TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA

Facultatea de Automatică și Calculatoare

Specializarea: Calculatoare și Tehnologia Informației

**Structura Sistemelor de Calcul**

# Proiectarea unui circuit de procesare a semnalelor primite de la senzori

**Student:** Stănuț Denisa-Alexandra

**Grupa:** 30234

**Profesor laborator:** Bozdog Raluca-Delia

An universitar: 2025–2026

# Cuprins

# Capitol 1

# Introducere

## 1.1 Propunere de proiect

Proiectul urmărește proiectarea și implementarea unui circuit digital în **VHDL**, destinat prelucrării în timp real a datelor de la senzori. Sistemul folosește protocolul **AXI4-Stream** pentru transferul eficient de date între modulele interne ale plăcii FPGA și comunică cu un microcontroler printr-o interfață **UART**. Datele primite sunt filtrate și prelucrate statistic, iar rezultatele sunt afișate pe un **afișaj cu 7 segmente**.

Implementarea pe FPGA permite folosirea avantajelor oferite de procesarea paralelă, aspecte importante pentru funcționarea în timp real a sistemului. Sistemul se potrivește bine în aplicații IoT, deoarece poate procesa local semnalele, reducând volumul de date transmis.

### 1.1.1 Descriere generală

Arhitectura propusă este alcătuită din patru module principale:

- **Modulul de recepție UART** – primește valorile măsurate de microcontroler prin interfața serială și le transformă în date paralele, ușor de prelucrat de către FPGA.
- **Modulul AXI4-Stream** – gestionează transferul datelor între blocurile interne și controlează semnalele de sincronizare (*TVALID*, *TREADY*, *TLAST*) pentru o comunicare eficientă.
- **Modulul de procesare** – prelucrează datele primite, aplicând filtre și calcule statistice.
- **Modulul de afișare** – convertește rezultatul în format **BCD** și îl trimite către afișajul cu 7 segmente, pentru vizualizarea în timp real a valorilor calculate.

Întregul sistem funcționează sincronizat la un semnal de ceas comun, ceea ce îi permite să prelucreze datele senzorilor într-un mod stabil și precis. Proiectul va fi implementat pe o platformă **FPGA** compatibilă cu **Xilinx Vivado**, astfel încât testarea și conectarea modulelor să fie cât mai simple. Pe termen lung, arhitectura propusă permite adăugarea ușoară a mai multor senzori sau implementarea unor algoritmi mai avansați.

### 1.1.2 Obiective

Obiectivele principale ale proiectului sunt:

1. Dezvoltarea unei interfețe **UART** capabila să recepționeze corect datele transmise de microcontroller.
2. Implementarea unei conexiuni interne bazate pe protocolul **AXI4-Stream**, pentru un transfer eficient și sincronizat al datelor între modulele FPGA.
3. Crearea unui modul de procesare care realizează operații de filtrare și calcul statistic în timp real, asigurând rezultate precise și relevante.
4. Dezvoltarea unui modul de afișare pe 7 segmente, pentru vizualizarea clară a valorilor procesate.
5. Testarea și validarea completă a întregului sistem, atât prin simulări, cât și prin implementare pe hardware.

Prin realizarea acestor obiective, sistemul va putea prelucra și afișa automat valorile măsurate, evidențiind avantajele implementării hardware, în special în ceea ce privește viteza de procesare.

### 1.1.3 Motivație și aplicații practice

Procesarea semnalelor provenite de la senzori devine tot mai importantă în domenii variate, precum automatizările industriale, sistemele IoT si multe alte domenii. Realizarea prelucrării direct pe FPGA permite reducerea transferului de date către procesor și, implicit, scade consumul de energie, făcând sistemul mai eficient și mai rapid.

Acest proiect oferă oportunitatea de a dezvolta competențe practice în proiectarea hardware, procesarea semnalelor și comunicarea în sisteme embedded.

## 1.2 Planificare

- **Săptămâna 4 (22 oct):** Propunerea proiectului, planificarea etapelor de lucru, studiu bibliografic.
- **Săptămâna 6 (5 nov):** Proiectarea modulului UART Receiver.
- **Săptămâna 8 (19 nov):** Implementarea modulului AXI4-Stream și testarea fluxului de date.
- **Săptămâna 10 (3 dec):** Proiectarea modului de afișare și integrarea acestuia în design.
- **Săptămâna 12 (17 dec):** Interconectare componente, testare, rezultate experimentale.

# Capitol 2

# Studiu Bibliografic

## 2.1 Notiuni teoretice

Inainte de realizarea efectivă a proiectului, a fost necesară o intelegere clară a diferențelor dintre procesarea software și cea hardware. Intr-un sistem bazat pe microcontroller, instrucțiunile sunt executate una câte una, în mod secvențial. În schimb, într-un **FPGA** (Field Programmable Gate Array), logica este implementată direct în hardware, iar mai multe operatii pot rula în paralel.

Această caracteristică face ca FPGA-urile să fie mult mai rapide în aplicații de tip *real-time*, unde datele trebuie procesate imediat ce sunt primite, fără intarzieri mari.

Arhitectura generală a sistemului este împărțită în patru blocuri principale:

- **Modulul UART** – primeste date seriale de la microcontrollerul Arduino;
- **Magistrala AXI4-Stream** – asigură transferul de date intre modulele FPGA într-un mod sincronizat;
- **Modulul de procesare** – prelucrează valorile primite, aplicând un filtru de medie mobilă;
- **Modulul de afisare** – preia rezultatul filtrat și îl afisează pe un display cu 7 segmente.

