Laboratorul 1

Comunicarea Seriala UART Implementare cu Automate cu Stări Finite

1.1 Interfețe de Comunicare - Cunoștințe Generale

În electronica embedded un rol esențial îl ocupă interconectarea circuitelor cu diverse sisteme, precum senzori, afișoare sau alte microcontrolere. Odată interconectate, circuitele pot interschimba informații pentru a fi procesate, dar pentru a putea realiza această interschimbare este nevoie de o modalitate de comunicare care să fie înțeleasă de ambele dispozitive care comunică. Modalitățile (sau regulile) de comunicare se mai numesc protocoale și pot fi de trei feluri: seriale, paralele sau hibride.

Protocoalele de comunicare paralele, așa cum le spune numele, permit transmiterea mai multor biți de date simultan. Aceste protocoale, deși au fost esențiale în dezvoltarea timpurie a calculatoarelor și a sistemelor de control industrial, în locuri unde viteza de transfer era crucială și distantele erau scurte, au devenit din ce în ce mai rare în utilizarea modernă, deoarece majoritatea sistemelor au migrat către protocoale seriale îmbunătățite care oferă o serie de avantaje clare în comparație cu protocoalele paralele, precum: utilizarea lățimii de banda mai eficient nefiind nevoie de sincronizări, reducerea interferențelor electromagnetice, distanțe mai mari de transmisie sau costuri și complexitate redusă. Cu toate acestea, câteva exemple de protocoale de comunicare paralelă moderne care sunt încă utilizate datorită vitezei foarte mari pe care o oferă sunt Parallel Peripheral Interface (care poate fi utilizată pentru interconectarea cu LCD-uri, memorii etc), Camera Link (utilizat preponderent în imagistica industrială în sisteme de inspecție industrială) sau Advanced Microcontroller Bus Architecture (AMBA) Advanced High-performance Bus (AHB) dezvoltat de firma ARM pentru interconectarea sistemelor de pe microcontrolere și sisteme pe cip (SoC).

Interfețele de comunicare serială transmit informația bit cu bit, acestea putând fi sincrone, asincrone sau hibride. Într-o comunicație serială sincronă, emițătorul și receptorul folosesc un semnal de ceas comun pentru a sincroniza transmiterea și recepția datelor. Semnalul de ceas indică când fiecare bit, care provine din fluxul continuu de date, trebuie citit sau scris. Exemple de intefețe seriale sincrone de comunicare sunt: I^2C(Inter-Integrated Circuit) sau SPI (Serial Peripheral Interface).

In comunicatiile seriale asincrone, nu există semnal de ceas comun pentru sincronizarea informației. Așadar, sistemele care comunică trebuie să aibă un set de reguli clare (sau un protocol) care să fie respectat. Deși oferă mai multă flexibilitate în ceea ce privește rata de transmitere, în comunicarea asincronă trebuie să fie un mecanism care să indice începutul sau finalul unui cadru de date. Comunicarea UART (Universal Asynchronous Receiver Transmiter) este o astfel de comunicare asincronă și serială.

Interfețele seriale hibride combină caracteristici de la comunicațiile sincrone și de la cele asincrone pentru a oferi flexibilitate, fiabilitate și performanță superioară în aplicații care necesită acest lucrul. Un exemplu de interfață hibrida de comunicare serială o reprezintă Universal Synchronous Asynchronous Receiver Transmiter (USART) care permite atât o comunicare sincronă serială cât și asincronă, permițând dispozitivelor să opereze în moduri diferite adaptându-se la cerințele sistemului.

Nu in ultimul rând, mai există interfețe de comunicare hibride, care combină avantajele interfețelor seriale și paralele. Un exemplu popular de interfață de comunicare hibridă este Peripheral Component Interconnect Express (PCIe), care folosește mai multe linii diferențiale de comunicație, pe fiecare din acestea informația fiind transmisă în mod serial și sincron, pentru a spori lățimea de bandă. Deși PCIe este serial, utilizarea mai multor linii de date simultan pentru

creşterea lăţimii de bandă îi conferă un comportament care poate fi comparat cu transmisia paralelă (deşi pe fiecare linie datele sunt transmise serial).

1.2 Automate cu Stări Finite

Maşinile sau automatele cu stări finite (Finite State Machines – FSM) sunt un model matematic utilizat în descrierea sistemelor care au un număr finit de stări şi care pot tranzita între aceste stări în funcţie de anumite reguli şi de intrări externe. Un FSM este definit dintr-un set de stări, (un sistem putând la un moment dat să se afle în una din aceste stări), un alfabet de intrare (un set de simboluri pe care automatul poate să le primească la intrare), o funcţie de tranziţie între stări care determină schimbarea de la o stare la alta, o stare iniţiala, un set de stări finale şi acţiunile asociate cu fiecare stare. În funcţie de modul de definire al ieşirilor, automatele cu stări finite se clasifică în două tipuri principale: **Automate Mealy** şi **Automate Moore.** Un automat Mealy este un FSM în care ieşirea depinde atât de starea curentă cât şi de simbolul de intrare, iar într-un automat Moore ieşirea depinde numai de starea curentă, indiferent de simbolul de intrare. O diagrama generică, preluata din [1], a unui automat Mealy este prezentată in Figura 1.1.

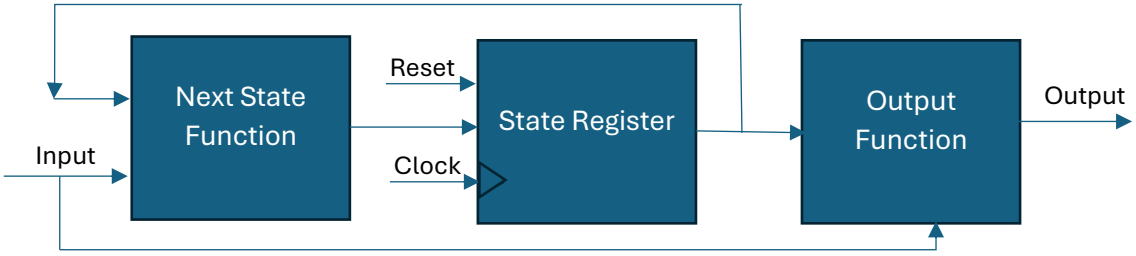

Figura 1.1 Diagrama generica a unui automat Mealy adaptata după modelul din [1]

Există mai multe modalităţi prin care Vivado IDE poate recunoaşte şi sintetiza FSM-urile scrise în VHDL. Astfel, în [1], la paginile 205-210 sunt exemplificate 3 modalităţi de a scrie FSM-urile cu 1, 2 sau 3 procese. De asemenea, tot în această anexă sunt prezentate exemple pentru FSM-uri pentru automate Mealy şi Moore generice, în cazul in care veţi avea nevoie să puteţi să le adaptaţi cerinţelor proiectelor pe care le veţi implementa.

În VHDL, stările vor fi declarate şi folosite folosind tipul enumeraţie, astfel nefiind nevoie de o codificare numerică în momentul în care dorim să le folosim, codificarea fiind realizată în mod implicit. În exemplul de mai jos este descrisă o astfel de declarare a unor stări.

```
type stari_posibile is (stare_1, stare_2, stare_3, stare_4);
signal state : stari_posibile;
```

Vivado poate aplica diferite metode de codificare a stărilor, utilizatorul putând să opteze pentru tehnica dorită folosind **Synthesize Settings – fsm_extraction.** Opţiunile pe care utilizatorul le poate selecta sunt Auto (implicit), One-hot, Gray, Johnson, Sequential. Avantajele şi dezavantajele fiecărei metode se poate găsi in XST User Guide [1].

1.3 Comunicarea Seriala UART

Comunicarea UART este o comunicare serială asincronă. Acest lucru înseamnă că datele sunt transmise sau recepţionate bit cu bit fără a fi nevoie de un semnal de ceas extern care să sincronizeze achiziţia sau transmisia. Cu toate acestea, protocolul UART foloseşte un set de reguli care să-i permită să eschiveze folosirea unui semnal de ceas extern şi care să asigure primirea sau transmiterea robustă şi fără erori a informaţiei. Elementele esenţiale ale protocolului UART sunt: rata de transfer (baud rate), biţii de sincronizare, biţii de date şi biţii de paritate.

Datorită faptului că protocolul de comunicare UART este un protocol foarte comun, majoritatea producătorilor moderni de microcontrolere includ un astfel de protocol de comunicare dar şi librării software dedicate care să permită dezvoltatorilor şi integratorilor de produse să dezvolte foarte rapid produse şi să nu fie nevoiţi să implementeze mereu de la 0 această comunicare serială. Cu toate acestea, pentru a înţelege în profunzime modul de funcţionare al acestui protocol, cât şi modelarea folosind automate cu stări finite, în acest laborator vom implementa protocolul UART.

1.3.1 Semnificaţia Elementelor Mecanismului UART

În această secţiune vom lua fiecare element din protocolul UART şi vom explica semnificaţia acestuia. Ambele sisteme care comunică trebuie să respecte aceeaşi parametri ai comunicării pentru ca aceasta să se realizeze cu succes.

Baud Rate

Acest parametru al transmisiei ne informează cu privire la rata de transfer a datelor, sau viteza transmisiei pe linia serială. Această mărime este exprimată în stări pe secundă, unde o stare poate fi un bit (o valoare de 0 sau 1), dar pot exista interfeţe în care o stare poate avea şi alte valori, nu doar 0 sau 1. Valorile cele mai folosite ale baud rate-ului sunt 1200, 2400, 4800, 19200, 38400, 57600, sau 115200.

Biţii de sincronizare

Biţii de sincronizare sunt necesari întrucât nu avem un semnal de ceas care să ne indice începutul sau finalul unui cadru de date. Aceşti biţi de sincronizare sunt biţii de start şi de stop. Aceştia, aşa cum le indică şi numele, marchează începutul cadrului de date şi finalul acestuia. Bitul de start este, de regulă, un singur bit, iar pentru stop putem avea unul sau doi biţi.

