να στέλνονται τα δεδομένα με τον ρυθμό που ορίζει ο baud controller. Το *TxD* παίρνει διαδοχικά τις τιμές που είναι αποθηκευμένες στον *tx_data_reg* και άρα τα δεδομένα αποστέλλονται με την εξής σειρα:

start bit (= 0) → least significant bit → … → most significant bit → parity bit → stop bit (= 1)

Για να λειτουργήσει ο αποστολέας δεν αρκεί να ενεργοποιηθεί μόνο μέσω του *Tx_EN*. Πρέπει ταυτόχρονα να ενεργοποιηθεί και το *Tx_WR*, ώστε να λάβει το *Tx_DATA*. Αν τα δυο σήματα γίνουν ταυτόχρονα 1, τότε ο *reset_baud* μηδενίζεται (και άρα αρχίζει η λειτουργία του baud controller), το *counter_sample* μηδενίζεται, το *tx_enable* και το *Tx_BUSY* γίνονται 1 (και άρα το σύστημα καταλαβαίνει ότι ο αποστολέας έχει τεθεί σε λειτουργία), το *parity_bit* παίρνει την τιμή 1 αν στο *Tx_DATA* υπάρχει μονός αριθμός άσων ή την τιμή 0 αν υπάρχει ζυγός, με την χρήση bitwise τελεστή XOR. Αν ο αποστολέας είναι ενεργοποιημένος, το *Tx_WR* είναι 0, το *Tx_BUSY* 1 και ο *counter* ίσος με 1011, που σημαίνει ότι έχει στείλει τα δεδομένα, το *TxD* παραμένει στο 1 (stop bit), ο baud controller επανεκκινείται μέσω του *reset_baud* (που γίνεται ίσο με 1) και το σήμα *Tx_BUSY* πέφτει στο 0, όποτε και ειδοποιείται το υπόλοιπο σύστημα ότι η μεταφορά των δεδομένων έχει τελειώσει.

Testbench Αποστολέα

Για το testbench του *uart_transmitter,* ορίζουμε τους καταχωρητές *clk_TB*, *reset_TB*, *Tx_EN_TB*, *Tx_WR_TB*, το 8-bit vector *Tx_DATA_TB*, το 3-bit vector *baud_select_TB* και τα wires *TxD_TB*, *Tx_BUSY_TB*. Όπως και στο προηγούμενο testbench, το *clk_TB* αρχικά βρίσκεται στο 0 και το *reset_TB* στο 1. Το *clk_TB* ορίζεται να έχει περίοδο 20ns. Αρχικοποιούνται επιπλέον τα *Tx_EN_TB* και *Tx_WR_TB* στο 0, το *Tx_DATA_TB* στο 1111_1111 (δηλαδή κενό), και ο *baud_select_TB* γίνεται ίσος με 111 (και άρα ο baud controller έχει τη μεγαλύτερη δυνατή ταχύτητα). Μετά από 100 timing units (ns), το *reset_TB* πέφτει και δίνουμε στο instantiation του αποστολέα ένα τυχαίο δεδομένο προς επεξεργασία (στην προκειμένη περίπτωση τον αριθμό 1100_1011), ενώ ταυτόχρονα ενεργοποιούνται το *Tx_EN_TB* και το *Tx_WR_TB*. Μετά από μια περίοδο του *clk_TB* (20ns), το *Tx_WR_TB* πέφτει ξανά στο 0, οπότε ο αποστολέας είναι ελεύθερος να ξεκινήσει την επεξεργασία των δεδομένων. Το testbench αρχικά υλοποιήθηκε για όλες τις ταχύτητες που θα μπορούσε να πάρει ο baud controller, καθώς όμως ο χρόνος που απαιτούνταν ήταν μεγάλος και το σύστημα κατέληγε σε time out, περιοριστήκαμε στην απόδειξη λειτουργίας για τη μεγαλύτερη ταχύτητα.