Astfel, valorile pleacă de la senzor, sunt convertite, filtrate si apoi afisate in timp real.

## 2.2 Prezentarea protocolului AXI4-Stream

Protocolul **AXI4-Stream** (Advanced eXtensible Interface) este un standard dezvoltat de ARM și utilizat pe scară largă în designul pe FPGA, în special în arhitecturile Xilinx. Scopul său este de a transmite fluxuri de date între module independente, intr-un mod eficient, fără a fi nevoie de adresare sau control extern complex.

AXI4-Stream functionează pe baza unui mecanism foarte simplu de *handshake* între două module:

- **TVALID** – semnal de la master (transmițător), care indică faptul că datele sunt valide;
- **TREADY** – semnal de la slave (receptor), care arată că este pregătit să primească date;
- **TDATA** – magistrala efectivă de date (8, 16, 32 sau 64 biți, în functie de aplicație);
- **TLAST** – marchează sfârșitul unui pachet de date (opțional);
- **TKEEP** – arată care bytes din `TDATA` sunt valizi (util în transferuri partiale).

## 2.3 Prezentarea arhitecturii FPGA și a limbajului VHDL

Un FPGA (Field Programmable Gate Array) este un circuit integrat care poate fi configurat după nevoile utilizatorului. In interior, conține mai multe tipuri de resurse:

- **CLB (Configurable Logic Blocks)** – conțin elementele logice de bază (LUT-uri, flip-flopuri);
- **Block RAM** – memorie internă foarte rapidă pentru stocarea temporară a datelor;
- **DSP Slices** – blocuri specializate pentru operatii matematice rapide (sumă, produs, comparatii);
- **Clock Management Units** – circuite care gestionează semnalele de ceas și sincronizarea sistemului.

Pentru a descrie modul în care aceste resurse sunt folosite, se utilizează limbaje hardware precum **VHDL** sau **Verilog**. VHDL (VHSIC Hardware Description Language) este limbajul folosit în acest proiect. El permite descrierea atat a comportamentului logic (ce face circuitul), cât și a structurii (cum sunt conectate componentele intre ele).

Un modul VHDL este alcătuit din două părți:

- **Entity** – defineste interfața (intrări, iesiri, dimensiunea semnalelor);
- **Architecture** – descrie comportamentul intern (cum reacționează semnalele la ceas sau la intrări).

Avantajul major al VHDL-ului este că toate instrucțiunile descrise în arhitectură se execută în paralel nu una după alta, ca într-un limbaj de programare clasic. Astfel, FPGA-ul poate procesa simultan mai multe operații diferite, crescând mult viteza de executie.

## 2.4 Metode de filtrare și prelucrare a semnalelor în timp real

Semnalele provenite de la senzori pot conține **zgomot electric**, variații rapide datorate interferențelor electromagnetice, erori de conversie analog–digitală sau chiar fluctuații de temperatura din mediul înconjurător. De aceea, este important ca aceste semnale să fie prelucrate înainte de afisare.

Una dintre cele mai simple și eficiente metode este **filtrul de medie mobila**. Acesta calculează media ultimelor $N$ valori primite, reducând variațiile bruște și obținând un semnal mai stabil.

Formula generală este:

$$m[n] = \frac{1}{N} \sum_{i=0}^{N-1} x[n-i]$$

Pe FPGA, implementarea se face ușor și eficient folosind:

- un **buffer circular** care retine ultimele $N$ valori;
- un **acumulator** (sumă) care se actualizează la fiecare nouă valoare;
- un **shift logic la dreapta** pentru a calcula media (dacă $N$ este o putere a lui 2).

Astfel, în loc să se facă o împărțire costisitoare în hardware, se face o simplă deplasare de biți.

Principalul avantaj este stabilitatea valorilor afisate. Singurul dezavantaj este o mică întârziere între valoarea reală și cea afisată, deoarece filtrul are nevoie de mai multe eșantioane pentru a calcula media.

Pe lângă filtrare, se pot calcula în timp real și alte statistici simple (valoare minimă, maximă, medie pe termen lung), dar pentru acest proiect am ales filtrul de medie mobilă.

Comunicarea dintre microcontroller și FPGA se realizează prin protocolul **UART** (Universal Asynchronous Receiver-Transmitter). Datele sunt trimise bit cu bit intr-un cadru format din:

[Bit start] [8 biți de date] [Bit stop]

La recepție, FPGA-ul folosește un mic **automat cu stări** care detectează începutul și sfârșitul fiecărui cadru și reconstruiește octetul original.

Rezultatul procesării este apoi transmis către un **display cu 7 segmente**, unde valoarea filtrată este afisată in timp real, sub formă numerică.

# Capitol 3

# Analiza

## 3.1 Analiza fluxului de date intre modulele FPGA

Sistemul proiectat are la baza un flux de date unidirectional, in care informatia este colectata, transmisa, procesata si afisata in timp real. Arhitectura generala poate fi exprimata prin urmatorul lant functional:

Senzor $\longrightarrow$ Microcontroller $\longrightarrow$ UART $\longrightarrow$ FPGA $\longrightarrow$ AXI4-Stream $\longrightarrow$ Procesare $\longrightarrow$ SSD

- **Senzorul** (de tip DS18B20 sau LM35) masoara temperatura si furnizeaza o valoare numerica la fiecare 30 de secunde.
- **Microcontrollerul** (Arduino UNO) citeste datele de la senzor, le converteste intr-un format binar pe 16 biti si le transmite catre FPGA prin interfata seriala UART, incapsulate intre octetii de start (`0xAA`) si stop (`0x55`).
- **FPGA-ul** primeste octetii prin modulul `UART_RX`, verifica integritatea pachetului in `PACKET_DECODER`, apoi transmite valoarea rezultata sub forma de flux AXI4-Stream pe 32 de biti.
- **Modulul de procesare** (`Sliding_Average_AXI`) calculeaza media valorilor primite in timp real, filtrand variatiile bruste.
- **SSD-ul** (`DISPLAY_7SEG`) afiseaza valoarea medie, actualizata periodic.