Biţii de date

Pachetul de date îl reprezintă datele concrete pe care dorim să le transmitem/primim. Informaţia din acest pachet de date poate fi între 5 şi 9 biţi, dar, de regulă, se folosesc 8 biţi de date. Ambele dispozitive care comunică vor trebui să fie de acord cu endiannesul, adică care bit va fi transferat primul, cel mai semnificativ (MSB), sau cel mai puţin semnificativ bit (LSB).

Biţii de paritate

Biţii de paritate reprezintă o modalitate elementară de a verifica dacă există erori în transmisie. Opţiunile disponibile pentru biţii de paritate sunt paritate „impară" sau „para". Bitul de paritate se obţine aplicând operatorul XOR (sau exclusiv) pe toţi biţii din cadrul de date cum este exemplificat in formula (1). Aplicarea operatorului XOR este exemplificat de simbolul $\oplus$, iar biţii de date sunt reprezentaţi de secvenţa C_{n-1} pana la C_0. Setarea bitului de paritate este o modalitate opţională de verificare a corectitudinii informaţiilor transmise, care se foloseşte, de regula, când se comunică pe distanţe lungi sau în medii cu zgomot.

$$Paritate_{para} = C_{n-1} \oplus C_{n-2} \oplus \dots \oplus C_0 \oplus 0$$

$$Paritate_{impara} = C_{n-1} \oplus C_{n-2} \oplus \dots \oplus C_0 \oplus 1 \quad (1)$$

Conexiunea între două sisteme care folosesc interfaţa UART se realizează ca în Figura 1.2, şi anume: pinul RX(recepţie) de la un sistem se va conecta la pinul TX (transmisie) de la celălalt şi invers.

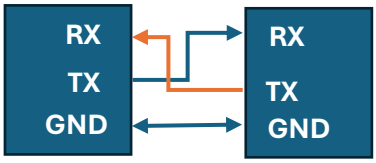

Figura 1.2. Exemplu de conexiune UART între două componente

1.3.2 Implementarea FSM-ului de transmisie

În aceasta secţiune şi în următoarea se vor prezenta diagramele automatelor cu stări finite necesare implementării transmisiei şi recepţiei UART. Este important de menţionat că diagramele corespund unei transmisii cu următoarele caracteristici: 1 bit de start, 1 bit de stop, 8 biţi de date, baud rate de 9600 şi fără biţi de paritate.

Diagrama FSM-ului pentru transmisia UART pe 8 biţi pentru componenta de transmisie este ilustrată în Figura 1.3, preluată şi adaptata din [2]. În această diagramă, fiecare stare este exemplificată cu un cerc, în interiorul căruia avem numele stării, arcele reprezintă tranziţii de la o stare la alta, iar textul de pe arce semnifică condiţia care trebuie îndeplinită astfel încât tranziţia între stări să se realizeze. În dreptul stărilor se află şi variabilele de stare care sunt setate în interiorul stării şi care îşi păstrează valoarea atâta timp cât sistemul se află în acea stare.

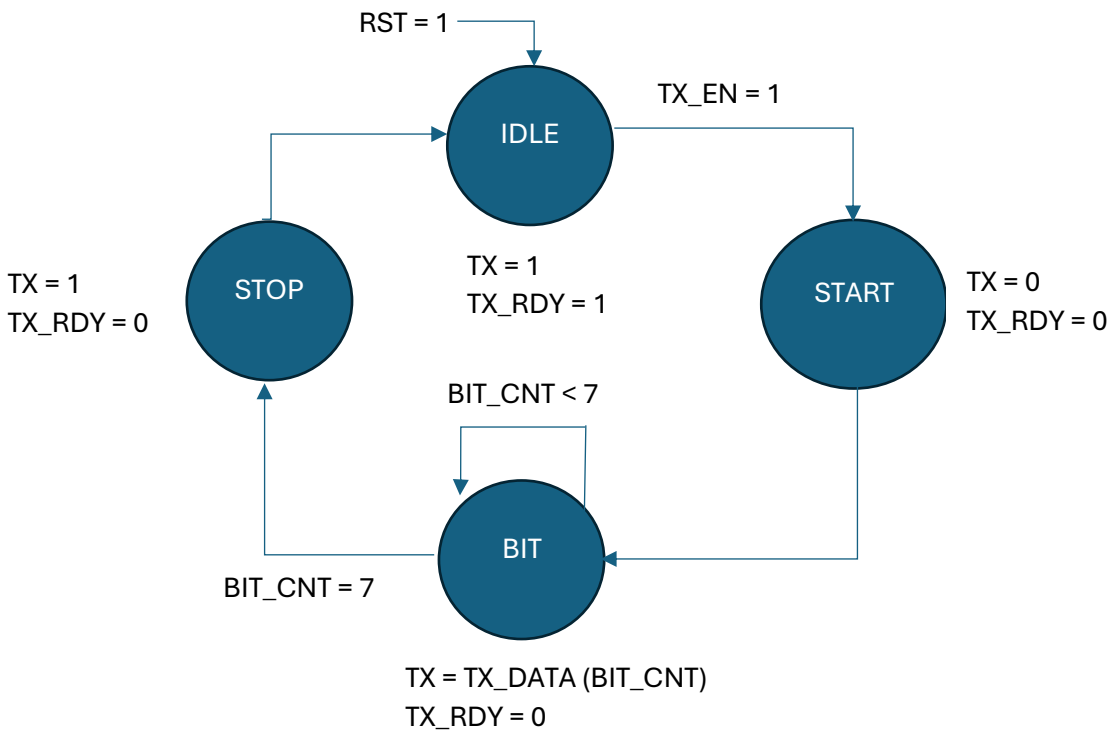

Figura 1.3. Diagrama de tranziţii pentru FSM-ul care ţine de partea de transmisie UART [2]

Tranziţiile intre stări au loc pe frontul crescător de ceas, când BAUD_ENABLE este '1'. Astfel, se garantează că fiecare bit rămâne pe linia de transmisie pentru durata stabilită de baud rate. Semnalul BIT_CNT funcţionează ca un contor în cadrul FSM-ului, indicând poziţia bitului curent care urmează să fie transmis din TX_DATA. Acesta trebuie incrementat în starea BIT şi resetat după fiecare caracter transmis (poate fi resetat în starea IDLE sau în toate stările, cu excepţia stării BIT).

Procesul de funcţionare al automatului de transmisie este următorul:

- **Iniţializare:** Când semnalul de reset (RST = 1) este activ, automatul este în starea IDLE, indicând că linia de transmisie (TX) este 1 (inactiv) şi că transmisia poate începe oricând (TX_RDY = 1).
- **Începerea transmisiei:** Când semnalul TX_EN devine 1, automatul trece din starea IDLE în starea START. În această stare, linia TX este setată la 0, semnalizând începutul transmisiei, adică TX = 0.
- **Transmiterea bit cu bit:** Fără nici o condiţie prealabila, FSM-ul de transmitere trece la starea bit, unde transmite fiecare bit al datelor în funcţie de contorul de biţi (BIT_CNT). Se transmit biţi până când contorul ajunge la valoarea 7, indicând că toţi cei 8 biţi au fost transmişi.

- **Finalizarea transmisiei:** Când BIT_CNT ajunge la 7, automatul trece în starea STOP, unde linia TX este din nou setată la 1 (indicând sfârşitul transmisiei). După aceea, automatul revine la starea IDLE, aşteptând o nouă comandă de transmisie.

Diagrama bloc pentru componenta de transmisie UART realizată prin automate de stări finite, TX_FSM este prezentata în Figura 1.4 [2].

Figura 1.4 Schema bloc a modulului de transmisie UART, unde sunt ilustrate intrările şi ieşirile componentei [2]

1.3.3 Implementarea FSM-ului de recepţie

Într-o comunicare seriala asincronă, precum UART, emiţătorul şi receptorul nu folosesc un semnal de ceas pentru sincronizare, ci se folosesc de o rată de eşantionare, numita baud rate, generată de fiecare participant al comunicării în mod independent. În componenta de recepţie, receptorul trebuie să citească (eşantionat) semnalul bit cu bit şi să extragă valoarea trimisă de sursă. Prin nesincronizarea perfectă a baud rate-urilor (care sunt generate independent), pot apărea decalaje care pot duce la citirea incorectă a informaţiei, din cauza omiterii unor biţi, citirea de mai multe ori ale aceluiaşi bit sau ratarea bitului de start.

Pentru a rezolva această problemă, semnalul este supra eşantionat (citit la o rata mai mare) de pe linia seriala decât rata cu care informaţia a fost transmisă. Riscul de decalaj este eliminat prin citirea în jurul aproximativ a mijlocului fiecărui bit, ceea ce asigură resincronizarea (Figura 1.5). Pentru a realiza supra eşantionarea, datele de pe linia seriala sunt citite la o rata de 16 ori mai mare decât rata de transmisie si informaţia păstrată este doar cea de la mijlocul intervalului de citire.

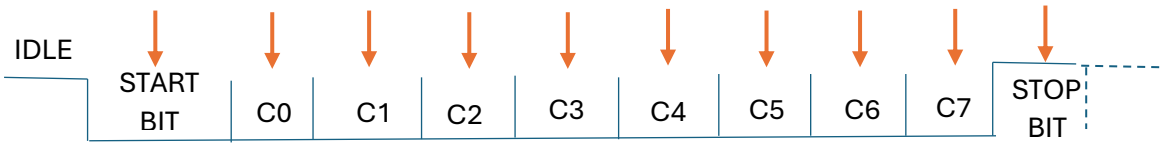

Figura 1.5. Săgeţile indică momentul în care elementul din mijlocului fiecărui bit ar trebui citit de receptor în mod ideal, când se face supra eşantionarea

Similar ca şi in cazul transmisiei, şi pentru recepţie trebuie definit baud rate-ul de 9600 biţi pe secundă, dar diferenţa în acest caz este că rata de eşantionare trebuie să fie de 16 ori mai mare, ţinând cont de nevoia de supra eşantionare. Ceilalţi parametri, precum bitul de paritate, biţii de stop sau biţii de date rămân la fel ca şi in cazul transmisiei (fără paritate, 1 bit de stop si 8 biţi de date).

In Figura 1.6, adaptată din [2], se prezintă diagrama de tranziţie a FSM-ului de recepţie, cu cele 5 stări relevante pentru acest automat, condiţiile de tranziţie dintre stări şi variabilele de stare relevante, împreuna cu valorile acestora.

