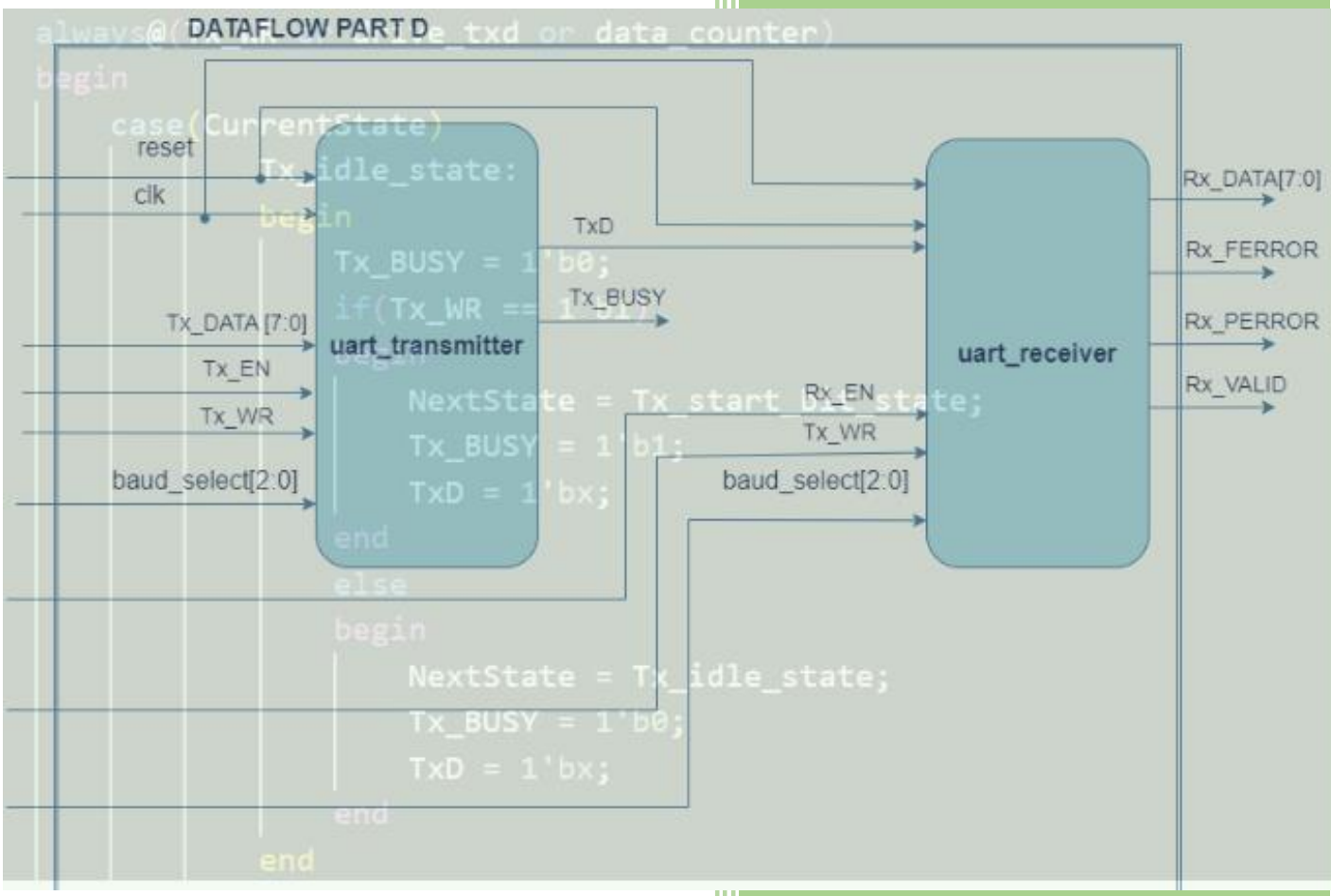
2η Εργαστηριακή Εργασία
Υλοποίηση Μονάδας Γενικού Ασύγχρονου Δέκτη Αποστολέα



Ον/μο: Γαρυφαλιά Γεωργιτζίκη

ΑΕΜ: 03218

Ημ/νία: 16/11/2022

ΠΕΡΙΕΧΟΜΕΝΑ

ΠΕΡΙΛΗΨΗ

Στην εργασία με τίτλο «Υλοποίηση Μονάδας Γενικού Ασύγχρονου Δέκτη Αποστολέα», θα δούμε την υλοποίηση της παραπάνω μονάδας με την χρήση του πρωτοκόλλου **UART** (Universal Asynchronous Receiver Transmitter). Η σειριακή επικοινωνία επιτυγχάνεται με έναν UART Αποστολέα και έναν UART Δέκτη. Η εργασία χωρίζεται σε τέσσερα μέρη, τα οποία είναι:

1. Μέρος Α: Ελεγκτής Baud Rate
2. Μέρος Β: Υλοποίηση UART Αποστολέα (Transmitter)
3. Μέρος Γ: Υλοποίηση UART Δέκτη (Receiver)
4. Μέρος Δ: Υλοποίηση UART Αποστολέα-Δέκτη για Σειριακή Μεταφορά Δεδομένων

ΕΙΣΑΓΩΓΗ

Ο στόχος στην εργασία αυτή είναι η δημιουργία ενός συστήματος σειριακής επικοινωνίας με βάση το πρωτόκολλο UART, το οποίο αποτελείται από έναν UART Αποστολέα που θα στέλνει τα δεδομένα και έναν UART Δέκτη που θα λαμβάνει τα δεδομένα. Η αλληλουχία ελέγχου που μεταφέρεται από την μία πλευρά στην άλλη είναι:

10101010(AA), 01010101(55), 11001100(CC) και 10001001(89)

Το UART είναι ένα πρωτόκολλο ασύγχρονης λήψης και μετάδοσης που χρησιμοποιείται για εφαρμογές σειριακής επικοινωνίας. Η ασύγχρονη επικοινωνία επιτυγχάνεται μέσω μιας ενσύρματης σύνδεσης ενός bit , όπου ο Αποστολέας την οδηγεί, και ο Δέκτης την δειγματοληπτεί και την εξετάζει.

ΜΕΡΟΣ Α: Ελεγκτής Baud Rate

Στο μέρος Α της εργασίας θα δούμε αναλυτικά πως υλοποιήθηκε ο **ελεγκτής Baud Rate.** Πιο συγκεκριμένα η λειτουργία του ελεγκτή αυτού είναι να ελέγξει την ταχύτητα (σε μονάδες Baud(bits/sec)) με την οποία επικοινωνεί ο Transmitter με τον Receiver. Έτσι η κάθε ταχύτητα επικοινωνίας του UART κωδικοποιείται από ένα σήμα 3-bit, το **BAUD_SEL,** όπως φαίνεται στον παρακάτω πίνακα.

BAUD_SEL	Baud Rate *(bits/sec)*
0 0 0	300
0 0 1	1200
0 1 0	4800
0 1 1	9600
1 0 0	19200
1 0 1	38400
1 1 0	57600
1 1 1	115200

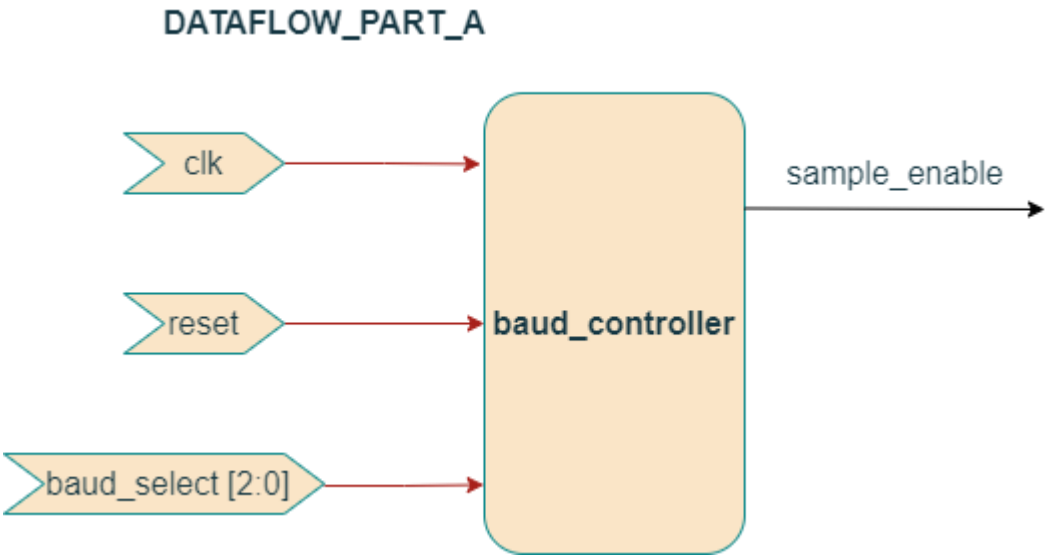
Για να υπολογίσουμε την περίοδο μετάδοσης κάθε ψηφίου από τον Transmitter χρησιμοποιήσαμε τον τύπο $T = \frac{1}{BaudRate}$. Ο UART Transmitter δειγματοληπτεί και εξετάζει με ρυθμό 16 φορές μεγαλύτερο από τον Receiver για μία συγκεκριμένη ταχύτητα. Στον παρακάτω πίνακα φαίνεται ο αριθμός κύκλων ρολογιού του Transmitter και του Receiver ώστε να λειτουργούν σε ένα συγκεκριμένο Baud Rate και υπολογίστηκαν από τον τύπο $\frac{100\times10^6}{16\times BaudRATE}$ όπου 100×10^6 είναι η συχνότητα 100MHz του ρολογιού της FPGA. Για λόγους ευκολίας έγινε στρογγυλοποίηση κατά ένα ψηφίο προς τα πάνω και στον πίνακα αναγράφεται και το σφάλμα της.