Fluxul dintre module este *asincron* la nivel de tranzactii AXI, intrucat fiecare bloc poate lucra la o viteza interna diferita. Sincronizarea si integritatea fluxului sunt garantate prin semnalele **TVALID** si **TREADY**, care asigura controlul de flux si evita pierderile de date. Comunicarea dintre etape este gestionata de un buffer FIFO de tip **AXI4-Stream Data FIFO**, care decupleaza temporal modulele si asigura o curgere stabila a datelor in intregul sistem.

Prin aceasta arhitectura, fiecare valoare provenita de la senzor este captata, procesata si afisata in timp real, fara intarzieri perceptibile si fara risc de pierdere a informatiei.

## 3.2 Analiza protocolului AXI4-Stream si a semnalelor asociate

Protocolul **AXI4-Stream** (Advanced eXtensible Interface 4 – Stream) este un standard dezvoltat de ARM pentru transferul de date in flux (*stream*) intre module hardware, fara a mai

fi nevoie de adresare sau magistrala comuna. Este ideal pentru aplicatii de procesare in timp real, unde mai multe blocuri lucreaza in lant (pipeline), fiecare cu propria frecventa, dar toate sincronizate printr-un mecanism simplu de control.

### Semnalele principale

| Semnal | Directie | Descriere |
|---|---|---|
| `TDATA` | Master → Slave | Vectorul de date transmis intre module, in acest proiect, are o latime de 32 de biti |
| `TVALID` | Master → Slave | Indica faptul ca valoarea de pe `TDATA` este valida si gata pentru a fi citita |
| `TREADY` | Slave → Master | Indica faptul ca receptorul este pregatit sa accepte o noua valoare |
| `TLAST` | Master → Slave | Marcheaza sfarsitul unui pachet (nefolosit in aceasta aplicatie) |

Table 3.1: Semnalele principale ale protocolului AXI4-Stream utilizate in proiect

### Mecanismul de handshake

Transferul efectiv are loc doar atunci cand ambele semnale de control sunt active simultan:

$$\texttt{TVALID} = 1 \quad \wedge \quad \texttt{TREADY} = 1$$

Acest mecanism simplu de **handshake** asigura o comunicatie sincronizata intre module, fara sa fie nevoie ca toate sa lucreze la aceeasi frecventa de ceas.

### Avantaje

- Transfer de date continuu si stabil;
- Poate conecta usor module diferite intr-un lant de procesare;
- Control de flux automat, gestionat de semnalele `TVALID`/`TREADY`;
- Compatibilitate directa cu IP-urile Xilinx precum `AXI4-Stream FIFO`, `Floating Point`, etc.

## 3.3 Analiza algoritmului de procesare

Scopul procesarii hardware este de a stabiliza semnalul de temperatura primit de la senzor si de a elimina fluctuatiile bruste cauzate de zgomot. Aceasta functie este realizata de modulul **Sliding_Average_AXI**, care implementeaza un **filtru de medie mobila**.

### 3.3.1 Filtrul de medie mobila

Filtrul de medie mobila calculeaza media aritmetica a ultimelor $N$ valori primite. Astfel, variatiile rapide sunt amortizate, iar iesirea devine mult mai stabila. Formula generala este:

$$m[n] = \frac{1}{N} \sum_{i=0}^{N-1} x[n-i]$$

In loc sa recalculeze suma completa la fiecare pas, sistemul foloseste o forma optimizata:

$$S' = S - a + n, \quad m' = \frac{S'}{N}$$

unde:

- $S$ – suma curenta a ultimelor valori din fereastra de mediere;
- $a$ – valoarea cea mai veche, care urmeaza sa fie eliminata;
- $n$ – noua valoare de intrare;
- $S'$ – suma actualizata, folosita pentru calculul noii medii.

Pentru hardware, impartirea cu $N$ se face printr-un simplu **shift logic la dreapta**, deoarece $N$ este o putere a lui 2 ($N = 16 \Rightarrow m = S >> 4$). Aceasta solutie reduce considerabil complexitatea aritmetica si economiseste resurse logice pe FPGA.

### Componente interne utilizate

Implementarea hardware a filtrului foloseste cateva blocuri simple dar eficiente:

- **Buffer circular** – retine ultimele 16 valori de intrare;
- **Acumulator pe 32 de biti** – mentine suma curenta;
- **Pointer circular** – indica pozitia din buffer care urmeaza sa fie inlocuita;
- **Adder**, **Subtractor** si **Register** – folosite pentru actualizarea sumei la fiecare nou esantion.

Astfel, noua valoare este adaugata in suma cu ajutorul sumatorului, cea veche este eliminata cu scazatorul, iar rezultatul este memorat intr-un registru sincron pe 32 de biti. La fiecare ciclu valid AXI4-Stream, noua medie este calculata si trimisa mai departe catre modulul de afisare.