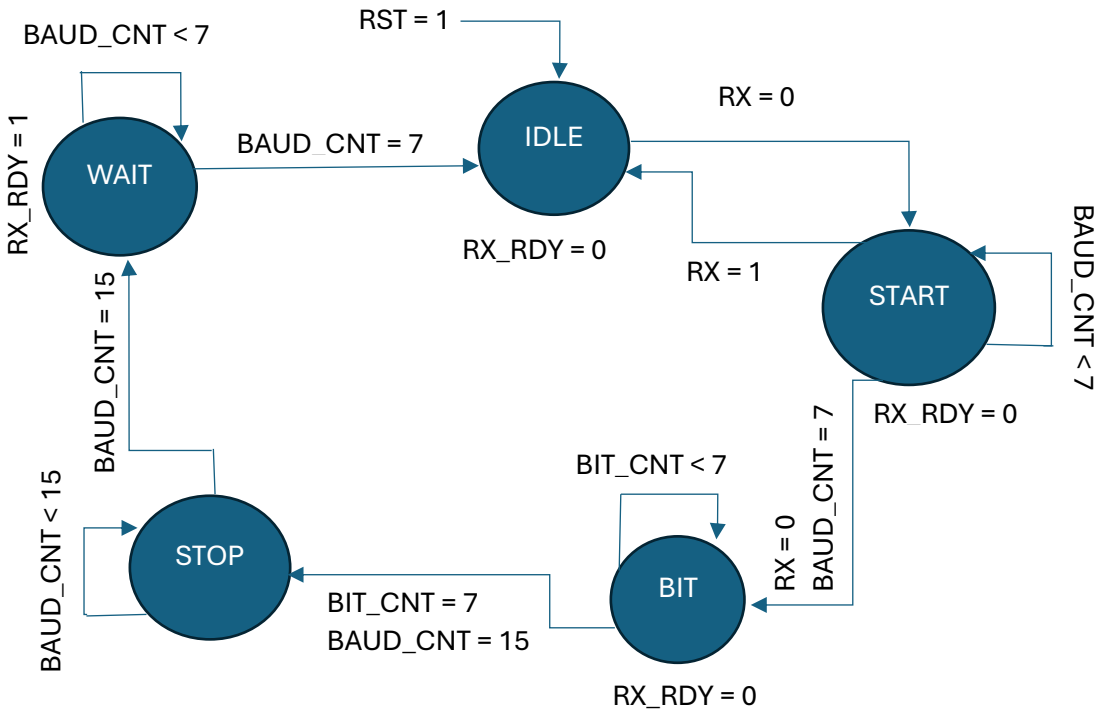

Figura 1.6. Diagrama de tranziții a FSM-ului care tine de partea de recepție UART [2]

Procesul de funcţionare al automatului de recepţie în funcţie de fiecare stare este următorul:

- **IDLE** (transmisia e inactivă)

Contorul BAUD_CNT este resetat la 0 şi semnalul RX_RDY este setat şi el pe 0, indicând că recepţia poate începe. Automatul rămâne in starea IDLE până când pe linia seriala apare 0 logic (RX=0) care indică bitul de start al unui nou pachet de date. În acel moment FSM-ul trece in starea START.

- **START**

Se incrementează treptat BAUD_CNT si FSM-ul eşantionează linia seriala la fiecare incrementare. În momentul când se ajunge la valoarea BAUD_CNT = 7, FSM-ul detectează mijlocul bitului de start, care ar trebui să fie RX=0. În cazul în care această condiţie este îndeplinită se trece la următoarea stare BIT.

- **BIT**

În această stare se citesc biţii de date. BAUD_CNT este incrementat pentru a ajunge la mijlocul fiecărui bit de date. Când BAUD_CNT ajunge la 15 (pentru ca tot timpul se citesc si cele 7 eşantioane ale bitului precedent) se salvează valoarea bitului curent, RX_DATA[BIT_CNT] = RX, se resetează BAUD_CNT şi se incrementează BIT_CNT pentru a trece la următorul bit din RX_DATA. În momentul în care BAUD_CNT ajunge la 15 şi BIT_CNT ajunge la 7, toţi cei 8 biţi de date au fost citiţi, iar automatul trece în starea STOP.

- **STOP**

În această stare FSM-ul aşteaptă să treacă un interval de 15 valori care acoperă ultima jumătate de eşantioane de la ultimul bit de date şi prima jumătate de eşantioane de la bitul de stop pentru a putea prelua valoarea bitului de stop. Când BAUD_COUNT ajunge la valoarea 15, automatul cu stări finite va tranzita în starea WAIT.

- **WAIT**

În această stare se anunţă că s-a recepţionat un caracter (RX_RDY = 1) şi se mai aşteaptă o jumătate de interval de bit pentru a se finaliza eşantionarea bitului de STOP. După aceasta aşteptare, FSM-ul revine la starea IDLE şi este pregătit pentru a recepţiona următorul caracter.

Diagrama bloc pentru componenta de recepție UART care ilustrează intrările și ieșirile ale acestui modul, realizată prin automatul RX_FSM este prezentată in Figura 1.7 [2].

Figura 1.7. Schema bloc a automatului RX_FSM pentru recepția UART [2]

1.4. Considerații Practice

Modulul UART de recepție și transmisie va fi implementat in limbajul VHDL pe placa Zybo. Întrucât această placă nu are un cip FTDI care să fie conectat la un port USB ce e accesibil din FPGA, pentru a realiza comunicarea serială cu computerul vom folosi un modul PmodUSBUART [3] ca cel din Figura 1.8.

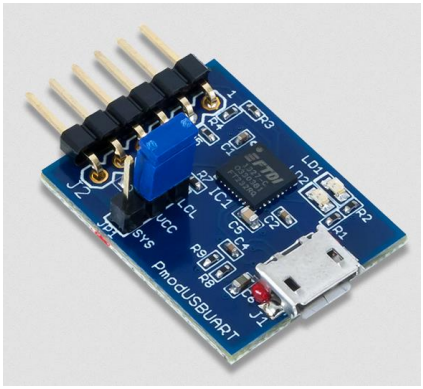

Figura 1.8. Modul cu FTDI folosite pentru comunicarea seriala preluata din [3]

Pinii acestui modul si semnificația acestora este prezentata in Tabelul I.

Tabelul I. Pinii modulului PMod USBUART

Conectorul J2		
Nr Pin	**Semnal**	**Descriere**
1	**RTS**	**Ready To Send**. Acest pin este folosit pentru a semnala că modulul este pregătit să trimită date către dispozitivul conectat prin interfața UART.
2	**RXD**	**Receive Data**. Pinul RXD este utilizat pentru a primi date din partea dispozitivului conectat prin interfața UART.
3	**TXD**	**Transmit Data**. Acest pin este responsabil pentru transmiterea datelor de la modulul Pmod USBUART către dispozitivul conectat.
4	**CTS**	**Clear To Send**. Pinul CTS este utilizat pentru a controla fluxul de date, indicând modulului USBUART că poate trimite date.
5	**GND**	**Ground**. Acest pin asigură împământarea comună între modul și dispozitivul gazdă.
6	**SYS3V3**	**Power Supply 3.3V**. Acesta este pinul de alimentare de 3.3V, utilizat pentru a alimenta modulul sau pentru a alimenta dispozitivul gazdă, în funcție de configurația jumperului.

Este important de menţionat faptul ca nu toţi pinii acestui modul trebuie folosiţi pentru a realiza o comunicare seriala. Spre exemplu, in cazul nostru vom folosi doar RXD, TXD, GND si SYS3V3 ceilalţi pini putând fi lăsaţi in aer.

Jumperul JP1 are două poziţii:

- **LCL3V3**: Dispozitivul ataşat este alimentat independent de modulul PMod USBUART.
- **SYS3V3**: Dispozitivul ataşat este alimentat prin intermediul cipului FTDI de pe modul.

În cazul nostru jumperul va fi lăsat pe LCL3V3.

Pentru a utiliza modulul de comunicare UART acesta trebuie conectat la placa Zybo ca în figura de mai jos, la portul JC, pe bareta de pini mamă superioară, ca şi în Figura 1.9.