Κατά την διάρκεια της εργασίας παρατήρησα ότι η στρογγυλοποίηση προς τα πάνω μάλλον δεν είναι η βέλτιστη λύση καθώς όπως φαίνεται από τον πίνακα το σφάλμα στις περισσότερες περιπτώσεις είναι μεγαλύτερο του 0,5. Λύση σε αυτό θα ήταν η στρογγυλοποίηση να γίνει

είτε προς τα πάνω είτε προς τα κάτω ανάλογα με τον κοντινότερο ακέραιο, ή να γίνει στρογγυλοποίηση όλων προς τα κάτω. Ωστόσο, λόγω πίεσης χρόνου , στον πίνακα παρακάτω αλλά και στον κώδικα οι τιμές έχουν παραμείνει με στρογγυλοποίηση προς τα πάνω.

Baud Rate (bits/sec)	Transmitter Clock Cycles (decimal)	Error Value	Transmitter Clock Cycles (binary)	Receiver Clock Cycles (decimal)	Error Value	Receiver Clock Cycles (binary)
300	333.334	0,67	0101 0001 0110 0001 0110	20.834	0,67	0101 0001 0110 0010
1.200	83.334	0,67	0001 0100 0101 1000 0110	5.209	0,67	0101 0010 0000 1001
4.800	20.834	0,67	0101 0001 0110 0010	1.303	0,917	0101 0001 0111
9.600	10.417	0,34	0010 1000 1011 0001	652	0,9584	0010 1000 1100
19.200	5.209	0,67	0101 0010 0000 1001	326	0,48	0001 0100 0110
38.400	2.605	0,84	1010 0010 1101	163	0,24	1010 0011
57.600	1.737	0,89	0110 1100 1001	109	0,49	0110 1101
115.200	869	0,945	0011 0110 0101	55	0,75	0011 0111

Το dataflow του μέρους Α φαίνεται παρακάτω:



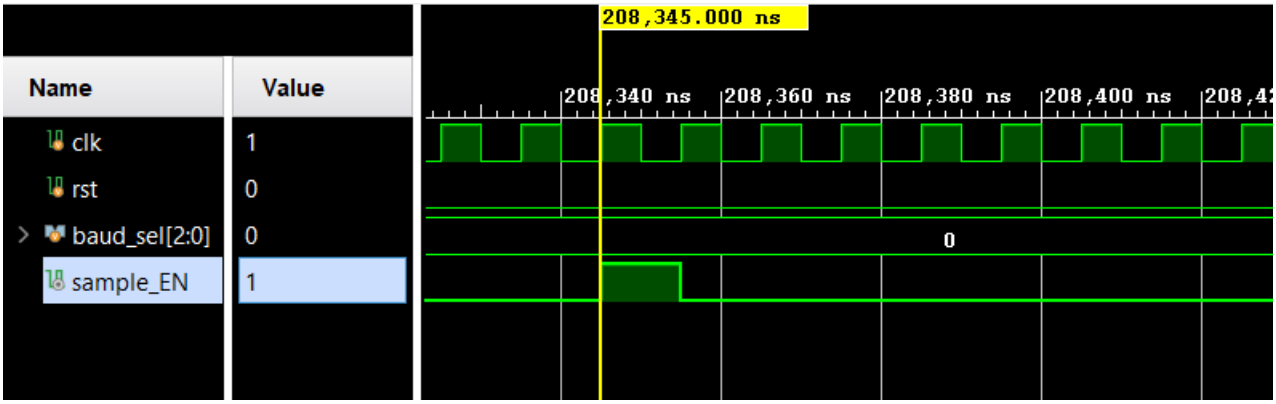
Όπως βλέπουμε, οι είσοδοι του **module baud_controller,** είναι το ρολόι της FPGA **clk**, το **reset** και το **baud_select** που είναι η 3-bit κωδικοποίηση που αναφέραμε παραπάνω, ενώ έξοδος είναι το σήμα **sample_enable**, το οποίο, ανάλογα με το baud_select, όταν γίνεται 1 δείχνει ότι ο μετρητής των κύκλων ρολογιού μέσα στο module έφτασε στην μέγιστη τιμή του. Ακολουθεί το αντίστοιχο τμήμα του κώδικα που υλοποιεί το παραπάνω:

```
/o counter tha metraei ta bits analoga me to baud select
always@(baud_select)
begin
    case(baud_select)
    //o counter lamvanei times me vasei to baud_rate apo ton typo
10^8/(16*baud_rate)
    baud_zero: max_count = 15'b101_0001_0110_0010;
    baud_one: max_count = 15'b001_0100_0101_1001;
    .
    .
    .
    baud_seven: max_count = 15'b000_0000_0011_0111;
    endcase
end
```

```
//ylopoihsh counter gia thn katametrhsh kyklwn rologiou analoga me to baud rate
always@(posedge clk)
begin
    if(count == max_count)
    begin
```

```
        count <= 15'b000_0000_0000_0000;
        sample_ENABLE <= 1'b1;
    end
    else
    begin
        count <= count + 15'b000_0000_0000_0001;
        sample_ENABLE <= 1'b0;
    end
end
```

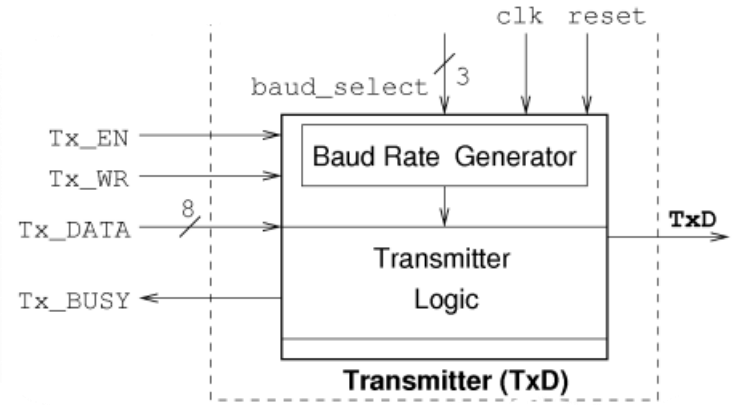
Σχετικά με το **testbench** του **baud_controller,** η υλοποίηση του είναι αρκετά απλή καθώς τρέχει το **module baud_controller** για ένα συγκεκριμένο **baud_sel,** την φορά ανάλογα με το baud_sel που δεν βρίσκεται σε σχόλια. Στην επόμενη εικόνα φαίνεται ότι για **baud_sel = 3'b000** δηλαδή **baud rate 300 bits/sec** το σήμα **sample_ENABLE** γίνεται 1 την χρονική στιγμή 208.345 δηλαδή αφού έχει μετρήσει 20.834 κύκλους ρολογιού (20.834*10ns(period)).



Παρακάτω, θα δούμε στα Testbench μια διαφορετική υλοποίηση για την εναλλαγή των baud_rate που θα αλλάζουν αυτόματα ανάλογα με την τιμή ενός μετρητή.

ΜΕΡΟΣ Β: ΥΛΟΠΟΙΗΣΗ UART ΑΠΟΣΤΟΛΕΑ (TRANSMITTER)

Στόχος του UART Transmitter είναι να μεταφέρει το σύμβολο που πήρε από το σύστημα, στον Receiver, και να ενημερώνει το σύστημα εάν είναι διαθέσιμος να λάβει το επόμενο σύμβολο ή βρίσκεται σε κατάσταση μεταφοράς δεδομένων. Τα bits που μεταφέρει ο Transmitter είναι 11 σε αριθμό και αποτελούνται από: **1 start bit** (το bit που σηματοδοτεί την έναρξη επικοινωνίας με τον Receiver), **8 data bit** (τα bit από τα οποία αποτελείται το σύμβολο που μεταφέρεται), **1 parity bit** (το bit που δείχνει αν το σύμβολο που στάλθηκε περιέχει άρτιο ή περιττό αριθμό άσσων και χρησιμεύει στην επαλήθευση σωστής επικοινωνίας), και **1 stop bit** (το bit που σηματοδοτεί το τέλος της επικοινωνίας). Στην παρακάτω εικόνα φαίνονται τα σήματα που έχει ως εισόδους και εξόδους το **module uart_transmitter.**