EDA Playground link: https://edaplayground.com/x/YN83

UART Δέκτης:

Ο Δέκτης UART είναι ενεργοποιήσιμος με τον ίδιο τρόπο όπως ο Αποστολέας και επιδεικνύει στο σύστημα ότι υπάρχει διαθέσιμο σύμβολο προς ανάγνωση. Το σήμα *Rx_EN* αποτελεί το σήμα ενεργοποίησης του Δέκτη, με ανάλογη συμπεριφορά με τον Αποστολέα. Κατά την ολοκλήρωση της λήψης από τον Δέκτη, αν διαπιστωθεί σφάλμα στην ισοτιμία ή στη σειριακή πλαισίωση των δεδομένων, θα πρέπει να γίνουν 1 τα αντίστοιχα σήματα λάθους,

Rx_PERROR και *Rx_FERROR*. Εναλλακτικά, αν δεν υπάρχει σφάλμα, το σύμβολο που θα παραληφθεί θα πρέπει να εμφανιστεί στο *Rx_DATA[7:0*], και το σήμα *Rx_VALID* θα πρέπει να τεθεί στην τιμή 1.

Οι είσοδοι του δέκτη είναι το *reset*, το *clock*, ο *baud_select* ο οποίος ρυθμίζεται ομοίως με τον αποστολέα, το Rx_EN που μας δηλώνει ότι το σύστημα είναι έτοιμο για λήψη δεδομένων και το *RxD* που είναι η σειριακή μεταφορά δεδομένων (11 bit – 1 start bit, 8 data bits, 1 parity και 1 stop bit).

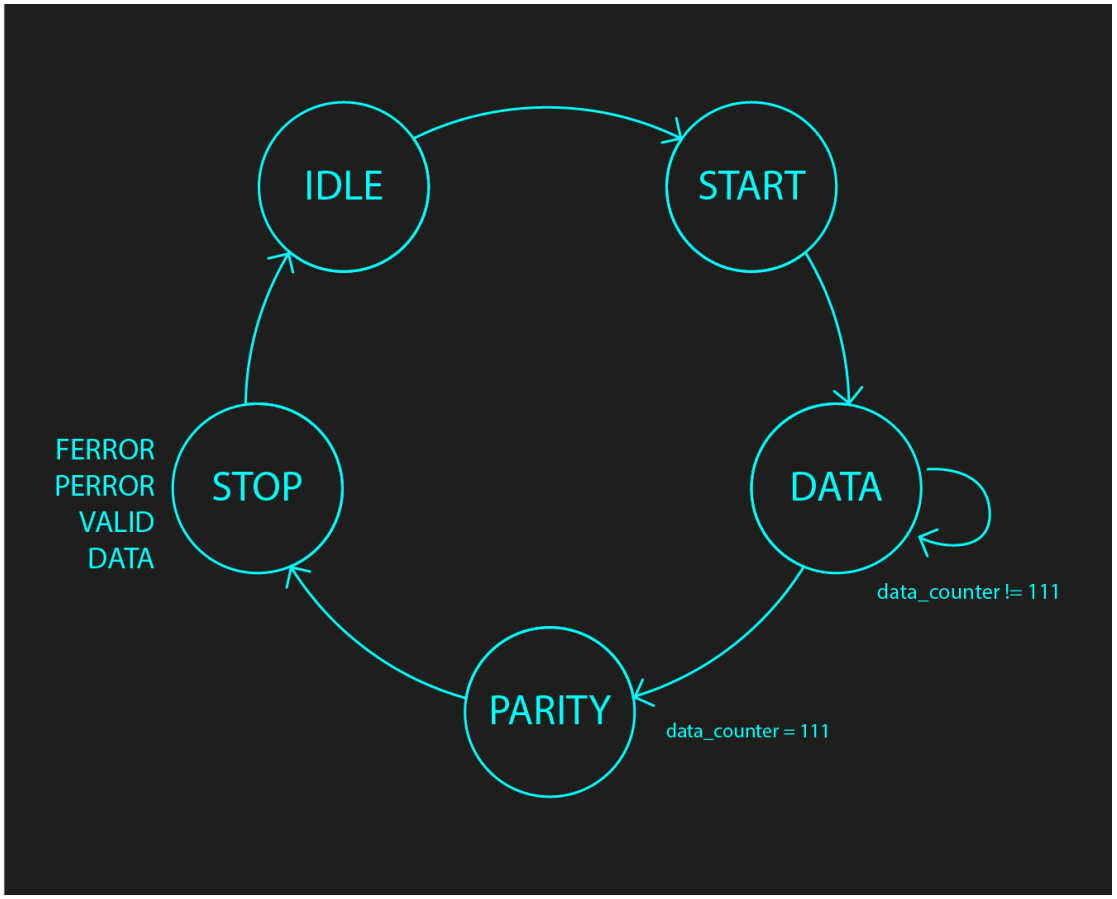
Οι έξοδοι του δέκτη είναι τα δεδομένα (8 data bit) που διαβάστηκαν, τα λάθη *Rx_FERROR* και *Rx_PERROR* και το *Rx_VALID*, που μας υποδηλώνει εάν τα δεδομένα που διαβάστηκαν είναι valid, δηλαδή δεν υπήρχε κανένα σφάλμα στην δειγματοληψία.