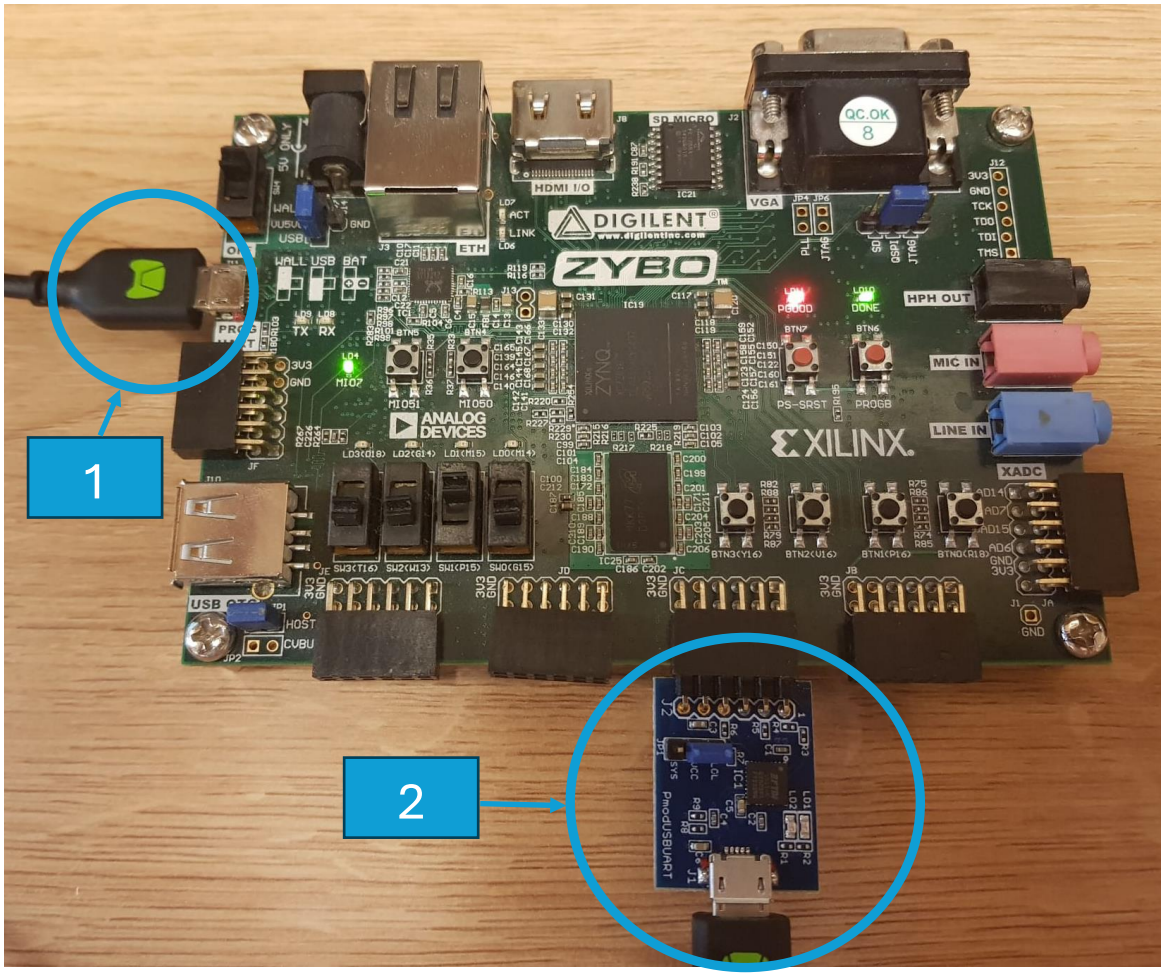

Figura 1.9. Conectarea modulului PMod USBUART la placa Zybo. În micro USB-ul prezentat cu 1 se introduce cablul de alimentare, care se conectează apoi la PC, iar cu 2 este reprezentat modulul PMod utilizat în comunicarea serială.

Pentru a putea realiza comunicarea serială între placa Zybo şi calculator este nevoie un program dedicat de comunicare. Vă recomandăm să utilizaţi HTerm [4] sau Tera Term [5]. Pentru a putea vedea pe ce ComPort comunica FTDI-ul, este recomandată următoarea procedura:

1. Scoateţi cablul USB conectat la FTDI din calculator.
2. Deschideţi Device Manager in Windows si navigaţi la secţiunea Ports (COM & LTPT) unde veţi observa porturile seriale care sunt momentan în folosinţă.

3. Având Device Manager în față, introduceți cablul USB care e conectat de PMod-ul USBUART într-un port USB, veți auzi un sunet de plug and play din partea sistemului de operare și observați cum apare un nou ComPort. Acesta este portul pe care se va face comunicarea seriala. Dacă în momentul în care ați introdus cablul USB nu apare nici o modificare, dați click dreapta pe secțiunea Ports (COM & LTPT) si apăsați „Scan for hardware changes”

In Figura 1.10 veți observa cum arată secțiunea de Porturi înainte și după conectare PMod-ului USBUART.

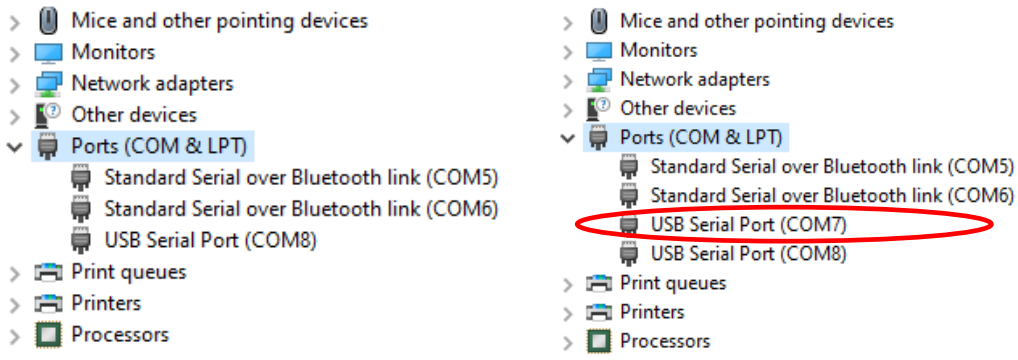

Figura 1.10. Vizualizarea noului ComPort creat odată cu introducerea în computer a PModului USBUART. În figura din stânga se observă imaginea din Device Manager înainte de introducerea cablului în calculator, iar în imaginea din dreapta se observă diferența care apare odată cu introducerea cablului in USB.

Deschideți IDE-ul Vivado 2024.1 si creați un proiect nou, numiți-l ZYBO_UART, și urmați aceeași procedură prezentata în îndrumătorul de laborator de la Arhitectura Calculatoarelor [2]. **Singura diferență va apărea în momentul în care alegeți placa pentru care se realizează proiectul**. Când ajungeți la configurarea plăcuței, selectați tabul **Boards** din partea stângă a ferestrei și în bara de căutare scrieți Zybo pentru a arăta toate versiunile de placi Zybo existente. Selectați plăcuța pe care o aveți (spre exemplu Zybo sau Zybo Z7-20) și apăsați pe butonul **Next**.

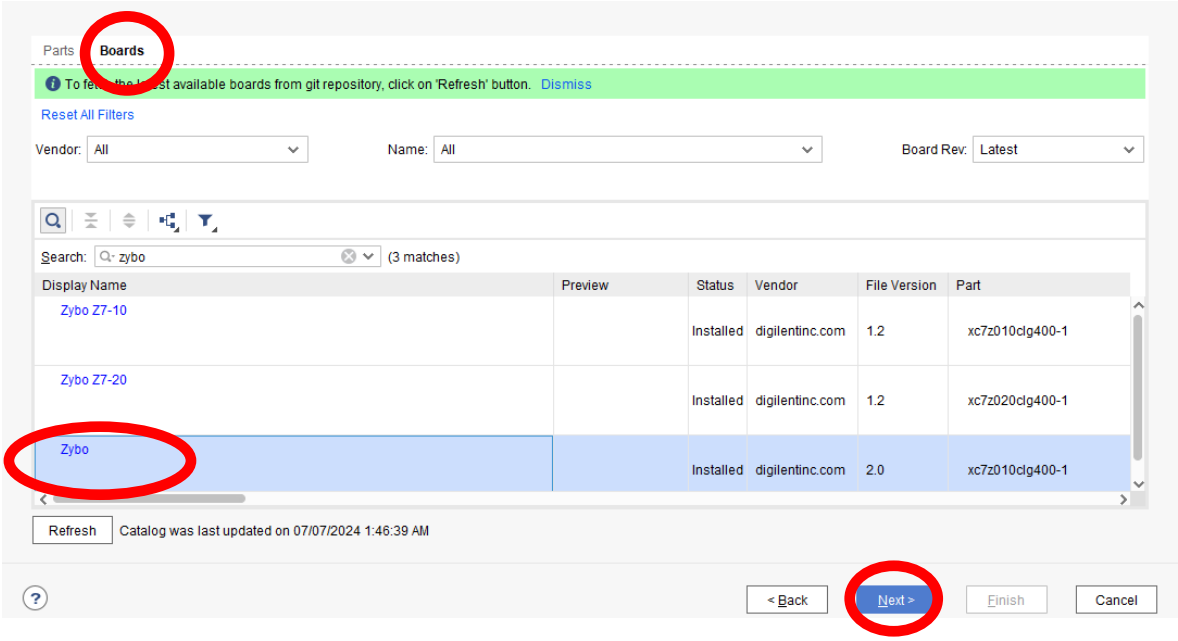

Figura 1.11. Selectarea plăcuței pe care se va implementa modulul UART

Alternativ din tabul Parts puteţi selecta configuraţia specifică plăcii pe care o folosiţi. Spre exemplu pentru **Zybo 7000** configuraţia este următoarea: Family Zynq-7000, Package clg400, Speed Grade -1, Default Part xc7z010clg400-1. Dacă folosiţi placa **Zybo 7020** configuraţia este următoarea: Family Zynq-7000, Package clg400, Speed Grade -1, Default Part xc7z02clg400-1 s.a. Apăsaţi butonul **Finish** pentru ca Vivado să vă genereze proiectul.

Pentru rezolvarea recepţiei şi transmisiei UART vom folosi un cod şablon (boilerplate code). Acest cod conţine 3 module. Un modul principal **top.vhd**, modulul de recepţie **receivefsm.vhd** si transmisie **transmitfsm.vhd**. Aceste fişiere vor trebui adăugate în proiectul creat în Vivado. Puteţi face acest lucru fie creând trei noi surse şi copiind conţinutul acestor fişiere sau să le adăugaţi prin adăugarea fişierelor la proiect.

Vom parcurge un exemplu cum putem adăuga conţinutul într-un fişier nou creat. În Vivado daţi click dreapta pe directorul creat automat numit **Design Sources** şi daţi click pe opţiunea **Add Sources.** Alternativ din **Flow Navigator** din meniul **Project Manager**, sub meniul **Settings** apăsaţi pe opţiunea **Add Sources**. In Figura 1.12 se pot observa aceste variate de a adăuga surse noi.

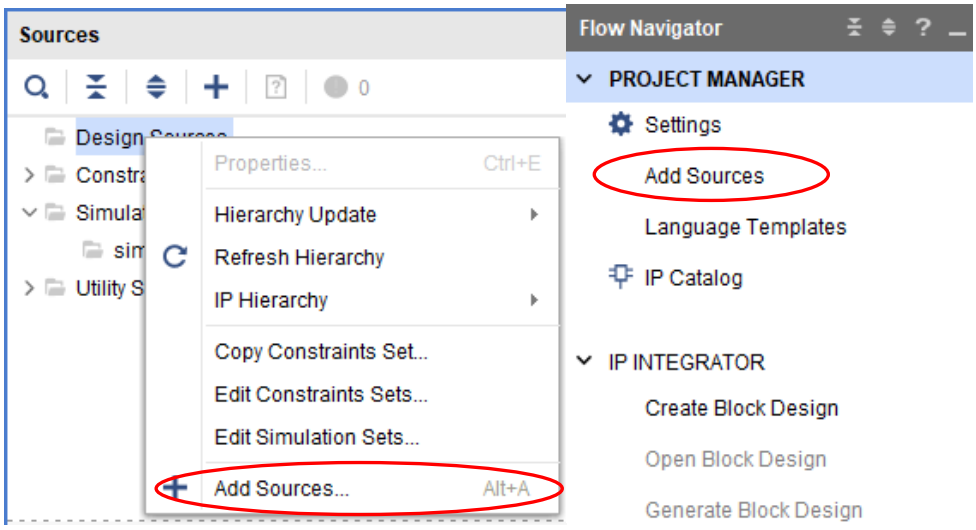

Figura 1.12 Adăugarea unei noi surse.