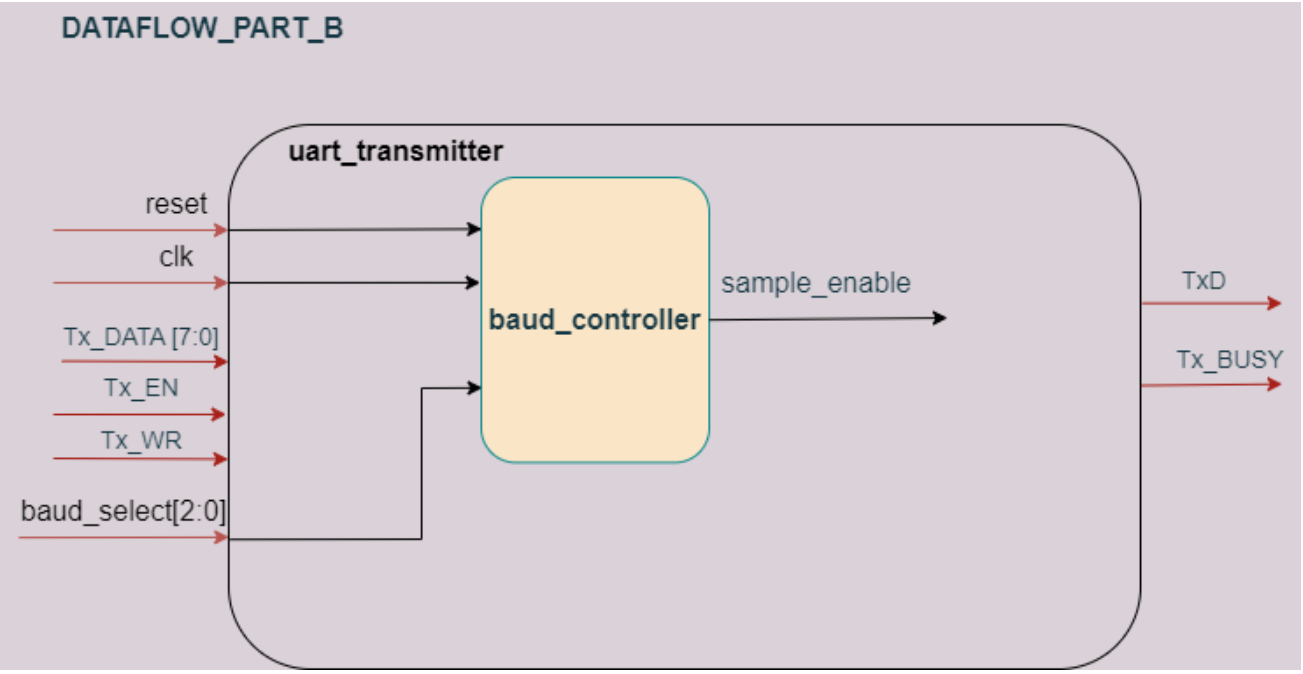
Το κάθε σήμα επιτελεί την εξής λειτουργία:

- **Tx_EN**: Είναι το σήμα ενεργοποίησης του Transmitter και μένει ενεργό από το testbench όσο ο Transmitter είναι σε απλά σε ενεργή κατάσταση ή στέλνει δεδομένα.
- **Tx_WR**: Σήμα που γίνεται 1 για ένα κύκλο και δείχνει ότι τα δεδομένα προς μεταφορά βρίσκονται στο **Tx_DATA**. Επίσης σηματοδοτεί την αρχή της αποστολής των δεδομένων.
- **Tx_BUSY**: Σήμα που στέλνεται από τον Transmitter στο σύστημα και σηματοδοτεί ότι ο Transmitter δεν είναι διαθέσιμος καθώς βρίσκεται σε κατάσταση αποστολής δεδομένων.

Επιπρόσθετα, έχουμε τα:

- **Tx_DATA**: Ενημερώνεται από το testbench με τα δεδομένα προς μεταφορά.
- **TxD**: Έξοδος του Transmitter που περιέχει το bit προς μεταφορά στον Δέκτη.
- **clk**: Το ρολόι με βάση το οποίο εκτελείται η διαδικασία.
- **reset**: Σήμα για την επαναφορά του UART στην αρχική του κατάσταση.
- **baud_select [3:0]**: Κωδικοποίηση 3 bit για το Baud_rate με το οποίο θα γίνει η επικοινωνία.

Παρακάτω φαίνεται το dataflow του μέρους Β:



Όπως βλέπουμε, το module baud_controller γίνεται instantiate μέσα στον transmitter ώστε το σήμα sample_enable να ελέγχει την αποστολή των δεδομένων. Επιπρόσθετα, οι τρείς είσοδοι του transmitter clk, reset και baud_select είναι επίσης είσοδοι στον baud_controller.

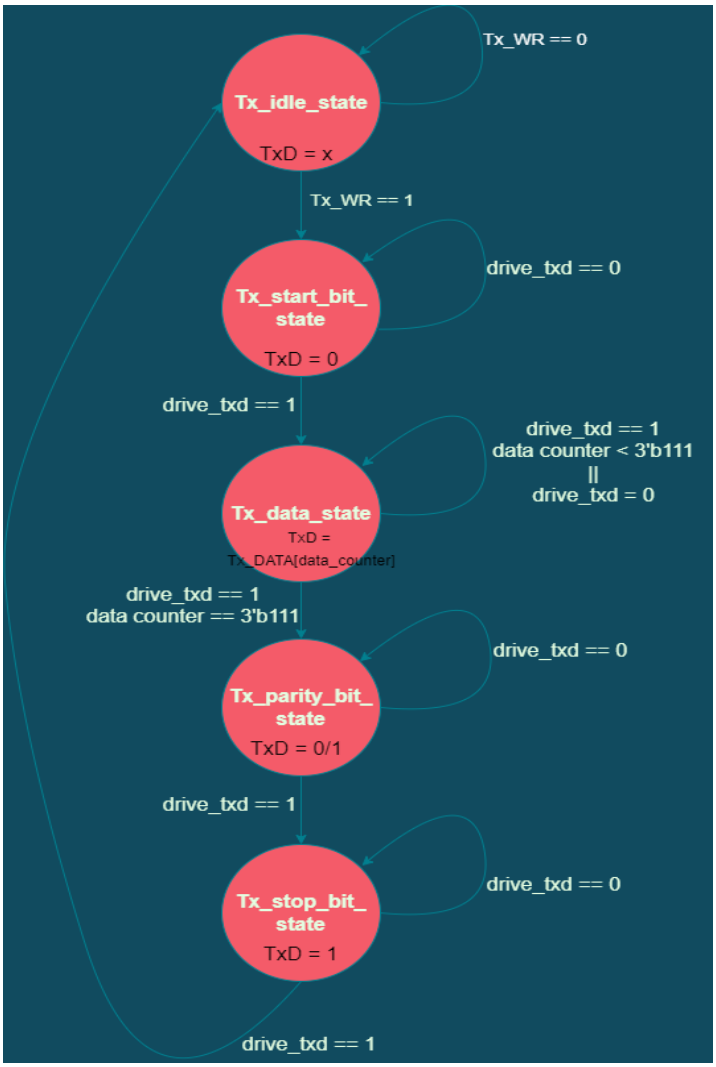
Ο UART Transmitter χρησιμοποιεί μία **FSM Mealy**, 5 καταστάσεων.

Οι καταστάσεις είναι οι εξής:

1. **Tx_idle_state**: Ανενεργή κατάσταση του Transmitter.

2. **Tx_start_bit_state**: Κατάσταση αποστολής του **start bit**
3. **Tx_data_state**: Κατάσταση αποστολής των **bit του συμβόλου**
4. **Tx_parity_bit_state**: Κατάσταση αποστολής του **parity bit**
5. **Tx_stop_bit_state**: Κατάσταση αποστολής του **stop bit**

Από το διάγραμμα της FSM παρακάτω, βλέπουμε ότι η FSM είναι Mealy καθώς η έξοδος εξαρτάται από την τρέχουσα κατάσταση αλλά και από το σήμα-είσοδο **drive_txd**. Η επιλογή της Mealy FSM έγινε διότι για να ελέγξουμε τον ρυθμό δειγματοληψίας χρειαζόμαστε έναν **sample_counter** που μετράει 16 Tx_sample_ENABLE από τον baud_controller,και άρα χρειαζόμαστε ένα σήμα που να οδηγεί την FSM.



Η υλοποίηση της FSM έγινε με την χρήση δυο τμημάτων always, ένα ακολουθιακό και ένα συνδυαστικό.