### Exemplu numeric

Pentru o fereastra de $N = 16$, daca media curenta este $m = 24.5$, valoarea eliminata este $a = 20$, iar cea noua este $n = 30$, noua medie devine:

$$m' = \frac{24.5 \times 16 - 20 + 30}{16} = 25.1$$

Dupa cateva iteratii, iesirea se stabilizeaza si variatiile devin foarte mici.

### Avantaje

- Structura simpla si usor de implementat pe FPGA;
- Timp de executie constant, indiferent de dimensiunea ferestrei;
- Valori stabile, fara salturi bruste la afisare.

**Dezavantaje**

- Reactie mai lenta la schimbari bruste ale temperaturii;
- Necesita memorie suplimentara pentru bufferul de date;
- Introduce o mica intarziere intre intrare si iesire.

### 3.3.2 Alegerea dimensiunii ferestrei de mediere

Alegerea lui $N$ este un compromis intre stabilitate si viteza de reactie. O fereastra mare face iesirea foarte stabila, dar mai lenta; una mica reactioneaza rapid, dar lasa mai mult zgomot. Pentru acest proiect, o valoare $N = 16$ este cea mai buna deoarece asa afisajul ramane stabil, dar raspunde suficient de repede la variatiile de temperatura.

Pentru a evita depasirea capacitatii interne, latimea acumulatoarelor trebuie aleasa cu atentie. La 32 de biti pentru fiecare valoare:

$$S_{\max} = N \cdot x_{\max} = 16 \times 2^{31}$$

De aceea, un registru de 64 de biti este folosit pentru suma interna, oferind siguranta si posibilitatea extinderii sistemului pe viitor.

### 3.3.3 Initializarea ferestrei

La pornirea sistemului, bufferul este gol. Pentru a preveni afisarea unor valori incorecte, exista doua variante:

1. **Initializare cu zero** – toate pozitiile bufferului sunt setate la 0, iar media se stabilizeaza treptat dupa primirea catorva valori;
2. **Pornire intarziata** – procesarea incepe doar dupa ce s-au primit primele $N$ valori valide.

In implementarea actuala s-a ales prima metoda, deoarece este mai simpla si duce la stabilizarea sistemului dupa primele iteratii.

### 3.3.4 Gestionarea overflow-ului si precizia calculului

Intrucat FPGA-ul foloseste registre cu dimensiune fixa, este esential sa se evite depasirea intervalului maxim al acumulatoarelor. Pentru $N = 16$ si date pe 32 de biti:

$$S_{\max} = 16 \times (2^{32} - 1)$$

Prin urmare, latimea acumulatoarelor a fost extinsa la 64 de biti, pentru a elimina complet riscul de overflow si a mentine precizia rezultatelor pe termen lung.

## 3.4 Sincronizare si temporizare

### Sincronizarea datelor UART

Transmisia dintre Arduino si FPGA se face la un baud rate fix de **9600 bps**. Pentru formatul standard **8N1**, durata unui octet transmis este:

$$T_{\text{byte}} = \frac{10}{B}$$

### Sincronizarea interna a modulelor

Toate modulele vor functiona sincronizat cu semnalul de ceas principal `clk`. `FIFO` va permite fluxului sa ramana stabil chiar daca viteza de procesare a diferitelor blocuri variaza.

### Temporizarea afisajului

Afisajul cu 7 segmente este comandat prin **multiplexare rapida**: fiecare cifra este activata pe rand, dar comutarea se face atat de repede incat ochiul uman percepe toate cifrele aprinse simultan. Pentru o afisare stabila, se respecta relatia:

$$f_{\text{refresh}} = N_{\text{cifre}} \times 1 \text{ kHz}$$

Astfel, pentru 4 cifre, frecventa totala de actualizare este aproximativ 4 kHz.

# Capitol 4

# Design

## 4.1 Prezentare generala

Sistemul proiectat are ca scop masurarea temperaturii mediului folosind un senzor analogic, prelucrarea digitala a datelor pe o placa **FPGA Basys 3** si afisarea rezultatului filtrat pe un **display cu 7 segmente**.

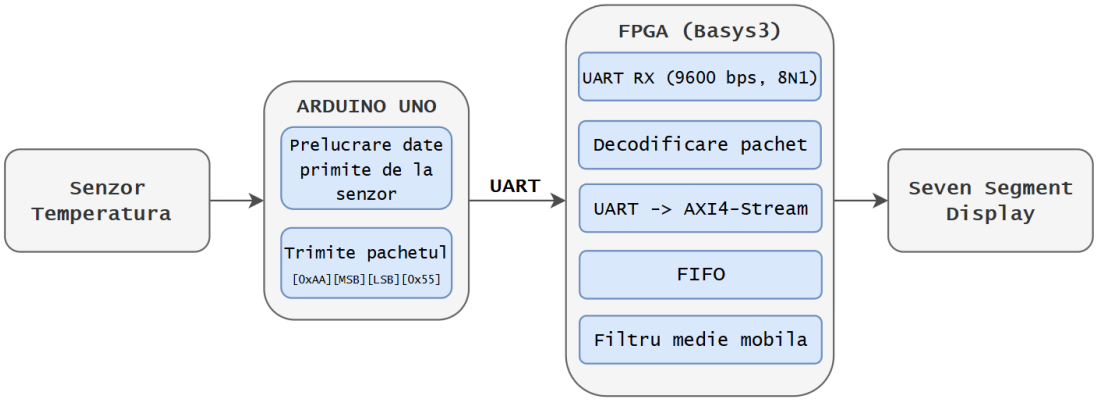



Figure 4.1: Diagrama generala

Sistemul este impartit in doua parti principale:

- **Partea analogica (Arduino + Senzor)** – se ocupa cu masurarea temperaturii si trimiterea valorilor catre FPGA prin UART.s
- **Partea digitala (FPGA – Basys 3)** – primeste valorile, le filtreaza, le transforma si le afiseaza in timp real pe un display cu 7 segmente.