În fereastra care apare selectaţi opţiunea **Add or create design sources** si apăsaţi butonul Next. În funcţie de metoda pe care doriţi să o folosiţi din cele menţionate, apăsaţi pe **Add files** sau **Create File**. În acest exemplu vom crea un fişier nou şi vom copia conţinutul fişierului de transmisie în acel fişier nou creat. Aşadar, apăsaţi butonul **Create File** şi selectaţi opţiunea VHDL din meniul File type. Numiţi noul fişier **transmitfsm** şi apăsaţi ok. Apăsaţi apoi butonul **Finish**, iar în fereastra care apare apăsaţi **Ok**, fără a completa nimic în secţiunea I/O definitions. Când apare fereastra cu mesajul „The module definition has not changed. Are you sure you want to use these values?", apăsaţi butonul **Yes**. Noul fişier creat va apărea în directorul Design Sources. Copiaţi conţinutul fişierului transmitfsm.vhd şi plasaţi-l în noul fişier creat. Repetaţi procedeul şi pentru celelalte 2 fişiere (top.vhd si receivefsm.vhd). Alternativ, dacă importaţi sursele, după ce apăsaţi pe butonul **Add Sources**, apăsaţi pe **Add Files**, localizaţi fişierul dorit pe disc si apăsaţi apoi pe butonul Ok.

În fiecare din fişierele menţionate mai sus lipsesc anumite procese pentru a face codul funcţional. Mai jos este explicat codul din fiecare fişier şi sunt descrise procesele care vor trebui implementate. Înainte de orice proces sau secvenţă de cod care va trebui implementată sunt scrise cuvintele **TO DO** pentru a identifica mai uşor unde trebuie făcute modificările. Fişierele necesare realizării acestui laborator mai pot fi descărcate de la linkul [6] din bibliografie.

1.4.1 Modulul de transmisie

Modulul de transmisie se bazează pe explicaţiile teoretice din secţiunea 1.3.2. Codul şablon se găseşte mai jos.

```
--Includerea librariilor necesare
library IEEE;
--ofera definitii pentru tipuri de date precum STD_LOGIC și STD_LOGIC_VECTOR
use IEEE.STD_LOGIC_1164.ALL;
--permite operatii aritmetice pe tipul std_logic
use IEEE.STD_LOGIC_UNSIGNED.ALL;
--entitatea pentru modulul de transmisie
entity transmitfsm is
   Port (
 --Semnalul de ceas (clock), folosit pentru sincronizarea FSM-ului.
        clk: in STD_LOGIC;
--Semnalul de resetare, care inițializează FSM-ul într-o stare de start.
        rst : in STD_LOGIC;
--Semnalul care controlează rata baud (viteza de transmisie).
        baud_en : in STD_LOGIC;
--Semnalul care permite inițierea transmisiei.
        tx_en : in STD_LOGIC;
--Datele de transmis, pe 8 biți (STD_LOGIC_VECTOR (7 downto 0)).
        tx_data : in STD_LOGIC_VECTOR (7 downto 0);
--Ieșirea serială, ce transmite datele bit cu bit.
        tx : out STD_LOGIC;
--Ieșire care indică dacă transmisia este gata să înceapă.
        tx_rdy : out STD_LOGIC);
end transmitfsm;

architecture Behavioral of transmitfsm is
--Tip de date pentru stările FSM-ului (idle, start, bits, stop).
type state_type is (idle, start, bits, stop);
--Semnal pentru a stoca starea curentă a FSM-ului, inițializată la idle.
signal state : state_type := idle;
--Contor de biți, folosit pentru a număra biții de date transmiși (de la 0 la 7).
signal bit_cnt : STD_LOGIC_VECTOR(2 downto 0);
begin
-- FSM state transition
--Aici ar trebui implementată logica de tranziție a stărilor FSM, pe baza intrărilor clk și rst.
--Acesta este locul în care FSM-ul va trece între stările idle, start, bits și stop în funcție de
--condițiile și semnalele de intrare
   process (clk, rst)
   begin
  --TO DO
  -- De implementat tranzitia de stari a FSM-ului pe baza Figurii  1.3 de la sectiunea 1.3.2
   end process;
```

```
 -- FSM outputs
--Descrie cum se generează ieșirile (tx și tx_rdy) în funcție de starea curentă a FSM-ului
  process(state, tx_data, bit_cnt)
  begin
    case state is
                when idle => tx <= '1'; tx_rdy <= '1';
                when start => tx <= '0'; tx_rdy <= '0';
                when bits => tx <= tx_data(conv_integer(bit_cnt));
                        tx_rdy <= '0';
                when stop => tx <= '1'; tx_rdy <= '0';
                when others => tx <= 'X'; tx_rdy <= 'X';
     end case;
  end process;
end Behavioral;
```

1.4.2 Modulul de receptie

Codul şablon este bazat pe explicaţiile din secţiunea 1.3.3 si Figura 1.6 care descrie tranziţia de stări din modulul de recepţie implementat in componenta receivefsm.

```
--includerea librariilor necesare rularii proiectului
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
--declararea entitatii de primire de date
entity receivefsm is
  Port (
-- Semnal de intrare pentru ceas (clock), folosit pentru sincronizarea FSM-ului
       clk : in STD_LOGIC;
-- Semnal de resetare. Când este activ ('1'), FSM-ul revine la starea inițială (idle).
       rst : in STD_LOGIC;
-- Semnal de activare a baudrate-ului. Când este activ ('1'), FSM progresează la următoarea
stare
       baud_en : in STD_LOGIC;
-- Semnalul de intrare care primește datele seriale.
       rx : in STD_LOGIC;
-- Datele recepționate sunt stocate în acest vector de 8 biți.
       rx_data : out STD_LOGIC_VECTOR (7 downto 0);
-- Semnal care indică atunci când datele sunt disponibile ('1'), altfel este '0'.
       rx_rdy : out STD_LOGIC);
end receivefsm;
--arhitectura modulului
architecture Behavioral of receivefsm is
--Definirea tipului state_type, care reprezintă toate stările posibile ale FSM-ului
--Atentie NU e typo ca wait e cu 2 de ‘t’
type state_type is (idle, start, bits, waitt, stop);
--Semnal care ține evidența stării curente a FSM-ului, inițializat cu starea idle.
signal state : state_type := idle;
```

```
-- Un contor de 4 biți pentru a număra ciclurile ceasului în funcție de baudrate.
signal baud_cnt : STD_LOGIC_VECTOR (3 downto 0);
-- Un contor de 3 biți pentru a număra biții dintr-un octet de date.
signal bit_cnt : STD_LOGIC_VECTOR (2 downto 0);
begin
   -- Tranzitia de stari a FSM-ului
   process(clk, rst)
   begin
--Dacă semnalul de resetare rst este activ ('1'), FSM revine la starea idle.
      if rst = '1' then
         state <= idle;
--Dacă nu este resetare, atunci la fiecare front crescător al ceasului (clk),
--FSM-ul verifică dacă semnalul baud_en este activ ('1').
      elsif rising_edge(clk) then
--Dacă baud_en este activ, FSM-ul își poate schimba starea conform logicii din
case state is.
         if baud_en = '1' then
            case state is
--TO DO
--de implementat tranzitia de stari pentru FSM-ul de receptie pe baza Figurii 1.6 din
sectiunea 1.3.3
            end case;
         end if;
      end if;
   end process;

   -- Iesirile FSM-ului
   process (state)
   begin
-- Logica pentru setarea semnalului rx_rdy în funcție de starea curentă
      case state is
--  idle, start, bits, stop: Setează rx_rdy la '0', indicând că nu sunt date disponibile.
         when idle => rx_rdy <= '0';
         when start => rx_rdy <='0';
         when bits => rx_rdy <='0';
         when stop => rx_rdy <='0';
--waitt: Setează rx_rdy la '1', indicând că datele sunt disponibile și pregătite să fie citite.
         when waitt => rx_rdy <='1';
--others: Acoperă alte stări neanticipate, setând rx_rdy la un semnal nedeterminat ('X'),
--ceea ce indică o situație de eroare sau o stare nevalidă
         when others => rx_rdy <='X';
      end case;
--finalizarea procesului cu iesirile FSM-ului
   end process;

end Behavioral;
```