Το ακολουθιακό τμήμα always που φαίνεται παρακάτω, για reset == 1 αρχικοποιεί την FSM στην κατάσταση Tx_idle state, διαφορετικά αναθέτει στο CurrentState το NextState.

```
always@(posedge clk)
begin
    if(reset == 1)
    begin
        CurrentState <= Tx_idle_state;
    end
    else
    begin
        CurrentState <= NextState;
    end
end
```

Το συνδυαστικό τμήμα always κάθε φορά που αλλάζει τιμή το σήμα Tx_WR ή το σήμα drive_txd ή ο data_counter οδηγεί την FSM με βάση το CurrentState. Επίσης, ανάλογα με την κατάσταση στην οποία βρίσκεται η FSM δίνει στο σήμα Tx_BUSY τιμή 0 ή 1. Ο data_counter μετράει ποιο bit του Tx_DATA[7:0] θα σταλθεί κάθε φορά στην έξοδο αρχίζοντας από το LSB και ο counter_ones μετράει πόσους άσσους έχει κάθε σύμβολο ώστε να οδηγήσει κατάλληλα το parity bit (για περιττό αριθμό άσσων TxD = 1, διαφορετικά TxD = 0). Ακολουθεί το αντίστοιχο τμήμα always:

```
always@(Tx_WR or drive_txd or data_counter)
begin
    case(CurrentState)
            Tx_idle_state:
            begin
                Tx_BUSY = 1'b0;
                if(Tx_WR == 1'b1)
                begin
                    NextState = Tx_start_bit_state;
                    Tx_BUSY = 1'b1;
                    TxD = 1'bx;
                end
                else
                begin
                    NextState = Tx_idle_state;
                    Tx_BUSY = 1'b0;
                    TxD = 1'bx;
                end
```

```
            end
            Tx_start_bit_state://stelnei to start bit gia na ksekinisei h metafora
dedomenwn
            begin
                TxD = 1'b0;//start bit
                Tx_BUSY = 1'b1;//transmitter is busy
                if(drive_txd == 1)
                begin
                    NextState = Tx_data_state;
                end
                else
                begin
                    NextState = Tx_start_bit_state;
                end
            end

            Tx_data_state://stelnei ta data bit pros bit apo to LSB sto MSB
            begin
                data_state = 1'b1;

                TxD = Tx_DATA[data_counter];
                Tx_BUSY = 1'b1;//transmitter is busy

                if(drive_txd == 1 && data_counter == 3'b111)
                begin
                    data_state = 1'b0;
                    NextState = Tx_parity_bit_state;
                    if(TxD == 1'b1) //counter gia thn katametrhsh twn asswn tou
symvolou
                    begin
                        counter_ones = counter_ones + 4'b0001;
                    end
                end
                else if(drive_txd == 1 && data_counter < 3'b111)
                begin
                    data_state = 1'b0;
                    NextState = Tx_data_state;
                    if(TxD == 1'b1) //counter gia thn katametrhsh twn asswn tou
symvolou
                    begin
                       counter_ones = counter_ones + 4'b0001;
                    end
                end
                else
                begin
                    NextState = Tx_data_state;
                end
            end
```

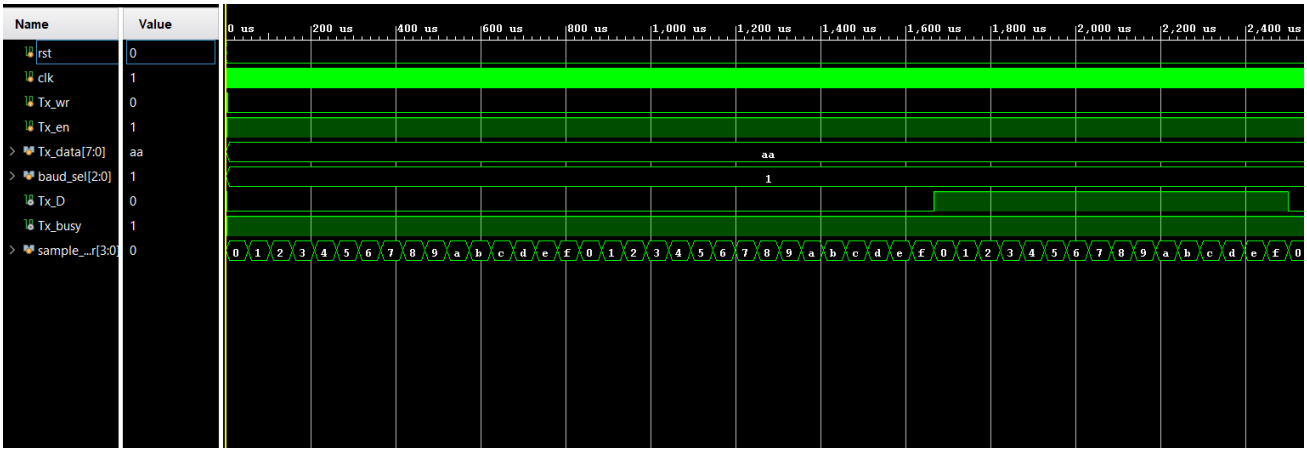
```
            Tx_parity_bit_state://stelnei to parity bit
            begin
                Tx_BUSY = 1'b1;//transmitter is busy
                if(counter_ones[0] == 1'b1) //sinthiki gia na doume an to symvolo
exei peritto h zygo arithmo asswn
                begin
                    TxD = 1'b1;//parity bit == 1 tote perittos arithmos asswn
                end
                else
                begin
                    TxD = 1'b0;//parity bit == 0 tote zygos arithmos asswn
                end

                if(drive_txd == 1)
                begin
                    NextState = Tx_stop_bit_state;
                end
                else
                begin
                    NextState = Tx_parity_bit_state;
                end
            end

            Tx_stop_bit_state://stelnei to stop bit kai deixnei tin liksi tis
metaforaw dedwmenwn
            begin
                TxD = 1'b1;//stop bit
                Tx_BUSY = 1'b1;//transmitter is busy

                if(drive_txd == 1)
                begin

                    counter_ones = 4'b0000;
                    NextState = Tx_idle_state;
                end
                else
                begin
                    NextState = Tx_stop_bit_state;
                end
            end
            default:
            begin
                TxD = 1'bx;
                NextState = Tx_idle_state;
            end
        endcase
end
```

Το σήμα **drive_txd** οδηγείται από τον **sample_counter** στην παρακάτω always, ο οποίος μετράει 16 **Tx_sample_enable** από τον **baud_controller** ώστε στο μέρος Γ ο Receiver να δειγματοληπτεί με ρυθμό 16 φορές μεγαλύτερο από τον Transmitter.

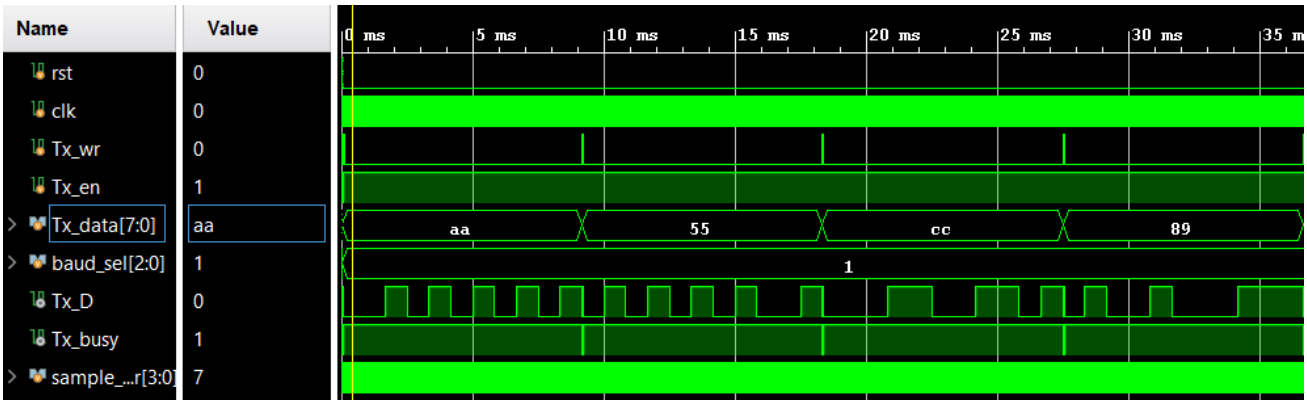
```
always@(posedge clk)
begin
    if(reset == 1)
    begin
        sample_counter <= 4'b0000;
    end
    else if(Tx_sample_ENABLE == 1'b1 && Tx_BUSY == 1'b1 && Tx_EN == 1'b1)
    begin
        sample_counter <= sample_counter + 4'b0001;
    end
end
always@(sample_counter)
begin
    if(sample_counter == 4'b1111)
    begin
        drive_txd = 1'b1;
    end
    else
    begin
        drive_txd = 1'b0;
    end
end
```

Για παράδειγμα, στην παρακάτω εικόνα, βλέπουμε ότι ο sample_counter μετράει από το 0 έως το 15 και στέλνει το ίδιο bit σε αυτό τον χρόνο.

Εδώ συγκεκριμένα, για το σύμβολο AA με κωδικοποίηση 8’b10101010, στέλνει αρχικά το start bit = 0, μετά το Tx_data[0] = 0, Tx_data[1] = 1,…., Tx_data[7] = 1 από το LSB στο MSB, το parity bit = 0 και το stop bit = 1.

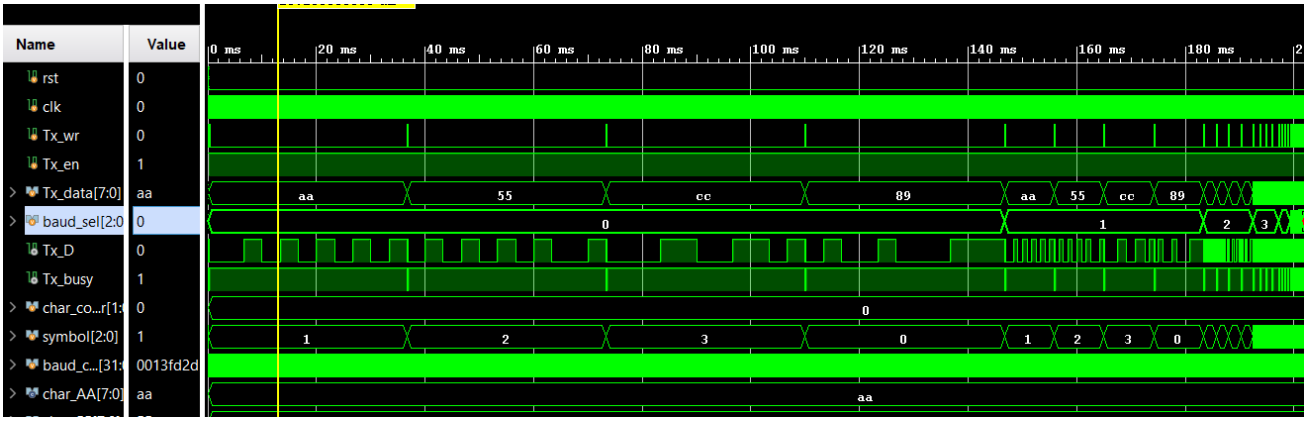


Ακολουθεί εικόνα από την αποστολή και των τεσσάρων συμβόλων (AA, 55, CC, 89):



Κατά την αποστολή ενός συμβόλου το σήμα Tx_BUSY παραμένει 1 και πέφτει μόνο όταν έχει τελειώσει η αποστολή του συγκεκριμένου συμβόλου. Αντίθετα, το Tx_WR σηκώνεται μόνο για έναν κύκλο, αφού τα δεδομένα ενός συμβόλου έχουν περαστεί στο Tx_DATA και είναι έτοιμα προς αποστολή.