Αρχικά, δηλώνουμε το sample, το οποίο έχει 4 bit, καθώς για κάθε bit θα κάνουμε δειγματοληψία 4 φορές. Η δειγματοληψία θα γίνεται στη μέση χρονική στιγμή του bit και μετά. Αυτό θα γίνει χρησιμοποιώντας έναν μετρητή, τον counter, ο οποίος μετράει τις φορές που το *sample_enable* είναι 1. Ξέρουμε ότι ο δέκτης είναι 16 φορές πιο γρήγορος από τον αποστολέα οπότε γνωρίζουμε ότι η διάρκεια κάθε bit είναι 16 φορές το *sample_enabl*e. Άρα η δειγματοληψία θα γίνει στις χρονικές στιγμές 7,8,9,10 δηλαδή από την μέση και μετά.

Δηλώνουμε, επιπλέον, ένα reg με το όνομα *data_counter* που μας υποδηλώνει ποιο bit διαβάζουμε, εάν είμαστε στη διαδικασία ανάγνωσης των 8 data bit. Επίσης, δηλώνουμε reg για τις μεταβλητές εξόδου καθώς και temporary μεταβλητές εξόδου, έτσι ώστε να ανανεώνονται οι έξοδοι μετά την ανάγνωση του stop bit και ένα reset για τον *baud_controller* το *reset_baud*.

Στο αρχικό always block κάνουμε reset τον counter, αλλιώς εάν το *sample_enable* γίνει 1 και το σύστημα του δέκτη είναι ενεργό (*Rx_EN*) τότε αυξάνουμε τον counter κατά 1. Εάν ο counter έχει πάρει τη μέγιστη τιμή του (15) τότε τον μηδενίζουμε και εάν βρισκόμαστε στην κατάσταση ανάγνωση των 8 data bit αυξάνουμε το *data_counter* κατά 1, γιατί ήρθε η ώρα να διαβάσουμε το επόμενο bit, καθώς μετρήθηκαν 16 κύκλοι του *sample_enable*.

Το FSM του δέκτη είναι το παρακάτω:



Έχουμε τις καταστάσεις *idle, start, data, parity* και stop (5 καταστάσεις). Όλες οι παρακάτω ενέργειες γίνονται εφόσον το σύστημα του δέκτη είναι ενεργό. Εάν πέσει οποιαδήποτε στιγμή το *Rx_EN*, τότε τερματίζεται η διαδικασία ανάγνωσης και γυρνάμε στην κατάσταση *idle*.

Η κατάσταση *idle* είναι η αρχική κατάσταση και λειτουργεί ως κατάσταση αναμονής. Κάνουμε ένα reset τον *baud_controller*, καθώς δεν θέλουμε να μετράμε τα *sample_enables* στο *idle*, διότι θα φύγουμε από αυτήν την κατάσταση μονάχα εάν διαβάσουμε 0 στην είσοδο *RxD*. Αρχικοποιούμε, επίσης τον *counter*, τον *data_counter* και το *sample* (με 4'bxxxx το τελευταίο). Εάν πέσει η είσοδος *RxD* στο 0 τότε μεταβαίνουμε στην επόμενη κατάσταση, η οποία είναι το start και ρίχνουμε το *reset_baud*, γιατί τώρα θέλουμε να μετράμε τα *sample_enables*.

Στην κατάσταση start, αρχικά πραγματοποιούμε την δειγματοληψία στις χρονικές στιγμές που αναφέραμε νωρίτερα. Όταν ο counter

πάρει την τιμή 15, δηλαδή μετρήθηκαν 16 *sample_enable*, μηδενίζουμε τον counter και το sample γίνεται ίσο με 4'bxxxx και ελέγχουμε να δούμε εάν όλες οι τιμές τις δειγματοληψίας συμφωνούν και είναι ίσες με το 0. Εάν υπάρχει κάποια διαφοροποίηση και βρούμε τιμή του *sample* διαφορετική του 0, τότε έχουμε frame error (*FERROR*) και δίνουμε την τιμή 1 στο *Rx_FERROR_temp*.