1.4.3 Modulul principal

În modulul principal top.vhd vom mapa componentele descrise anterior şi vom implementa un anumit scenariu de utilizare. Dacă switchul numărul 1 este ‚1' atunci se va afişa un mesaj constant format din 4 caractere. Dacă switchul are valoare logica ‚0' atunci se va afişa ultima valoare recepţionată formată tot din 4 caractere. Transmisia datelor este iniţializată atunci când apăsam butonul 0. Pentru a elimina efectul de bouncing vom folosi un mono-pulse generator(MPG). Deşi codul pentru MPG este oferit, în cazul in care doriţi clarificări în legătura cu aceasta componenta şi modul ei de funcţionare puteţi consulta laboratorul 1 de Arhitectura Calculatoarelor din [2]. Similar cu modulele de recepţie şi transmisie, codul pentru modulul principal este oferit mai jos.

```
--includerea librariilor necesare rularii proiectului
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
--creerea entitatii top si includerea elementelor hardware de pe placa
--aceste elemente trebuie sa fie definite in fisierul de constrangeri XDC cu acelasi nume
entity top is
 Port (
     clk : in STD_LOGIC; -- semnalul de ceas provenit de la placa
     led : out std_logic_vector(3 downto 0); --cele 4 leduri ale placii
     sw : in std_logic_vector(3 downto 0); --cele 4 switchuri ale placii
     btn : in std_logic_vector(3 downto 0);--butoanele placii
     rx : in STD_LOGIC;--portul de rx necesar comunicarii UART
     tx : out STD_LOGIC; --portul de tx necesar comunicarii UART
     );
end top;

--arhitectura componentei top
architecture Behavioral of top is
--maparea componentei mono-pulse generator
component MPG is
   Port ( en : out STD_LOGIC;
        input : in STD_LOGIC;
        clock : in STD_LOGIC);
end component;

--maparea componentei de transmitere
component transmitfsm
   Port ( clk: in STD_LOGIC;
        rst : in STD_LOGIC;
        baud_en : in STD_LOGIC;
        tx_en : in STD_LOGIC;
        tx_data : in STD_LOGIC_VECTOR (7 downto 0);
        tx : out STD_LOGIC;
        tx_rdy : out STD_LOGIC);
end component;
```

```
--Semnalele baud_en și baud_en_x16 controlează sincronizarea transmiterii și recepției la un
--anumit baud rate. baud_en este pentru transmisie, iar baud_en_x16 este valoarea
--esantionata la o rata mai mare care este utilizata in recepție pentru a creste acuratetea.
signal baud_en, baud_en_x16 : std_logic;
--Controlează inițierea și procesul de transmisie serială. tx_rdy semnalizează când datele sunt
gata de transmisie, iar tx_start inițiază transmisia.
signal tx_start, tx_en, tx_rdy, tx_rdy1: std_logic;
--Controlează procesul de recepție. rx_rdy indică momentul în care datele au fost
recepționate, iar rx_rdy1 este o versiune întârziată cu un ciclu a acestuia pentru sincronizare
signal rx_rdy, rx_rdy1 : STD_LOGIC;
--Semnalele cnt și cnt_x16 sunt contoare pentru a număra ciclurile de ceas, utilizate in
--generarea semnalelor de baud rate pentru transmisie si receptie (baud_en și baud_en_x16).
signal cnt : STD_LOGIC_VECTOR(13 downto 0) := (others => '0');
signal cnt_x16 : STD_LOGIC_VECTOR(9 downto 0) := (others => '0');
--Semnalele tx_reg și rx_reg sunt registre de 24 de biți pentru stocarea datelor transmise și
--recepționate
signal tx_reg, rx_reg : STD_LOGIC_VECTOR(23 downto 0);
--Semnalele tx_digit și rx_digit stochează câte 6 biți din datele transmise sau recepționate,
--corespunzând unui digit.
signal tx_digit, rx_digit : STD_LOGIC_VECTOR(5 downto 0);
--Semnalele tx_data și rx_data stochează datele de transmisie și recepție în format ASCII,
fiecare de 8 biți
signal tx_data : STD_LOGIC_VECTOR(7 downto 0);
signal rx_data : STD_LOGIC_VECTOR(7 downto 0) := (others => '0');
--Semnalele tx_digit_cnt și rx_digit_cnt sunt contoare pentru a ține evidența digit-urilor
transmise sau recepționate (câte 2 biți, deoarece sunt 4 digiti în total).
signal tx_digit_cnt : STD_LOGIC_VECTOR(1 downto 0);
signal rx_digit_cnt : STD_LOGIC_VECTOR(1 downto 0) := (others => '0');
--Semnalul en activează inițierea transmisiei si este generat de MPG
signal en: STD_LOGIC;
--Semnalul transmitionData stochează datele care urmează a fi transmise sau recepționate.
signal transmitionData : std_logic_vector(23 downto 0);

begin

--maparea mono pulse generatorului la butonul 0 pentru a elimina fenomenul de debuncing
--iesirea en este folosita pentru a initializa transmisia
    monopulse1: MPG port map(en, btn(0), clk);

-- *************** UART Tx ***************
--initiaza procedura de transmitere de date cand butonul este apasat
   process(clk)
   begin
      if rising_edge(clk) then
         tx_start <= en;
      end if;
   end process;
```

```
--Acest proces generează semnalul baud_en pentru un ceas de 125 MHz. Contorul cnt se
-- incrementează la fiecare ciclu de ceas și resetează atunci când atinge valoarea #Numar,
--care determină frecvența semnalului de baud rate.
-- baud_en este setat la '1' o dată la #Numar de cicluri de ceas.
process(clk)
 begin
--TO DO de calculat factorul de divizare (#Numar) pentru generarea semnalului baud_en
--pentru un ceas cu frecventa de 125 Mhz
          if rising_edge(clk) then
                if cnt = #Numar then
                  baud_en <= '1';
                  cnt <= (others => '0');
                else
                  baud_en <= '0';
                  cnt <= cnt + 1;
                end if;
              end if;
end process;
--Acest proces generează semnalul tx_en care activează transmisia de date. tx_en devine '1'
-- la inițierea transmisiei (tx_start = '1') și revine la '0' atunci când transmisia este completă
--(tx_digit_cnt = 3 și baud_en = '1')
 process(clk)
  begin
    if rising_edge(clk) then
      if tx_start = '1' then
        tx_en <= '1';
      elsif baud_en = '1' and tx_digit_cnt = 3 then
        tx_en <= '0';
      end if;
    end if;
  end process;

--Acest proces controlează contorul tx_digit_cnt, care numără cifrele din datele transmise.
--Contorul este resetat la 0 când începe transmisia (tx_start = '1') și se incrementează de
--fiecare dată când o cifra a fost transmisa (tx_rdy trece de la '0' la '1').
  process(clk)
  begin
    if rising_edge(clk) then
      tx_rdy1 <= tx_rdy; -- 1 clk delay
      if tx_start = '1' then
        tx_digit_cnt <= (others => '0');
      elsif tx_rdy = '1' and tx_rdy1 = '0' then
        tx_digit_cnt <= tx_digit_cnt + 1;
      end if;
    end if;
  end process;
```

```
--Acest proces încarcă datele din transmitionData în registrul tx_reg când tx_start devine activ
  process(clk)
  begin
    if rising_edge(clk) then
      if tx_start = '1' then
        tx_reg <= transmitionData;
      end if;
    end if;
  end process;

--Acest segment selectează câte un digit de 6 biți din registrul tx_reg pe baza valorii lui
--tx_digit_cnt. Se utilizează un multiplexor pentru a alege care parte a datelor va fi
--transmisă
  with tx_digit_cnt select
    tx_digit <= tx_reg(23 downto 18) when "00",
          tx_reg(17 downto 12) when "01",
          tx_reg(11 downto 6)  when "10",
          tx_reg(5 downto 0)   when "11",
          (others => 'X') when others;

--Acest segment convertește digitii selectați (6 biți) din tx_digit în caractere ASCII
--corespunzătoare (tx_data) pentru transmisie. Aceasta permite trimiterea caracterelor '0'
--până la '9' și 'A' până la 'Z'    with tx_digit select
     tx_data <= x"30" when "000000", -- '0'
           x"31" when "000001", -- '1'
           x"32" when "000010", -- '2'
           x"33" when "000011", -- '3'
                    ...  --TO DO De definit restul secventei
           x"39" when "001001", -- '9'
           x"41" when "001010", -- 'A'
           x"42" when "001011", -- 'B'
           x"43" when "001100", -- 'C'
                     ... --TO DO De definit restul secventei
           x"5A" when "100011", -- 'Z'
           (others => 'X') when others;

-- transmitfsm este un modul instanțiat care implementează mașina de stări finite pentru
--transmisia serială. Parametrii includ ceasul (clk), semnalele de control (tx_en, baud_en),
--datele de transmisie (tx_data), ieșirea UART (uart_txd_out), și semnalul care indică că datele
--sunt gata de transmisie (tx_rdy)
  inst_TFSM: transmitfsm port map(clk, '0', baud_en, tx_en, tx_data, tx, tx_rdy);

-- *************** UART Rx ***************
--blocul de instanțiere a unui FSM (Finite State Machine) pentru recepția datelor seriale
 inst_RFSM: receivefsm port map(clk, '0', baud_en_x16, rx, rx_data, rx_rdy);
```

```
--Acest proces este responsabil cu generarea ratei supra esantionare, utilizata in receptie
--TO DO scrieti care este valoare lui #Numar2 astfel incat sa se poata genera o rata de
--supra esantionare pentru semnalul de receptive pentru un baud rate de 9600 si o
--frecventa a ceasului de 125Mhz
  process(clk)
  begin
 --la fiecare muchie ascendenta a ceasului contorul cnt_x16 este incrementat
    if rising_edge(clk) then
--Când contorul ajunge la #Numar2, semnalul baud_en_x16 este activat ('1'), indicând
momentul în care se poate genera un puls pentru baud rate
      if cnt_x16 = #Numar2 then
--Semnalul baud_en_x16 este activ timp de un ciclu de ceas și se dezactivează imediat după
aceea, până când contorul ajunge din nou la #Numar2
        baud_en_x16 <= '1';
--Contorul este resetat la 0 după activarea semnalului, iar procesul se repetă
        cnt_x16 <= (others => '0');
      else
        baud_en_x16 <= '0';
        cnt_x16 <= cnt_x16 + 1;
      end if;
    end if;
  end process;

--un decodificator care în funcție de valoarea ascii primită în rx_data, mapează datele
primite la o valoare binară (rx_digit)
with rx_data select
    rx_digit <= "000000" when x"30", -- '0'
        "000001" when x"31", -- '1'
        "000010" when x"32", -- '2'
        "000011" when x"33", -- '3'
        "000100" when x"34", -- '4'
        "000101" when x"35", -- '5'
        "000110" when x"36", -- '6'
                    ... --TO DO De definit restul secventei
        "001001" when x"39", -- '9'
        "001010" when x"41", -- 'A'
        "001011" when x"42", -- 'B'
        "001100" when x"43", -- 'C'
                     ... --TO DO De definit restul secventei
        "100011" when x"5A", -- 'Z'
        (others => 'X') when others;

--Acest semnal continue datele finale care vor fii transmise la calculator
--Cand sw1 este 1 se transmit un sir de 4 caractere constante
--Cand sw1 este 0 se transmite ultima valoare receptionata de modulul de receptie
transmitionData <= B"010110_010110_011001_000001" when sw(1) = '1' else rx_reg;
```

```
--Detectează când o nouă valoare (rx_digit) a fost recepționată prin UART (semnalul rx_rdy)
--In scenariul nostru se receptioneaza 4 grupuri de 6 biti
  process(clk)
  begin
    if rising_edge(clk) then
      rx_rdy1 <= rx_rdy; -- 1 clk delay
      if rx_rdy = '1' and rx_rdy1 = '0' then
        case rx_digit_cnt is
--Stochează secvențial bitii receptionati în registrul rx_reg, în funcție de valoarea contorului
--rx_digit_cnt.
                    when "00" => rx_reg(23 downto 18) <= rx_digit; -- 6 biți
                    when "01" => rx_reg(17 downto 12) <= rx_digit;
                    when "10" => rx_reg(11 downto 6)  <= rx_digit;
                    when "11" => rx_reg(5 downto 0)   <= rx_digit;
                    when others => rx_reg(5 downto 0) <= (others => 'X');
          end case;
--Incrementarea contorului permite ca următorul digit să fie stocat în altă parte a registrului,
 --astfel încât să fie colectate patru grupuri de câte 6 biți (un total de 24 de biți)
            rx_digit_cnt <= rx_digit_cnt + 1;
      end if;
    end if;
  end process;
end Behavioral;
```