Παρακάτω φαίνεται και η αποστολή των δεδομένων για όλα τα baud rate:



Όσον αφορά την επαλήθευση της υλοποίησης, το testbench δίνει στον baud_controller το Baud rate με το οποίο θα στείλει τα δεδομένα ο Transmitter μέσω της 3-bit κωδικοποίησης baud_sel και στην συνέχεια αφού έχει ενεργοποιηθεί ο αποστολέας (ενεργοποιείται μέσα σε initial block στο testbench) και δεν βρίσκεται σε κατάσταση αποστολής δεδομένων (Tx_en == 1 και Tx_busy == 0), γράφει τα bits του συμβόλου στο Tx_data, «σηκώνει» το σήμα Tx_wr για έναν κύκλο και προχωράει στο επόμενο σύμβολο προς αποστολή.

```
always@(posedge clk)
begin
    if(Tx_en == 1'b1 && Tx_busy == 1'b0 )
    begin
        case(symbol)
            s_AA: Tx_data <= char_AA;
            s_55: Tx_data <= char_55;
            s_CC: Tx_data <= char_CC;
            s_89: Tx_data <= char_89;
            default: Tx_data <= 8'b00000000;
        endcase

        Tx_wr <= 1'b1;
        #(`period)Tx_wr <= 1'b0;

        if(symbol == s_89)
        begin
            symbol <= 3'b100;
        end
        else
        begin
            symbol <= symbol + 3'b001;
        end
    end
    else if(Tx_busy == 1'b1)
    begin
        Tx_en <= 1'b1;
    end
end
```

Για να αλλάξει ο baud_controller το baud_rate, αφού έχουν μεταδοθεί όλα τα δεδομένα (AA, 55, CC, 89), μέτρησα από τις κυματομορφές τους χρόνους που κάνει να σταλθεί ένα πακέτο για κάθε ένα από τα 8

διαφορετικά baud rate , τους μετέτρεψα σε κύκλους ρολογιού και υλοποίησα έναν μετρητή που μετράει αυτούς του κύκλους και δίνει το επόμενο baud_rate την κατάλληλη στιγμή.

```
always@(posedge clk)
begin
    if(rst == 1'b1 || baud_counter == 32'b0000_0001_0011_0011_1101_1100_0100_0100)
    begin
        baud_counter <= 32'b00000000000000000000000000000000;
    end
    else if(Tx_en == 1'b1)
    begin
        baud_counter <= baud_counter + 32'b00000000000000000000000000000001;
    end
end
always@(baud_counter)
begin
    case(baud_counter)
    baud_one:
        begin
            baud_sel = 3'b001;
        end
     .
     .
     .
      endcase
end
```

Τέλος η always που υλοποιεί το ρολόι φαίνεται παρακάτω και έχει χρησιμοποιηθεί σε όλα τα υπόλοιπα testbench.

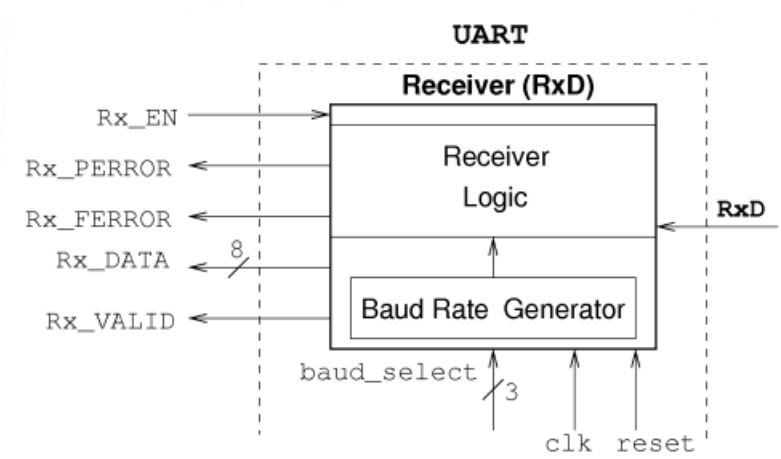
```
always
begin
      #(`period/2)clk = ~clk;
end
```

Κατά την υλοποίηση της FSM υπήρξε ένα πρόβλημα καθώς από το implementation φάνηκε να δημιουργείται 1 latch στο data_counter, τον μετρητή για την θέση του bit στο σύμβολο προς μεταφορά. Δυστυχώς, ύστερα από κάποιες δοκιμές δεν κατάφερα να διορθώσω το πρόβλημα καθώς παρότι πρόσθεσα μία ‘else’ στον κώδικα για να καλύπτονται όλες οι περιπτώσεις δεν άλλαξε κάτι και το latch εξακολουθεί να εμφανίζεται

ως warning στο Vivado. Ωστόσο, ο Transmitter δουλεύει κανονικά και στέλνει τα σωστά δεδομένα στον επιθυμητό χρόνο ανάλογα το baud rate.

ΜΕΡΟΣ Γ: ΥΛΟΠΟΙΗΣΗ UART ΔΕΚΤΗ (RECEIVER)

Στο μέρος Γ της εργασίας μας ζητήθηκε η υλοποίηση ενός UART Δέκτη (Receiver) ο οποίος λαμβάνει τα δεδομένα που έχει στείλει ο Transmitter που περιγράψαμε προηγουμένως. Ο Receiver όπως προαναφέρθηκε, δειγματοληπτεί τα δεδομένα που λαμβάνει με ρυθμό 16 φορές μεγαλύτερο από ότι στέλνει ο Transmitter, δηλαδή λειτουργεί με ταχύτητα **16*Baud Rate**. Επιπλέον, ο Receiver ελέγχει εάν τα δεδομένα που έλαβε είναι σωστά προτού ολοκληρώσει την διαδικασία παραλαβής.



Οι είσοδοι του Receiver είναι:

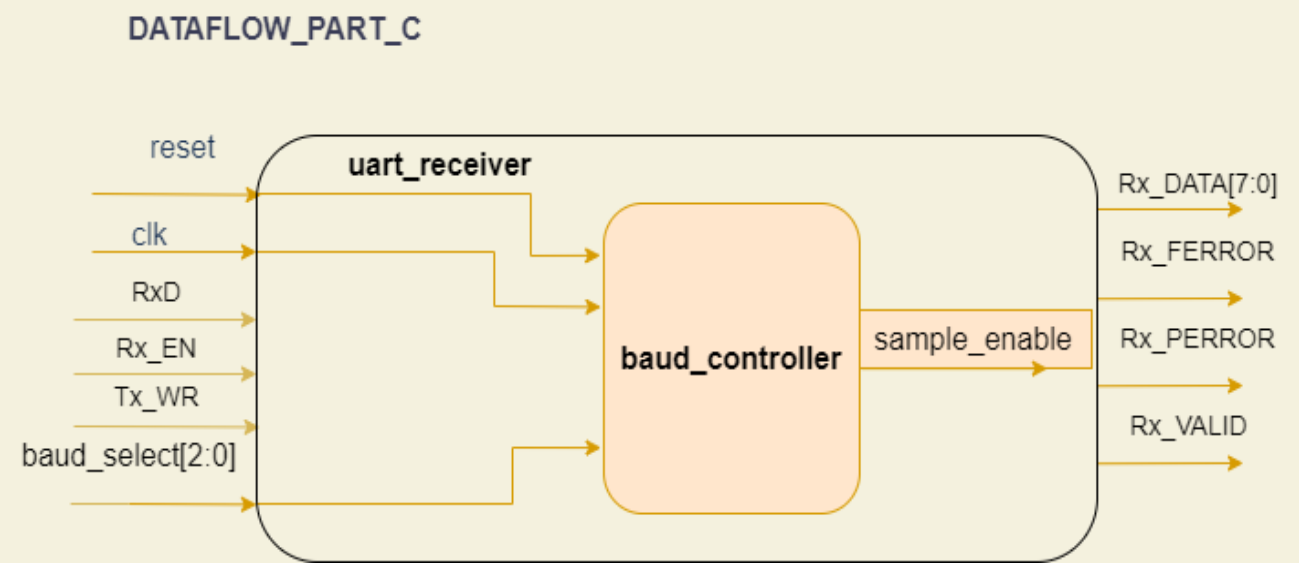
- **Rx_EN:** Σήμα ενεργοποίησης του Receiver.
- **RxD:** Μεταφέρει στον Receiver το bit που δέχτηκε από τον Transmitter..
- **baud_select [3:0]**: Κωδικοποίηση 3 bit για το Baud_rate με το οποίο θα γίνει η επικοινωνία.
- **clk**: Το ρολόι με βάση το οποίο εκτελείται η διαδικασία.
- **reset**: Σήμα για την επαναφορά του UART στην αρχική του κατάσταση.