Στη συνέχεια μεταβαίνουμε στην κατάσταση data στην οποία θα μείνουμε 8*T, όπου T = 1/baud_rate, επειδή θα δειγματοληπτήσουμε τα 8 bits δεδομένων.

Πραγματοποιούμε τη δειγματοληψία όπως νωρίτερα και όταν ο counter πάρει τη μέγιστη τιμή του τότε ελέγχουμε εάν το *sample* έχει ίδια όλα τα bit του. Αυτό το πραγματοποιούμε με την συνθήκη:

if(!*(sample* == 4'b0000 || *sample* == 4'b1111). Εάν οι τιμές του sample είναι διάφορες του 0000 ή του 1111 τότε έχουμε frame error (*FERROR*).

Στη συνέχεια αποθηκεύουμε την πρώτη τιμή της δειγματοληψίας, δηλαδή αυτή στη μέση του T στην κατάλληλη θέση του *Rx_DATA_temp*. Εφόσον ο αποστολέας στέλνει τα bit δεδομένων σειριακά από το LSB προς το MSB, τότε ο data_counter μας δίνει την θέση στο διάνυσμα *Rx_DATA_temp*. Όταν έρθει το 1ο bit δεδομένων, δηλαδή το LSB, αυτό θα αποθηκευτεί στην μηδενική θέση του vector, κοκ, μέχρι το 8ο bit δεδομένων, δηλαδή το MSB, το οποίο θα αποθηκευτεί στην 7η θέση του vector. Όταν ο *data_counter* πάρει τη μέγιστη τιμή του (7) τότε προχωράμε στην επόμενη κατάσταση.

Στην κατάσταση parity γίνονται οι ίδιες διαδικασίες όπως πριν και επιπλέον ελέγχουμε την ισοτιμία του parity bit από την δειγματοληψία του 10ου bit με το parity bit από τα δειγματοληπτημένα 8 bit δεδομένων, το οποίο το βρίσκουμε χρησιμοποιώντας XOR. Αυτό το ελέγχουμε με την εντολή if(*parity_bit* != ^*Rx_DATA_temp*). Εάν είναι διαφορετικά τα 2 bit τότε έχουμε parity error (*PERROR*). Επόμενη κατάσταση είναι η stop

Τελική μας κατάσταση είναι η stop, στην οποία γίνονται οι ίδιες ενέργειες με πριν με μόνη διαφορά το ότι ελέγχουμε το sample με το 1111, καθώς γνωρίζουμε ότι το stop bit πρέπει να είναι 1. Επόμενη κατάσταση είναι η κατάσταση αναμονής *idl*e.

Τέλος έχουμε ένα ακόμη always block στο οποίο κάνουμε reset τις μεταβλητές μας (το *Rx_VALID* είναι 1 στην υλοποίηση μας έως ότου βρεθεί κάποιο σφάλμα). Αλλιώς περνάμε την τιμή της επόμενης κατάστασης στην τρέχουσα και ελέγχουμε εάν η επόμενη κατάσταση είναι το *idle* Τότε, περνάμε τις τιμές των *FERROR, PERROR* και *Rx_DATA_temp* στην έξοδο και δίνουμε τιμή στο *Rx_VALID* ανάλογα με τις τιμές των error. Εάν οποιοδήποτε από τα *error* μας είναι 1 τότε το *valid* πρέπει να είναι 1.

Testbench:

Για το testbench του *uart_receiver*, ορίζουμε τους καταχωρητές *clk_TB reset_TB, Rx_EN_TB και RxD_TB,* το 3-bit vector *baud_select_TB* και τα wires *Rx_FERROR_TB, Rx_PERROR_TB, Rx_VALID_TB, Rx_DATA_TB, 4-bit vector sample_TB και 8bit-vector Rx_DATA_temp_TB.*

Το *baud_select_TB* αρχικοποιείται με την τιμή 111, η οποία είναι και η πιο γρήγορη, καθώς το EDA Playground έχει περιορισμό στον χρόνο της προσομοίωσης και πετάει σφάλμα σε μεγαλύτερες περιόδους (μικρότερα *baud_rates*). Το *clk_TB* αρχικά βρίσκεται στο 0, το *reset_TB* στο 1 το *Rx_EN_TB* στο 0 και το *RxD_TB* στο 1 (stop bit). Το *clk_TB* ορίζεται να έχει περίοδο 20ns.