Fluxul complet de date poate fi reprezentat astfel:

Senzor $\longrightarrow$ Arduino (ADC + UART) $\longrightarrow$ FPGA (AXI + Filtru) $\longrightarrow$ Afisaj 7-Segmente

Pe scurt, Arduino masoara temperatura si o trimite, FPGA-ul o proceseaza si afiseaza rezultatul, iar utilizatorul vede o valoare stabila si corecta pe ecran.

### 4.1.1 Functionarea senzorului DHT11

În cadrul acestui proiect am utilizat senzorul de temperatură **DHT11**, un **senzor digital**. Senzorul este capabil să măsoare atât **temperatura mediului**, cât și **umiditatea**, furnizând valorile sub formă de date digitale, fără a necesita conversie analog–digitală suplimentară.

Pentru scopul acestui proiect, a fost utilizată **exclusiv componenta de măsurare a temperaturii**, valorile de umiditate fiind ignorate. Temperatura este exprimată direct în grade Celsius, cu o rezoluție adecvată aplicațiilor de monitorizare ambientală, ceea ce permite prelucrarea ulterioară a datelor într-un mod eficient și stabil.

Un avantaj important al senzorului DHT11 este faptul că acesta comunică printr-un **protocol digital dedicat**, controlat prin software, folosind o singură linie de date.

Datorită acestor caracteristici, DHT11 reprezintă o soluție potrivită pentru transmiterea și procesarea digitală a datelor de la senzor, prin microcontroller, până la prelucrarea pe FPGA și afișarea rezultatului filtrat.

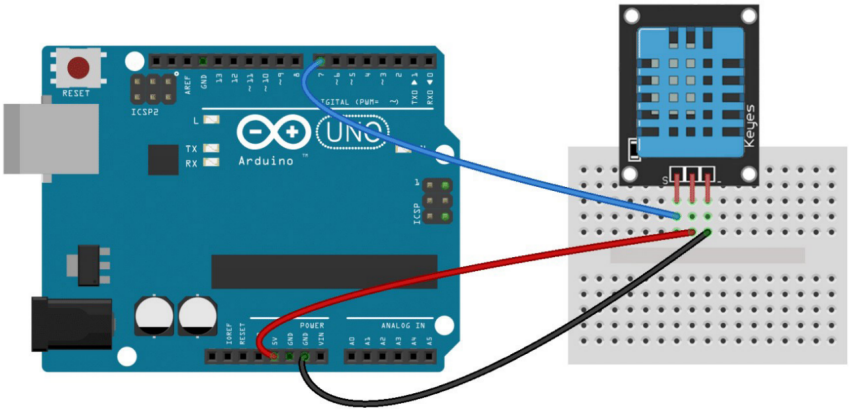



Figure 4.2: Schema de conectare a senzorului DHT11

Conectarea senzorului pe placa Arduino UNO este urmatoarea:

- Pinul **VCC** al senzorului la iesirea de **5V** de pe Arduino;
- Pinul **DATA** conectat la un pin digital Arduino ( `D7`);
- Pinul **GND** la masa (*GND*).

Spre deosebire de alti senzori digitali precum DS18B20, senzorul DHT11 nu necesita o rezistenta externa de tip *pull-up*, aceasta fiind integrata in modulul senzorului. Comunicarea dintre Arduino si senzor se realizeaza printr-un protocol digital propriu, sincronizat prin software, controlat complet de microcontroler.

### 4.1.2 Citirea valorilor pe Arduino

Arduino comunică cu senzorul **DHT11** printr-un **protocol digital**, care utilizează o singură linie de date pentru transmiterea informațiilor. Spre deosebire de senzorii analogici (precum LM35), DHT11 furnizează direct valoarea temperaturii în format digital, eliminând necesitatea unui convertor ADC și reducând influența zgomotului electric asupra măsurătorii.

Citirea senzorului este realizată cu ajutorul unei biblioteci software dedicate, care gestionează temporizările stricte impuse de protocolul DHT și asigură extragerea corectă a datelor de temperatură. Valoarea obținută este exprimată în grade Celsius și este prelucrată local pe Arduino înainte de a fi transmisă mai departe.

Transmiterea datelor către placa FPGA se realizează prin intermediul portului serial **UART**, la viteza de **9600 bps**. Datele sunt trimise periodic, la un interval prestabilit de aproximativ **5 secunde**, asigurând un flux stabil de informații către partea de procesare digitală.

### 4.1.3 Citirea valorilor pe Arduino

Arduino comunică cu senzorul prin protocolul **One-Wire**, utilizând o singură linie de date pentru transmiterea și recepția informațiilor. Spre deosebire de senzorii analogici (precum LM35), DS18B20 furnizează direct valoarea temperaturii în format digital, eliminând necesitatea unui convertor ADC și reducând erorile de măsurare.

Transmiterea datelor către FPGA se realizează prin portul serial **UART**, la viteza de **9600 bps**, folosind formatul standard **8N1** (8 biți de date, fără paritate, 1 bit de stop). Trimiterile au loc la un interval fix de aproximativ 30 de secunde.