Οι έξοδοι του Receiver είναι:

- **Rx_DATA[7:0]:** Καταχωρητής 8 bit του σύμβολο που έλαβε ο Receiver.
- **Rx_VALID:** Σήμα ότι ο Receiver έλαβε επιτυχώς το σύμβολο.

- **Rx_PERROR:** Σηματοδοτεί ότι υπήρξε σφάλμα κατά την ισοτιμία των bit που στάλθηκαν με αυτά που έλαβε ο Receiver (parity bit), άρα τα δεδομένα δεν είναι σωστά και πρέπει να αγνοηθούν.
- **Rx_FERROR:** Σηματοδοτεί λάθος κατά την λήψη του start bit ή του stop bit.

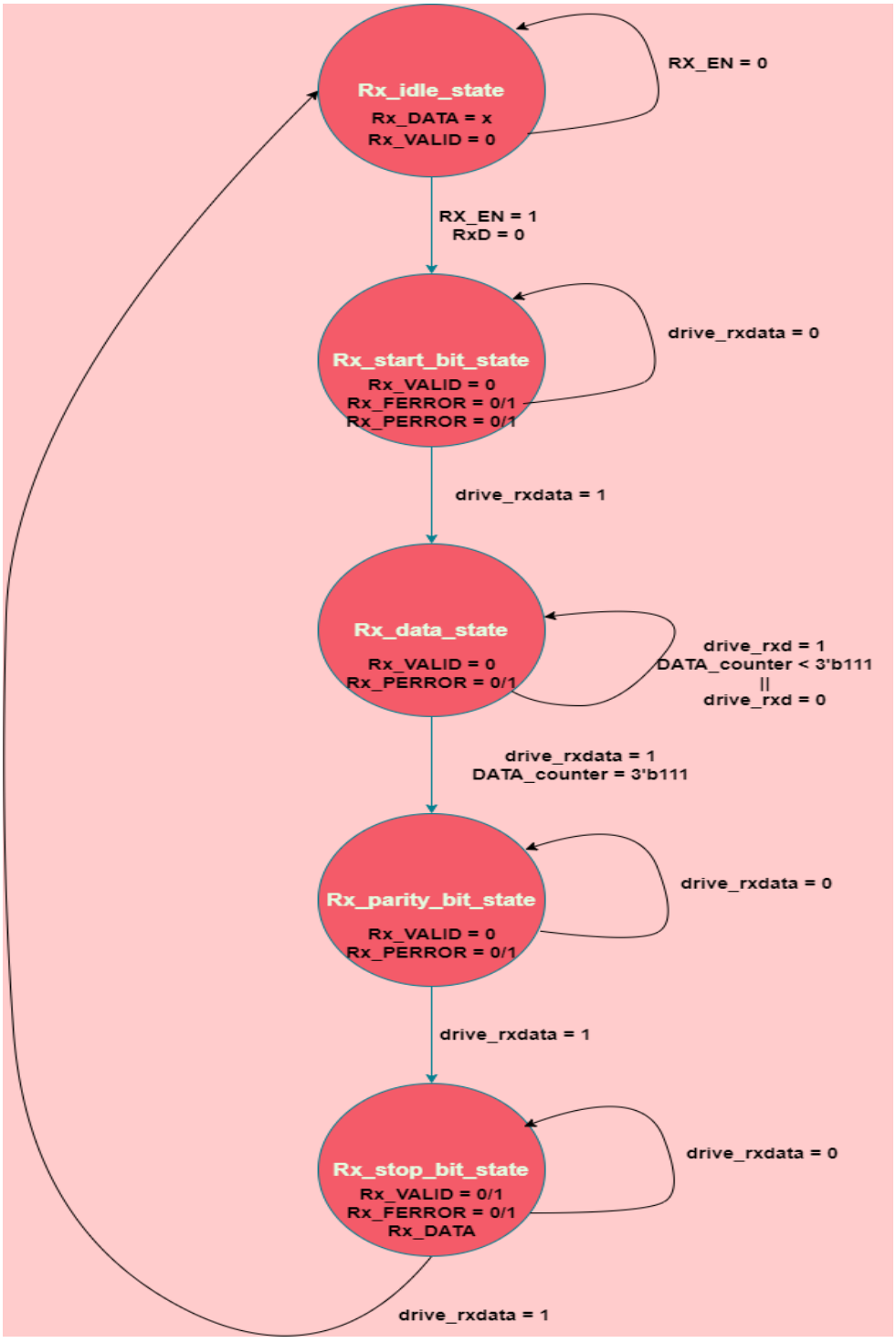
Το σχετικό dataflow:



Παρατηρούμε ότι, το module baud_controller γίνεται instantiate μέσα στον receiver ώστε το σήμα sample_enable να ελέγχει την παραλαβή των δεδομένων με τον κατάλληλο ρυθμό δειγματοληψίας. Ακόμη, οι τρείς είσοδοι του receiver clk, reset και baud_select είναι επίσης είσοδοι στον baud_controller.

Όσον αφορά την **FSM** του Receiver είναι και αυτή **Mealy**, αφού η έξοδος εξαρτάται από την τρέχουσα κατάσταση και την είσοδο, που σε αυτή την περίπτωση είναι το σήμα **drive_rxdata** και το σήμα **check_errors.** Το σήμα **drive_rxdata** οδηγεί είτε στην επόμενη κατάσταση είτε στο επόμενο bit προς παραλαβή, ενώ το σήμα **check_errors** ελέγχει για Parity Error ή Framing Error στο μέσον του κάθε bit.

Παρακάτω φαίνεται το διάγραμμα της FSM του Receiver:



Όπως βλέπουμε υπάρχουν 5 καταστάσεις οι οποίες είναι:

1. **Rx_idle_state**: Ανενεργή κατάσταση του Receiver
2. **Rx_start_bit_state**: Κατάσταση παραλαβής και ελέγχου του **start bit**
3. **Rx_data_state**: Κατάσταση παραλαβής των **bit του συμβόλου**
4. **Rx_parity_bit_state**: Κατάσταση παραλαβής και ελέγχου του **parity bit**

5. **Tx_stop_bit_state**: Κατάσταση παραλαβής και ελέγχου του **stop bit**

Ο κώδικας υλοποίησης της FSM που παρατίθεται παρακάτω, αποτελείται από δύο τμήματα always, το πρώτο ακολουθιακής λογικής και το δεύτερο συνδυαστικής λογικής. Το ακολουθιακό τμήμα always για **reset** == 1 αρχικοποιεί την FSM στην κατάσταση Rx_idle state, διαφορετικά δίνει στην FSM την επόμενη κατάσταση.

```
always@(posedge clk)
begin
    if(reset == 1)
    begin
        CurrentState <= Rx_idle_state;
    end
    else
    begin
        CurrentState <= NextState;
    end
end
```

Το συνδυαστικό τμήμα always εναλλάσσει τις 5 καταστάσεις της FSM ανάλογα με την είσοδο που δέχεται. Εάν η FSM βρίσκεται στο Rx_start_bit_state ή Rx_stop_bit_state και εντοπιστεί σφάλμα κατά την παραλαβή των αντίστοιχων bit, τότε σηκώνεται το σήμα **Rx_FERROR** και η FSM οδηγείται στο Rx_idle_state. Επίσης αν εντοπιστεί σφάλμα **Rx_PERROR** στην κατάσταση Rx_parity_bit_state η FSM πάλι οδηγείται στο Rx_idle_state. Διαφορετικά, τα δεδομένα που παραλήφθηκαν οδηγούνται στο **Rx_DATA**[7:0] και το σήμα **Rx_VALID**