1.4.4 Configurarea fişierului de constrângeri XDC

PMod-ul USBUART poate fi introdus într-o bareta de pini a plăcii Zybo fără a necesita lipituri adiţionale. În acest caz vom alege portul JC pentru a introduce componenta. Pentru a putea utiliza pinii RX si TX ai componentei trebuie să le identificam mai întâi numărul în bareta de pini şi apoi numele, pentru a le putea găsi mai apoi in fişierul de constrângeri *.xdc. În Tabelul II preluat din [7] sunt numele şi numărul pinilor de pe fiecare port.

Tabelul II. Numărul şi numele pinilor de pe fiecare port al plăcii Zybo 7000

Pmod JA (XADC)	Pmod JB (Hi-Speed)	Pmod JC (Hi-Speed)	Pmod JD (Hi-Speed)	Pmod JE (Hi-Speed)	Pmod JF (MIO)
JA1: N15	JB1: T20	JC1: V15	JD1: T14	JE1: V12	JF1: MIO-13
JA2: L14	JB2: U20	JC2: W15	JD2: T15	JE2: W16	JF2: MIO-10
JA3: K16	JB3: V20	JC3: T11	JD3: P14	JE3: J15	JF3: MIO-11
JA4: K14	JB4: W20	JC4: T10	JD4: R14	JE4: H15	JF4: MIO-12
JA7: N16	JB7: Y18	JC7: W14	JD7: U14	JE7: V13	JF7: MIO-0
JA8: L15	JB8: Y19	JC8: Y14	JD8: U15	JE8: U17	JF8: MIO-9
JA9: J16	JB9: W18	JC9: T12	JD9: V17	JE9: T17	JF9: MIO-14
JA10: J14	JB10: W19	JC10: U12	JD10: V18	JE10: Y17	JF10: MIO-15

De asemenea, în Figura 1.13 este prezentată o bareta de pini generică în care este ilustrat locul de unde se începe numerotarea pinilor.

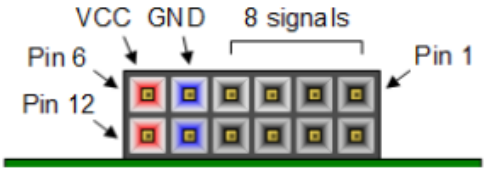

Figura 1.13. Exemplu unei barete de pini generice unde se evidentiaza seminificatia pinilor si cum se numeroteaza acestia pe placile Zybo

Pinii de pe PMod care sunt de interes pentru comunicarea UART sunt pinii 2 si 3 care corespund semnalelor Rx si Tx (vezi Tabelul I). Pe bareta de pini, la portul JC, aceşti pini corespund intrărilor JC2 (W15) si JC3 (T11). Aceste intrări sunt identificate apoi în fişierul de constrângeri şi sunt modificate în tx si rx pentru a fi consecvente cu denumirile din fişierul top.vhd. De asemenea, au fost decomentate toate liniile din fişierul de constrângeri care au legătura cu elementele prezente în entitatea fişierului top.vhd. Puteţi comenta celelalte declaraţii pe care nu le folosiţi. După modificări fişierul de constrângeri trebuie să arate ca şi cel de mai jos.

```
##Clock signal
set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 } [get_ports { clk }];
###Switches
set_property -dict { PACKAGE_PIN G15   IOSTANDARD LVCMOS33 } [get_ports { sw[0] }];
set_property -dict { PACKAGE_PIN P15   IOSTANDARD LVCMOS33 } [get_ports { sw[1] }];
set_property -dict { PACKAGE_PIN W13   IOSTANDARD LVCMOS33 } [get_ports { sw[2] }];
set_property -dict { PACKAGE_PIN T16   IOSTANDARD LVCMOS33 } [get_ports { sw[3] }];
###Buttons
set_property -dict { PACKAGE_PIN R18   IOSTANDARD LVCMOS33 } [get_ports { btn[0] }];
set_property -dict { PACKAGE_PIN P16   IOSTANDARD LVCMOS33 } [get_ports { btn[1] }];
set_property -dict { PACKAGE_PIN V16   IOSTANDARD LVCMOS33 } [get_ports { btn[2] }];
set_property -dict { PACKAGE_PIN Y16   IOSTANDARD LVCMOS33 } [get_ports { btn[3] }];
##LEDs
set_property -dict { PACKAGE_PIN M14   IOSTANDARD LVCMOS33 } [get_ports { led[0] }];
set_property -dict { PACKAGE_PIN M15   IOSTANDARD LVCMOS33 } [get_ports { led[1] }];
set_property -dict { PACKAGE_PIN G14   IOSTANDARD LVCMOS33 } [get_ports { led[2] }];
set_property -dict { PACKAGE_PIN D18   IOSTANDARD LVCMOS33 } [get_ports { led[3] }];
##Pmod Header JC
set_property -dict { PACKAGE_PIN W15   IOSTANDARD LVCMOS33 } [get_ports { tx }];
set_property -dict { PACKAGE_PIN T11   IOSTANDARD LVCMOS33 } [get_ports { rx }];
```

După ce creaţi acest fişier de constrângeri, setaţi-l ca şi target constraint file. Modul in care se creează un fişier de constrângeri in Vivado este prezentat in [2].

1.4.5. Vizualizarea Rezultatelor

După ce finalizaţi implementarea UART-ului în vhdl, generaţi bitstreamul apăsând pe butonul Generate Bitstream din Flow Navigator si apoi apăsaţi Ok. Această etapa se poate vizualiza in Figura 1.14.

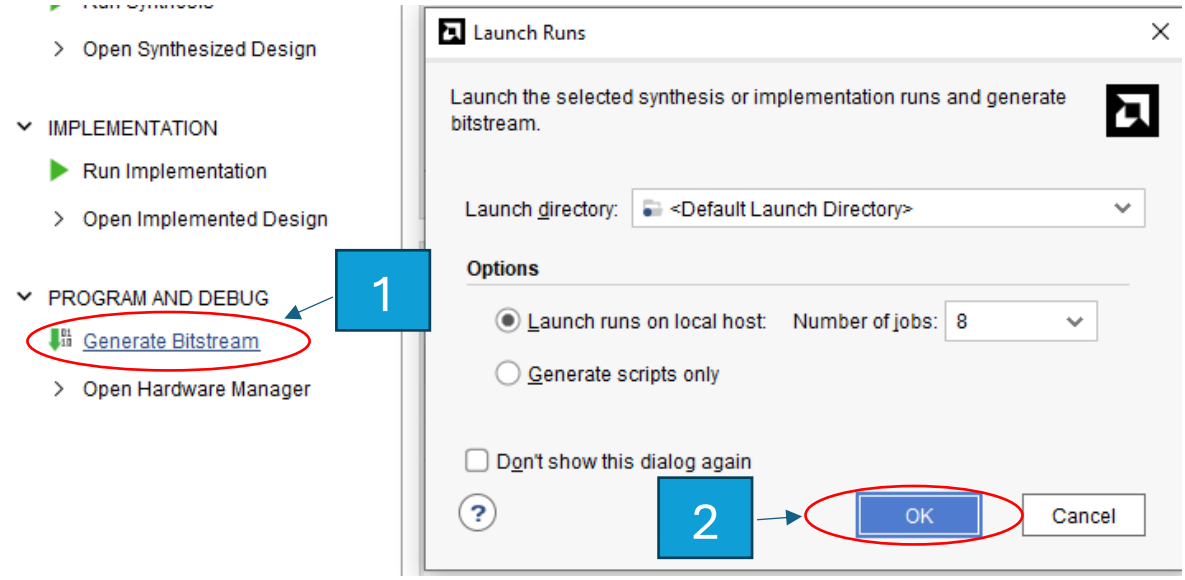

Figura 1.14. Generarea bitstreamului pentru componenta UART. Mai întâi se apasă butonul Generate Bitstream, iar apoi butonul Ok.

După ce s-a finalizat de generat bitstreamul încărcați-l pe plăcută Zybo. Acest lucru este realizat ca și pe orice alt FPGA din mediul Vivado. Dați click pe **Open Hardware Manager-> Open Target->Open New Target->Next->Next->Finish.** Apoi apăsați pe butonul **Program Device**, selectați plăcuța si apăsați pe butonul **Program** pentru a scrie fișierul *.bit pe placă. Acest proces, de încărcare a fișierului *.bit pe placă este descris si in [2].

Odată încărcat fișierul pe Zybo, deschideți aplicația de terminal preferată și realizați setările corespunzătoare aplicației noastre. În acest exemplu vom folosi aplicația HTerm [4]. Aceste setări sunt vizibile și în Figura 1.15. Portul serial este selectat conform indicațiilor prezentate în Figura 1.10, ceilalți parametri sunt baud rate: 9600, Biți de Date 8, Biți de Stop 1, Biți de paritate None. Odată realizate aceste configurații apăsați pe butonul **Connect**.