### 4.1.4 Structura pachetului transmis

Pentru ca placa FPGA să poată identifica ușor fiecare transmisie, pachetele de date trimise de Arduino sunt delimitate clar. Formatul este următorul:

[START (0xAA)] [MSB] [LSB] [STOP (0x55)]

Table 4.1: Formatul Pachetului UART transmis de Arduino către FPGA

| Octet | Valoare | Descriere | Dimensiune |
|---|---|---|---|
| 1 | `0xAA` | Byte de START | 8 biți |
| 2 | MSB | Octetul superior al temperaturii | 8 biți |
| 3 | LSB | Octetul inferior al temperaturii | 8 biți |
| 4 | `0x55` | Byte de STOP | 8 biți |

**Exemplu:** Pentru temperatura 24.5°C (reprezentată ca `0x00F5` pe 16 biți):

```
[0xAA] [0x00] [0xF5] [0x55]
```

Astfel, logica implementată în FPGA (prin modulele `UART_receiver` și `Packet_Decoder`) poate recunoaște limitele fiecărui pachet, extrage valoarea efectivă a temperaturii și o trimite mai departe în lanțul de procesare AXI4-Stream.

## 4.2 Interfatarea pe FPGA

Arduino comunica cu FPGA-ul prin protocolul serial UART. Linia **TX** de pe Arduino este conectata la linia **RX** de pe Basys 3, iar masa comuna (GND) este obligatorie pentru referinta comuna de tensiune. Semnalul de 5V de la Arduino este coborat la 3.3V cu ajutorul unui convertor de nivel logic.

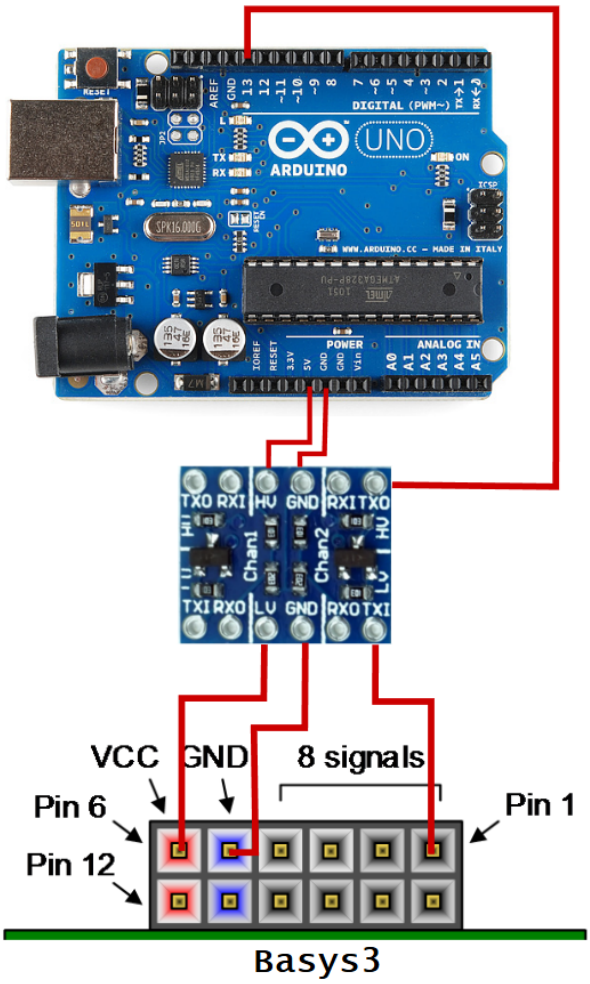



Figure 4.3: Schema de conectare intre Arduino si placa FPGA

Odata primite pe FPGA, datele seriale sunt prelucrate de un modul VHDL – `UART_receiver` care reconstruieste octetii si semnaleaza cand un pachet a fost receptionat.

## 4.3 Arhitectura interna pe FPGA

Pe FPGA, designul este organizat pe mai multe etape logice care proceseaza datele una dupa alta, folosind fluxul de date **AXI4-Stream**. Acest protocol permite transferul sincronizat intre module independente.
Fluxul este complet sincron si controlat prin semnalele `TVALID` si `TREADY`, astfel incat nicio valoare nu este pierduta chiar daca unul dintre module proceseaza mai lent.