γίνεται 1. Ακολουθεί το αντίστοιχο τμήμα always:

```
always@(Rx_EN or drive_rxdata or check_errors or sample_counter or data_state)
begin
    case(CurrentState)
        Rx_idle_state:
        begin
            Rx_VALID = 1'b0;
            if(Rx_EN == 1'b1 && RxD == 1'b0)//eftase to start bit
            begin
               NextState = Rx_start_bit_state;
               received_data = 7'b0000000;
               Rx_DATA = 7'b0000000;
```

```
               counter_ones = 4'b0000;
            end
            else
            begin
                NextState = Rx_idle_state;
                received_data = 7'b0000000;
            end
        end
        Rx_start_bit_state:
        begin
            Rx_VALID = 1'b0;
            if (check_errors == 1'b1 && RxD != 1'b0)//elegxos an einai swsto to
start bit
            begin
                Rx_FERROR = 1'b1; //lathos sto start bit
                NextState = Rx_idle_state;
            end
            else if(drive_rxdata == 1'b1)
            begin
                NextState = Rx_data_state;
                Rx_FERROR = 1'b0;
            end
            else
            begin
              NextState = Rx_start_bit_state;
              Rx_FERROR = 1'b0;
            end
        end

        Rx_data_state:
        begin
            data_state = 1'b1;
            Rx_VALID = 1'b0;
            if(sample_counter == 4'b0001)
            begin
                received_data[DATA_counter] = RxD;
            end
            if(check_errors == 1'b1 && received_data[DATA_counter] != RxD)//elegxos
sto meson tou bit an einai swsto
            begin
                data_state = 1'b0;
                NextState = Rx_idle_state;
            end
            else if(drive_rxdata == 1'b1 && DATA_counter < 3'b111) //paei sto
epomeno bit
            begin
                data_state = 1'b0;
```

```
                if(received_data[DATA_counter] == 1'b1) //counter gia thn
katametrhsh twn asswn tou symvolou
                begin
                    counter_ones = counter_ones + 4'b0001;
                end
                NextState = Rx_data_state;
            end
            else if(drive_rxdata == 1'b1 && DATA_counter == 3'b111) //paei sto
epomeno state
            begin
                data_state = 1'b0;
                if(received_data[DATA_counter] == 1'b1) //counter gia thn
katametrhsh twn asswn tou symvolou
                begin
                    counter_ones = counter_ones + 4'b0001;
                end
                NextState = Rx_parity_bit_state;
            end
            else //deigmatoleiptei ksana to idio mexri == 16
            begin
                NextState = Rx_data_state;
            end
        end

        Rx_parity_bit_state:
        begin
            Rx_VALID = 1'b0;
            if (check_errors == 1'b1 && RxD != counter_ones[0])//elegxos an einai
swsto to parity bit
            begin
                Rx_PERROR = 1'b1; //lathos sto parity bit
                NextState = Rx_idle_state;
            end
            else if(drive_rxdata == 1'b1)
            begin
                Rx_PERROR = 1'b0;
                NextState = Rx_stop_bit_state;
            end
            else
            begin
                Rx_PERROR = 1'b0;
                NextState = Rx_parity_bit_state;
            end
        end

        Rx_stop_bit_state:
        begin
```

```
            if (check_errors == 1'b1 && RxD != 1'b1)//elegxos an einai swsto to stop
bit
            begin
                Rx_FERROR = 1'b1; //lathos sto stop bit
                Rx_VALID = 1'b0;
                NextState = Rx_idle_state;
            end
            else if(drive_rxdata == 1'b1)
            begin
                Rx_FERROR = 1'b0;
                NextState = Rx_idle_state;
                Rx_VALID = 1'b1; //epityxia
                Rx_DATA = received_data;
                received_data = 7'b0000000;
            end
            else
            begin
                Rx_FERROR = 1'b0;
                NextState = Rx_stop_bit_state;
            end
        end
        default:
        begin
            Rx_DATA = 7'b0000000;
            NextState = Rx_idle_state;
        end
    endcase
end
```

Σχετικά με τα **Rx_PERROR** και **Rx_FERROR**, η σωστή λειτουργία τους επιβεβαιώθηκε όταν κατά την υλοποίηση του Receiver υπήρχε λάθος κατά την παραλαβή δεδομένων και εμφανιζόταν είτε **Rx_PERROR** είτε **Rx_FERROR**. Ωστόσο, δεν αποθήκευσα κάποιο σχετικό ελαττωματικό κομμάτι κώδικα για να το υποβάλλω μαζί με την υπόλοιπη καθώς δεν σκέφτηκα ότι είναι αναγκαίο.

Για το testbench του Receiver χρησιμοποιήθηκε η ίδια δομή για να αλλάξει ο baud_controller το baud_rate, με έναν μετρητή που μετράει τους κύκλους που κάνει να σταλθεί ένα σύμβολο ανάλογα με το baud rate και δίνει το επόμενο baud rate την κατάλληλη στιγμή. Η διαφορά είναι ότι τώρα το testbench μιμείται τον Transmitter, και στέλνει τα σύμβολα προς

μεταφορά όπως θα τα έστελνε αυτός. Γι αυτό το λόγο έχει υλοποιηθεί ένας **counter** που μετράει τους κύκλους ρολογιού που χρειάζεται κάθε σύμβολο για να σταλθεί και μέσα από μία case στέλνει το αντίστοιχο σύμβολο (AA, 55, CC, 89).

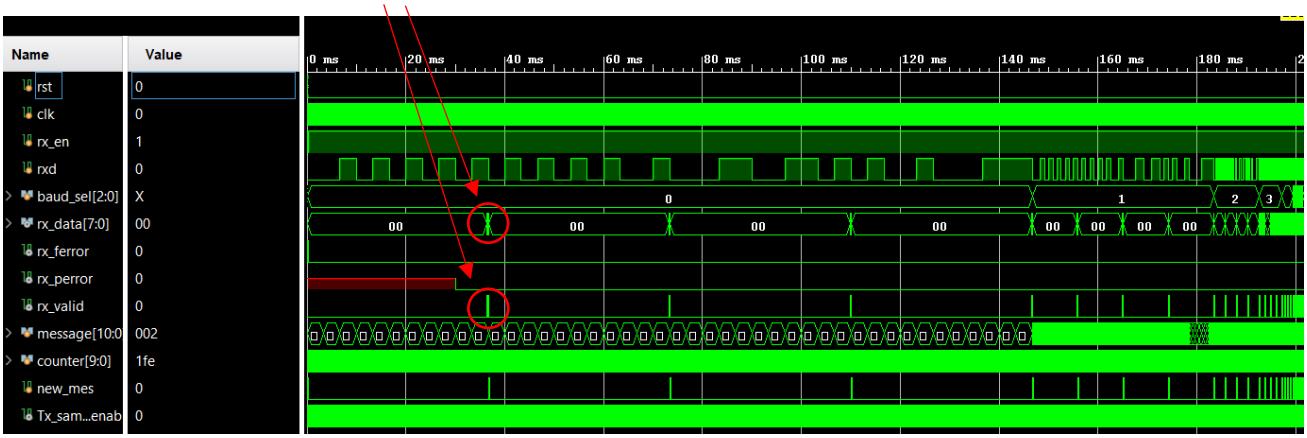
```
always@(counter)
    begin
        case(counter)
            10'b00_0000_0000: begin //0 kykloi
                            message = message_AA;
                            new_mes = 1'b1;
                            end
            10'b00_1011_0000: begin// 16*11 = 176 kykloi
                            message = message_55;
                            new_mes = 1'b1;
                            end
            10'b01_0110_0000: begin// 2*16*11 = 352 kykloi
                            message = message_CC;
                            new_mes = 1'b1;
                            end
            10'b10_0001_0000: begin// 3*16*11 = 528 kykloi
                            message = message_89;
                            new_mes = 1'b1;
                            end
            default: new_mes = 1'b0;
        endcase
    end
```

Τέλος, σε ένα άλλο always block, ο **counter** που προαναφέραμε γίνεται modulo με το 16 ώστε κάθε φορά που το υπόλοιπο θα είναι 0 (άρα ο counter θα είναι πολλαπλάσιο του 16) , θα γίνονται right shift κατά ένα τα bit του συμβόλου και θα στέλνονται στο **rxd**.

```
    always@(counter)
    begin
        if(new_mes == 1'b0)
        begin
            if(counter % 8'b00010000 == 0)// kano modulo 16 wste an to ypoloipo
einai 0, o counter einai pollaplasio tou 16 kai kano right shift to minima
            begin
                //ylopoiisi shifter poy metaferei to bit symvolou sto rxd
                message = message >> 1;
                rxd = message[0];
            end
```