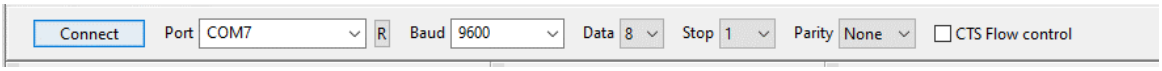

Figura 1.15. Configurația HTerm pentru aplicația noastră de UART

După ce ați realizat aceasta configurație, având switchul numărul 1 pe ‚1' logic apăsați pe butonul 0. Veți observa afișarea unui mesaj constant „MMP1" în terminal precum în Figura 1.16.

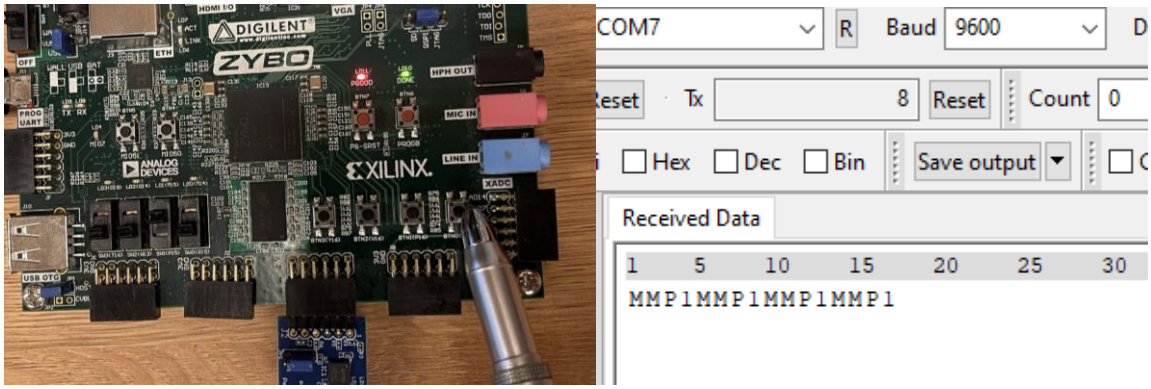

Figura 1.16. Prezentarea partii de transmitere a unui mesaj constant (switchul 1 e '1')

Pentru a testa modulul de recepţie, pe calculator in programul HTerm, în text boxul de la **Input control** introduceţi 4 caractere (spre exemplu „ZYBO”) si apăsaţi tasta **Enter**. Mutaţi apoi switchul 1 pe ‚0’ logic şi apăsaţi butonul 0. În figura 1.17 se poate observa rezultatul transmisiei ultimelor 4 caractere recepţionate.

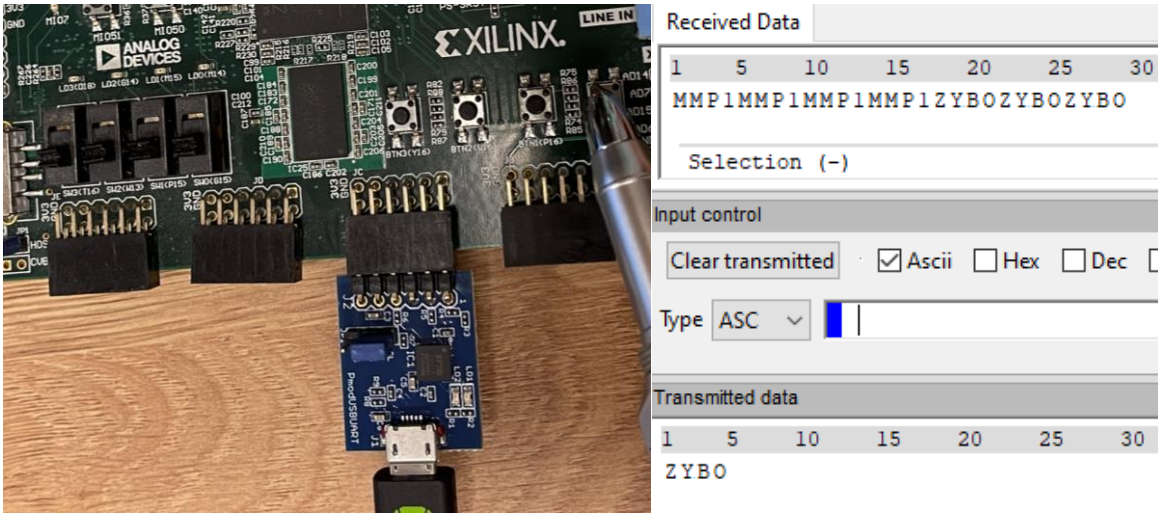

Figura 1.17. Recepția a 4 caractere transmise pe linia serială

Exerciţii

1. Completaţi fişierul transmitfsm.vhd cu codul pentru automatul cu stări finite pentru modulul de transmisie, realizat conform Figurii 1.3.
2. Completaţi fişierul receivefsm.vhd cu automatul cu stări finite pentru a finaliza modulul de recepţie. Automatul cu stări finite trebuie să se bazeze pe Figura 1.6.
3. Calculaţi valorile **#Numar** şi **#Numar2** necesare divizoarelor de frecvenţă utilizate în modulele de transmisie şi recepţie pentru generarea baud rate-ului. Pentru a genera un semnal de baud rate folosind un ceas cu frecvenţa F_{clk} formula pentru calculul numărului de cicluri de ceas între fiecare impuls al semnalului baud_en este dată de ecuaţia (1).

$$Numar\ de\ cicluri = \frac{F_{clk}}{baud\ rate} - 1 \qquad (1)$$

   Spre exemplu pentru un ceas cu frecventa 100Mhz si un baud rate de 9600, numărul de cicluri ar fi cel din ecuaţia (2).

$$Numar\ de\ cicluri = \frac{100{,}000{,}000}{9600} - 1 \cong 10416 \qquad (2)$$

   Frecvenţa ceasului de la Zybo e de 125Mhz iar pentru recepţie se face supra eşantionare la o frecvenţă de 16 ori mai mare decât cea de transmisie, pentru a creşte precizia detecţiei caracterului transmis.

4. Completaţi decodificatoarele ASCII la binar şi invers din fişierul top.vhd cu toate valorile de la 0 la 9 si de la A la Z.
5. Implementaţi un modul prin care să selectaţi din mai multe baud rate-uri prestabilite (4800, 57600, 115200, 128000) pentru a face transmisia şi recepţia. Selecţia baudrate-ului dorit să fie realizată de pe switchurile 2 si 3.

Tema

În exemplul implementat în laborator se transmit şi se recepţionează doar 4 caractere. Îmbunătăţiţi codul oferit pentru a putea transmite, respectiv recepţiona, un număr aleator de caractere.

Bibliografie

[1] XST User Guide, https://users.utcluj.ro/~baruch/resources/ISE_14.7/xst.pdf
[2] ARHITECTURA CALCULATOARELOR Îndrumător de laborator, Florin ONIGA, Mihai NEGRU, Editura UTPRESS, Cluj-Napoca, 2019, ISBN 978-606-737-350-9
[3] PMod USB UART Reference Manual. [Ultima data a fost accesat in 11.09.2024] https://digilent.com/reference/pmod/pmodusbuart/reference-manual?redirect=1
[4] HTerm http://www.der-hammer.info/terminal/hterm.zip
[5] Tera Term https://teratermproject.github.io/index-en.html
[6] Linkul de Git cu boilerplate codul pentru Labul 1 la disciplina Structura Sistemelor de Calcul: https://github.com/mirceamp/StructuraSistemelorDeCalcul/tree/main/Lab%201/Boilerplate%20Code
[7]Manual Zybo, https://digilent.com/reference/programmable-logic/zybo/reference-manual

Anexa 1

Exemplu Generic de automat cu stari finite Mealy

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity mealy is
port(
                data : in integer range 0 to 15;
                clk : in std_logic;
                rst : in std_logic;
                sum : out integer range 0 to 45
);
end mealy;

architecture Behavioral of mealy is

        signal s1 : integer range 0 to 15;
        signal s2 : integer range 0 to 15;
        signal s3 : integer range 0 to 15;
        type state is ( a,b,c,d);
        signal cur, nex : state;
begin

mealy : process(data , cur,rst)
begin
                case cur is
                    when a =>
                        nex <= b;
                    when b =>
                         nex <= c;
                    when c =>
                        nex <= d;
                    when d =>
                        nex <= a;
                end case;
end process mealy;

clock : process(clk,data,rst)
begin

if(rst = '1') THEN
   sum <= 0;
   cur <= a;
else
   if(clk'EVENT AND CLK = '1') then
```

```
        case cur is
                when a =>
                        s1 <= data;
                when b =>
                        s2 <= data;
                when c =>
                        s3 <= data;
                when d =>
                        sum <= s1 + s2 + s3;
        end case;
        cur <= nex;
    end if;
end if;
  end process clock;

end Behavioral;
```

Exemplu Generic de automat cu stari finite Moore

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity moore_fsm is
  port (
    clk    : in  std_logic;
    reset  : in  std_logic;
    output : out std_logic_vector(1 downto 0)
  );
end moore_fsm;

architecture Behavioral of moore_fsm is

  type state_type is (S0, S1, S2);
  signal state, next_state: state_type;

begin
  -- Proces de gestionare a stării
  process(clk, reset)
  begin
    if reset = '1' then
      state <= S0;
    elsif rising_edge(clk) then
      state <= next_state;
    end if;
  end process;
```

```
  -- Proces de tranziție a stărilor
  process(state)
  begin
    case state is
      when S0 =>
        next_state <= S1;
      when S1 =>
        next_state <= S2;
      when S2 =>
        next_state <= S0;
      when others =>
        next_state <= S0;
    end case;
  end process;

  -- Ieșirile FSM de tip Moore, bazate pe starea curentă
  process(state)
  begin
    case state is
      when S0 =>
        output <= "00";
      when S1 =>
        output <= "01";
      when S2 =>
        output <= "10";
      when others =>
        output <= "00";
    end case;
  end process;

end Behavioral;
```