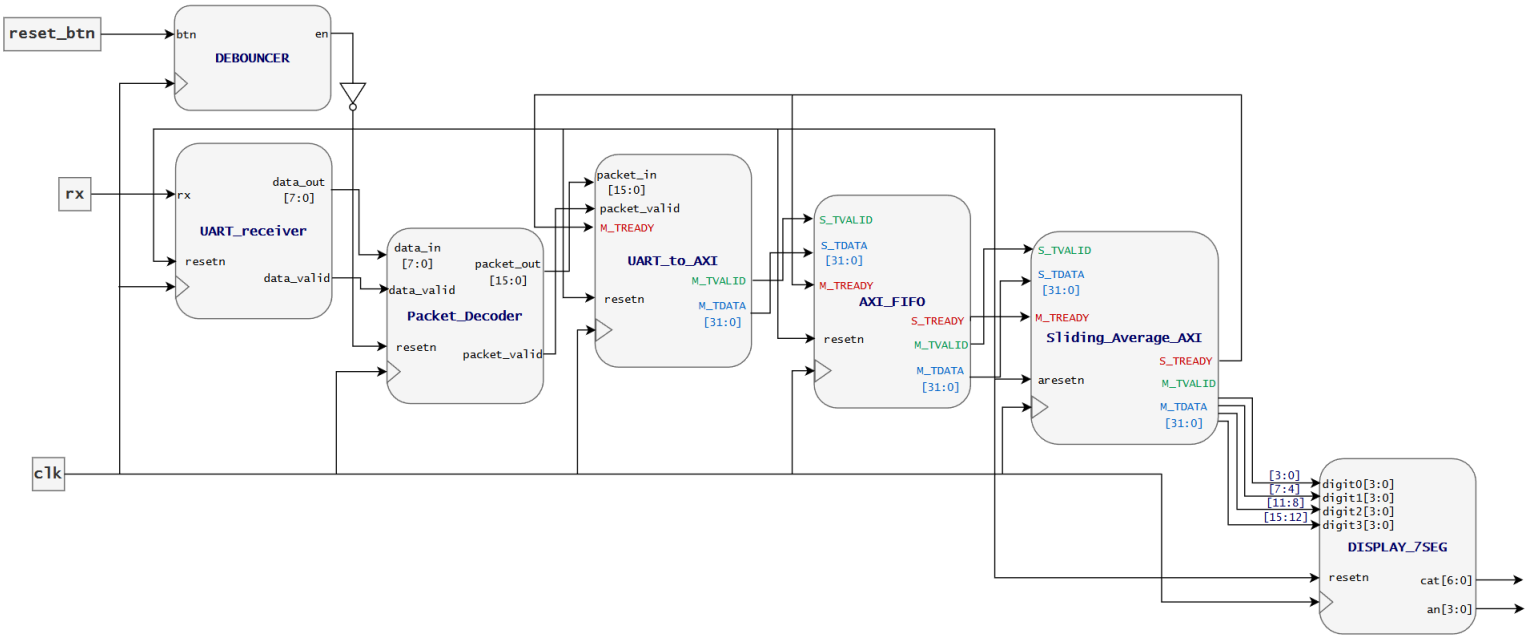



Figure 4.4: Structura logică internă a sistemului FPGA.

## 4.4 Componentele VHDL utilizate

### 4.4.1 Modulul `UART_receiver`

Are rolul de a receptiona datele seriale transmise de Arduino. Acesta identifica bitul de start (0 logic), citeste cei 8 biti de date, verifica bitul de stop (1 logic) si semnaleaza finalizarea unui octet.

### 4.4.2 Modulul `Packet_Decoder`

Acest modul citeste fluxul de octeti receptionati de UART si cauta secventa specifica de pachet (0xAA ... 0x55). Dupa detectarea capetelor, extrage cei doi octeti de date (MSB si LSB), formeaza o valoare pe 16 biti si o transmite catre modulul urmator pentru conversie AXI.

### 4.4.3 Modulul `UART_to_AXI`

Transforma fluxul de date serial in cuvinte de 32 de biti, pentru compatibilitate cu restul arhitecturii AXI4-Stream. Dupa ce aduna patru octeti, genereaza un semnal `TVALID = '1'` si emite pachetul complet catre FIFO.

### 4.4.4 Modulul `AXI_FIFO`

FIFO-ul este un buffer circular care poate stoca 16 valori pe 32 de biti. Rolul sau este de a sincroniza fluxul intre module cu viteze diferite. Daca modulul urmator nu este pregatit (`TREADY = '0'`), datele sunt pastrate intern pana cand pot fi transmise in siguranta.

### 4.4.5 Modulul `Sliding_Average_AXI`

Acest modul reprezintă blocul principal de procesare al sistemului, fiind responsabil pentru filtrarea valorilor de temperatură recepționate de la senzor. Scopul său este de a calcula o **medie mobilă** pe ultimele 16 valori succesive, reducând astfel fluctuațiile bruște și obținând o temperatură afișată mai stabilă și realistă.

Algoritmul implementat are forma:

$$S_{nou} = S_{vechi} - x_{vechi} + x_{nou}, \quad \text{Media} = S_{nou} >> 4$$

unde:

- $x_{nou}$ – noua valoare primită prin AXI4-Stream;
- $x_{vechi}$ – cea mai veche valoare din buffer (care urmează să fie înlocuită);
- $S_{vechi}$ – suma valorilor curente din fereastra de mediere;
- $S_{nou}$ – suma actualizată după adăugarea noii valori și eliminarea celei vechi.

Impartirea cu 16 (numarul de eșantioane din fereastră) se realizează printr-un simplu **shift logic la dreapta cu 4 poziții**, ceea ce echivalează cu o împărțire la $2^4$. Această metodă este foarte eficienta, deoarece nu necesita divizoare complexe, ci doar o deplasare de biți, economisind resursele logice ale FPGA-ului.

#### Structura interna si componentele utilizate

Implementarea hardware a modulului `Sliding_Average_AXI` se bazează pe trei blocuri VHDL de bază:

1. **Adder_32b** – sumator pe 32 de biți, care realizează operația de adunare $S_{intermediar} = S_{vechi} + x_{nou}$; Acest modul primește doi vectori de 32 de biți și generează rezultatul in același format, fiind folosit pentru acumularea valorilor noi.