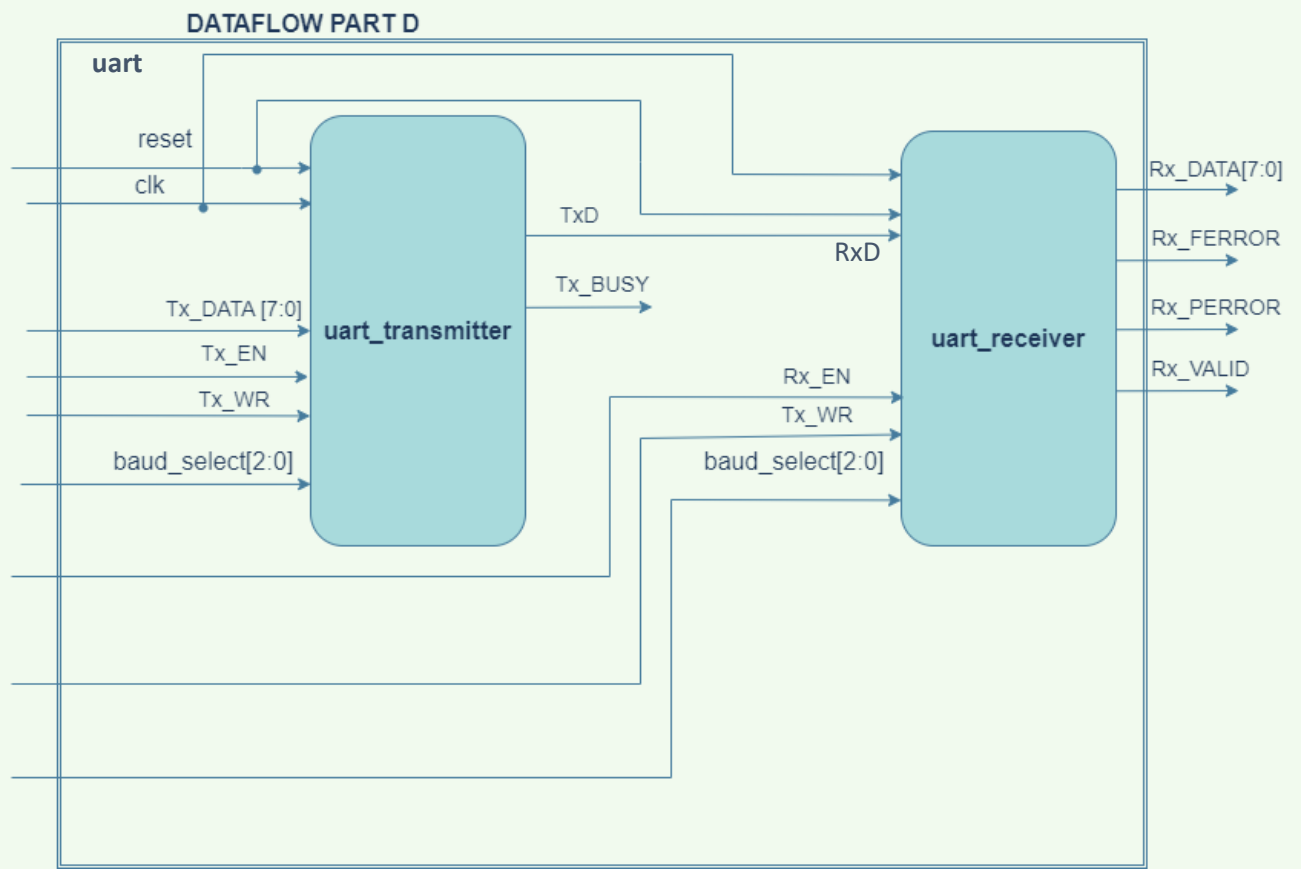
```
        end
        else
        begin
            rxd = message[0];
        end
    end
```

Παρακάτω ακολουθεί εικόνα κατά την παραλαβή όλων των δεδομένων και για τα 8 διαφορετικά baud rate. Παρατηρούμε ότι μέχρι να ληφθεί ολόκληρο το σύμβολο, το **rx_data** παραμένει αρχικοποιημένο σε 0, και παίρνει τιμή μόλις παραλάβει και το stop_bit όπου σηκώνει και το σήμα Rx_VALID μέχρι να επιστρέψει ξανά στο Rx_idle_state.



ΜΕΡΟΣ Δ: Υλοποίηση UART Αποστολέα-Δέκτη για Σειριακή Μεταφορά Δεδομένων

Στο τελευταίο μέρος της εργασίας στόχος είναι η κατάλληλη ένωση του Transmitter και του Receiver σε κανάλι UART για την επίτευξη της επικοινωνίας τους και την μεταφορά δεδομένων. Παρακάτω φαίνεται το dataflow που περιγράφει το module uart το οποίο είναι το top module της υλοποίησης.



Όπως βλέπουμε τα δύο module **uart_transmitter** και **uart_receiver** ενώνονται μέσω της εξόδου **TxD** του πρώτου και της εισόδου RxD του δεύτερου. Κατά αυτόν τον τρόπο το σύμβολο που λαμβάνει ο Transmitter από το testbench στέλνεται ο λαμβάνεται στον Receiver ο οποίος το δειγματοληπτεί και το βγάζει στην έξοδο **Rx_DATA** μαζί με το σήμα **Rx_VALID**. Η συνένωση αυτή σε κώδικα φαίνεται παρακάτω:

```
uart_transmitter uart_transmitter_inst(.reset(reset), .clk(clk), .Tx_DATA(Tx_data),
.baud_select(baud_sel), .Tx_WR(Tx_wr), .Tx_EN(Tx_en), .TxD(Tx_D),
.Tx_BUSY(Tx_busy));
```

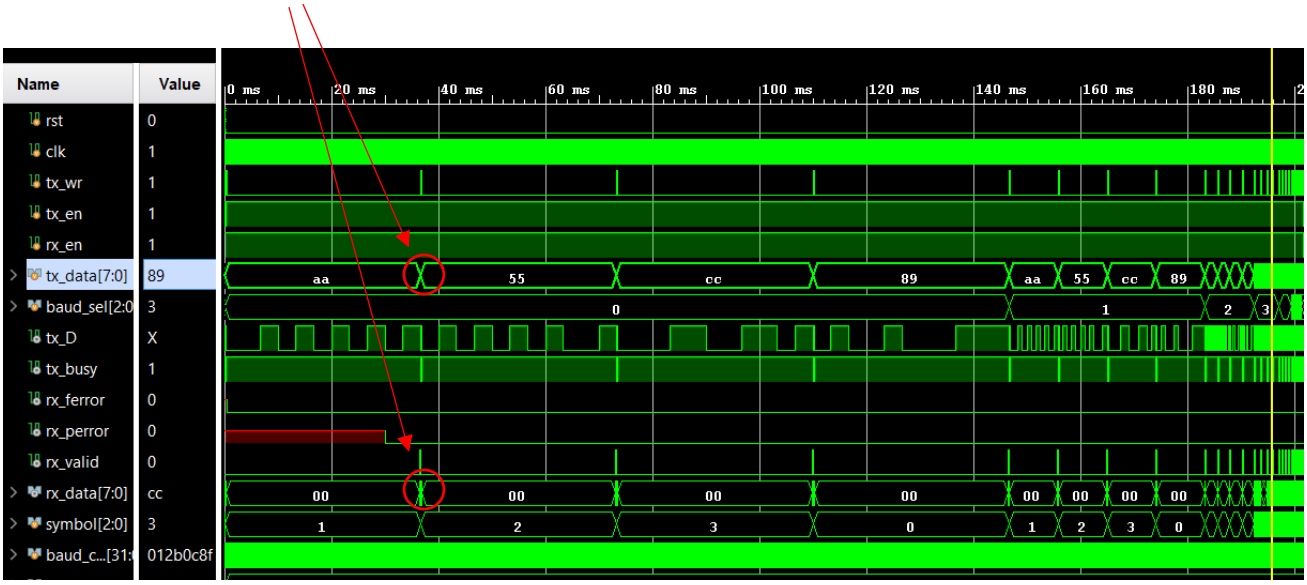
```
uart_receiver uart_receiver_inst(.reset(reset), .clk(clk), .baud_select(baud_sel),
.Rx_EN(rx_en), .RxD(Tx_D),  .Rx_DATA(rx_data), .Rx_FERROR(rx_ferror),
.Rx_PERROR(rx_perror), .Rx_VALID(rx_valid));
```

Για την υλοποίηση του testbench χρησιμοποιήθηκε ένας **baud_counter** που όπως και προηγουμένως εναλλάσσει τα baud rate και ένα always block κατά το οποίο όταν **tx_en** = 1 και **tx_busy** = 0 αλλάζει το σύμβολο προς μεταφορά και πάει στο επόμενο.

```
always@(posedge clk)
begin
    if(tx_en == 1'b1 && tx_busy == 1'b0 )
    begin
        case(symbol)
            s_AA: tx_data <= char_AA;
            s_55: tx_data <= char_55;
            s_CC: tx_data <= char_CC;
            s_89: tx_data <= char_89;
            default: begin
                    tx_data <= 8'b00000000;
                    tx_en <= 1'b0;
                    rx_en <= 1'b0;
                    end
        endcase

        tx_wr <= 1'b1;
        #(`period)tx_wr <= 1'b0;

        if(symbol == s_89 && baud_sel == 3'b111)
        begin
            symbol <= 3'b100;
        end
        else if(symbol == s_89 && baud_sel < 3'b111)
        begin
            symbol <= 3'b000;
        end
        else
        begin
            symbol <= symbol + 3'b001;
        end
    end
    else if(tx_busy == 1'b1)
    begin
        tx_en <= 1'b1;
```

```
    end
end
```

Παρακάτω βλέπουμε ότι την στιγμή έχουν σταλθεί όλα τα bits ενός συμβόλου από το **tx_data** αυτό μεταφέρετε στο **rx_data.**



ΣΥΜΠΕΡΑΣΜΑΤΑ

Συνοψίζοντας θεωρώ πως η δυσκολία της εργασίας έγκειται κυρίως στην υλοποίηση της δειγματοληψίας από τον Transmitter και τον Receiver, και στον συγχρονισμό αυτών των δύο. Επιπρόσθετα, αρκετές δυσκολίες εμφανίστηκαν κατά την υλοποίηση του Testbench για τον Receiver καθώς έπρεπε να μιμείται τον Transmitter και ήταν κάτι ασυνήθιστο για εμάς σε Testbench. Επιπλέον, κατά την εργαστηριακή εξέταση διαπιστώθηκε ότι υπήρχε ένα latch, που παρόλο που δεν επηρέαζε το Simulation και όλα έτρεχαν σωστά, δεν θα έπρεπε να υπάρχει αλλά δυστυχώς δεν διορθώθηκε επιτυχώς. Ωστόσο η υλοποίηση Μονάδας Γενικού Ασύγχρονου Δέκτη Αποστολέα έγινε επιτυχώς με τα δεδομένα να μεταφέρονται από τον Transmitter στον Receiver αναλλοίωτα και ήταν μια αρκετά ενδιαφέρουσα και ιδιαίτερη εργασία.