Μετά από 100 timing units (ns), το *reset_TB* πέφτει, το *Rx_EN_TB* γίνεται 1 και αρχίζουν να αλλάζουν οι τιμές του *RxD_TB* ανά 8850 ns (όσο περίπου και η περίοδος του T). Το νούμερο είναι αυτό συγκεκριμένα για να γίνεται η δειγματοληψία στο κέντρο κάθε bit.

Τρέχοντας το testbench επιβεβαιώνεται η ομαλή λειτουργία του module *uart_receiver*. Τα δεδομένα δειγματοληπτούνται σωστά από τον δέκτη όπως φαίνεται και στο log (τυπώνουμε ορισμένες τιμές με το $monitor).

Το μόνο πρόβλημα που υπάρχει είναι στο parity bit καθώς ο δέκτης φαίνεται να σταματάει όταν έρθει η ώρα να πάει από την κατάσταση data στην parity. Δεν είχαμε χρόνο να επιλύσουμε το συγκεκριμένο πρόβλημα αλλά πιστεύουμε πως η υλοποίηση είναι σωστή, απλώς

υπάρχει κάποιο λαθάκι στον receiver και στους *counter* ή στο *next state*.

Παρακάτω βλέπουμε τις τιμές στο testbench του *RxD_TB* με τις καθυστερήσεις:

RxD_TB = 0; #8850 //start bit

RxD_TB = 1; #8850 //LSB

RxD_TB = 0; #8850

RxD_TB = 1; #8850

RxD_TB = 1; #4470 //there should be ferror

RxD_TB = 0; #4380

RxD_TB = 0; #8850

RxD_TB = 1; #8850

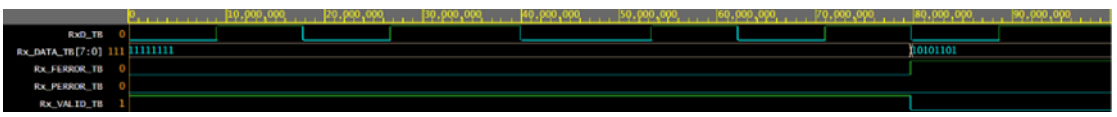
RxD_TB = 0; #8850

RxD_TB = 1; #8850 //MSB

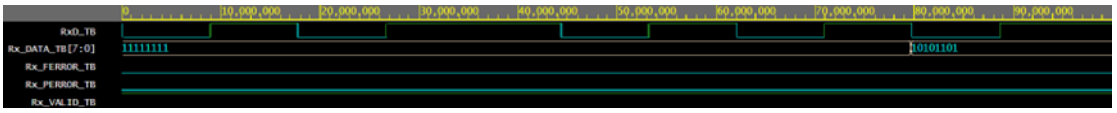
RxD_TB = 0; #8850 //perror

RxD_TB = 1; //end bit

Εισάγουμε το ferror στο 4ο data bit που στέλνουμε καθώς αλλάζει η τιμή του από 1 σε 0 στην μέση του T. Όπως βλέπουμε παρακάτω ο δέκτης εντοπίζει το σφάλμα και δινει την τιμή 1 στο *FERROR*, συνεπώς την τιμή 0 στο *VALID*. Συγκεκριμένα το sample παίρνει την τιμή 1100.

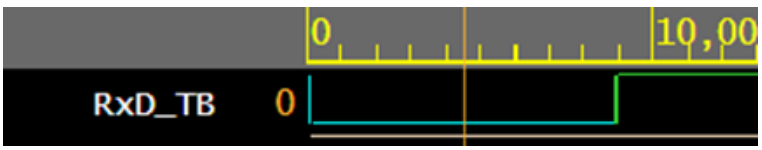


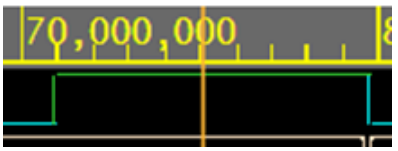
Εάν αφαιρέσουμε το *FERROR* από την προσομοίωση και βάλουμε στο 4ο bit την τιμή 1 τότε έχουμε την παρακάτω κυματομορφή.



Παρατηρούμε ότι η έξοδος *Rx_DATA_TB* έχει την επιθυμητή τιμή και στις 2 περιπτώσεις.

Στις παρακάτω κυματομορφές βλέπουμε ότι η δειγματοληψία γίνεται στις σωστές χρονικές στιγμές σε όλα τα bit από το πρώτο ως το τελευταίο.





EDA Playground link: https://edaplayground.com/x/tZLD

UART Channel:

Εδώ γίνεται η συνένωση Αποστολέα και Δέκτη. Αρχικά, δημιουργούμε 2 modules για τον *baud_controller*, έναν για τον αποστολέα και έναν για τον δέκτη. Σαν είσοδο στο channel έχουμε το reset, το clock, το πακέτο των 2 χαρακτήρων (8 bit) και το baud_select και σαν έξοδο έχουμε το πακέτων χαρακτήρων μετά την διαδικασία αποστολής και λήψης δεδομένων.

Στο πρώτο always block αρχικοποιούμε τους reg μας όταν έρχεται το reset από το testbench και όταν πέφτει το reset ενεργοποιούνται ο αποστολέας και ο δέκτης. Όταν έρθει σήμα ότι ο αποστολέας είναι απασχολημένος πέφτει το *Tx_WR* έτσι ώστε εάν έρθουν νέα

δεδομένα δεν θα τα παραλάβει ο αποστολέας καθώς είναι ήδη σε διαδικασία αποστολής δεδομένων. Στη συνέχεα, φτιάχνουμε τα instance DUT και τελικά σε ένα always block δίνουμε στην έξοδο το αποτέλεσμα του δέκτη εάν η λήψη δεδομένων ήταν σωστή, αλλιώς δίνουμε την κωδικοποίηση του συμβόλου *FF*.

EDA Playground link: https://edaplayground.com/x/qvqG

Προβλήματα που συναντήσαμε στο Μέρος Β:

Προσπαθήσουμε να υλοποιήσουμε την ανάθεση τιμών στον receiver στα 8 data bit με shift right, όμως αυτο δεν λειτούργησε γι αυτό το κάναμε διαφορετικά με το data_counter.

Το parity bit δεν βγάζει σωστά αποτελέσματα.

Δεν μπορέσαμε να κατεβάσουμε το Quartus και να δουλέψουμε εκεί καθώς είχαμε ένα πρόβλημα με το lisence. Έτσι δεν βγάλαμε τα schematic που ζητούνται καθώς στο EDA Playground χρησιμοποιήσαμε το Yosys το οποίο δεν μπορεί να κάνει synthesis με παραπάνω από ένα module. Έτσι δεν βγάλαμε τα σχηματικά διαγράμματα.

ΜΕΡΟΣ Γ’

Στο τελικό μέρος της εργασίας γίνεται η συνένωση των πρώτων 2 part. Στο project μας έχουμε όλα τα module του channel και τα modules του πρώτου μέρους. Από το testbench θα μεταδίδουμε τους 4 χαρακτήρες και μέσα στο design θα σπάμε το μήνυμα σε 2 πακέτα των 8 bit. Θα στέλνουμε ένα πακέτο τη φορά στο channel και θα αποθηκεύουμε την έξοδο σε μια μεταβλητή τύπου reg. Όταν ολοκληρωθεί η διαδικασία του καναλιού και των 4 χαρακτήρων θα σπάσουμε κάθε πακέτο των 8 bit σε 2 υπο-πακέτα των 4 bit τα οποία θα μπαίνουν ως είσοδο στον FourDigitLEDdriver. Έτσι στις κυματομορφές θα παρατηρούμε την αλλαγή των a, b, c, d, e, f, g ανάλογα με ποιον χαρακτήρα παίρνει σαν είσοδο ο FourDigitLEDdriver.

Δυστυχώς δεν προλάβαμε να υλοποιήσουμε το Μέρος Γ της εργασίας πιστεύουμε, όμως, ότι εάν υπήρχε ο χρόνος θα το ολοκληρώναμε.