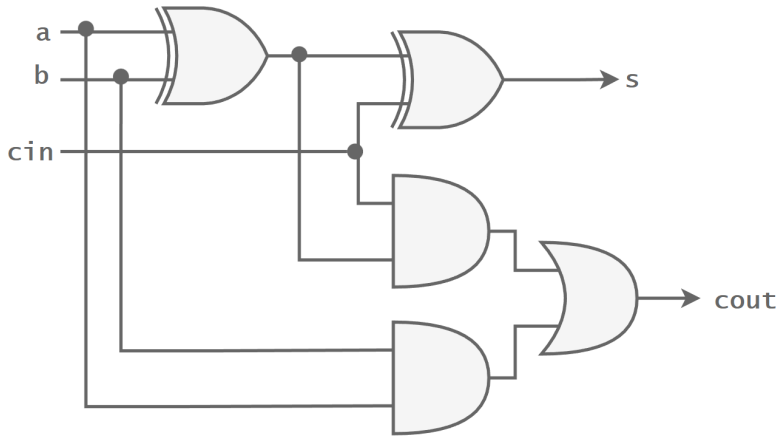



Figure 4.5: Sumator pe 1 bit cu porti logice

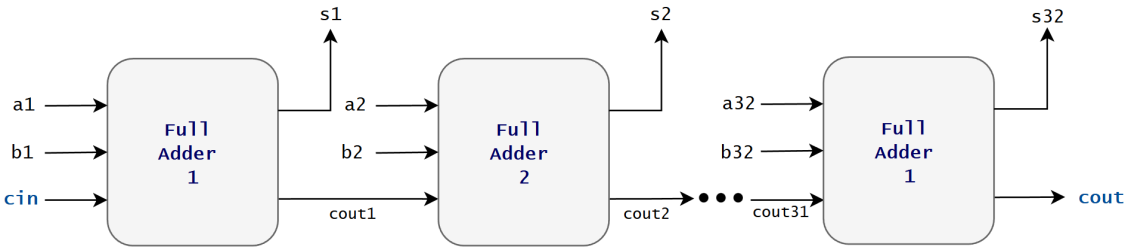



Figure 4.6: Sumatorul pe 32 de biti

2. **Subtractor_32b** – scazator pe 32 de biți, care efectuează operația $S_{nou} = S_{intermediar} - x_{vechi}$; Scăderea se realizeaza prin complementul fata de doi, iar rezultatul este folosit pentru actualizarea sumei curente.

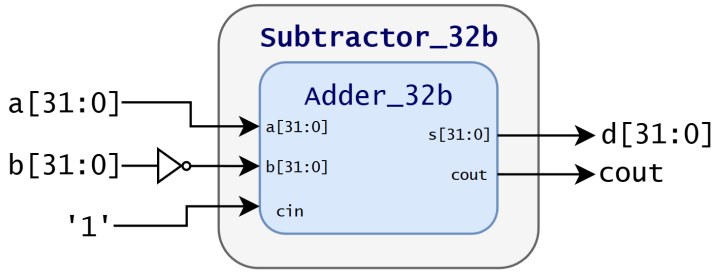



Figure 4.7: Scazatorul pe 32 de biti folosind sumatorul

3. **Register_32b** – registru sincron pe 32 de biți, utilizat pentru memorarea intermediară a sumei și pentru asigurarea sincronizării cu semnalul de ceas principal (`clk`); La fiecare ciclu de date (`TVALID = 1`), registrul stochează noua suma calculată, astfel incat procesarea următoarei valori să se bazeze pe starea corecta anterioară.

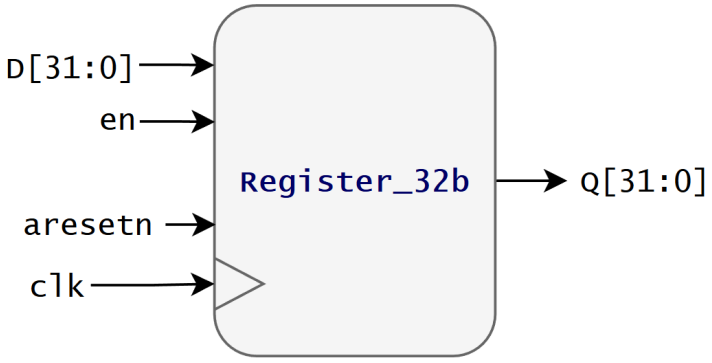



Figure 4.8: Registrul pe 32 de biti

### 4.4.6 Modulul `DISPLAY_7SEG`

Modulul final are rolul de a converti rezultatul filtrat in format zecimal si de a-l afisa pe cele 4 cifre ale display-ului cu 7 segmente. Afisajul este multiplexat, fiecare cifra se aprinde pe rand, dar la o frecventa suficient de mare, astfel incat ochiul uman percepe toate cifrele aprinse simultan.

# Capitol 5

# Implementare

In acest capitol se descriu etapele de implementare hardware. Toate modulele au fost dezvoltate în VHDL și interconectate prin intermediul protocolului **AXI4-Stream**, care permite un transfer de date sincronizat și eficient între blocuri.

## 5.1 Descrierea modulelor VHDL

### 5.1.1 Modulul `UART_Receiver`

Reprezintă interfața de recepție a datelor seriale provenite de la microcontroller. El transformă fluxul serial UART (8N1, 9600 bps) într-un cuvânt de 8 biți, sincronizat cu semnalul de ceas de 100 MHz.

Modulul detectează bitul de start ('0' logic), apoi citește fiecare bit de date la intervale stabilite printr-un contor intern. După bitul de stop ('1' logic), valoarea completă este disponibilă pe ieșirea `data_out`, iar semnalul `data_valid` devine activ pentru un ciclu.

```
entity UART_Receiver is
    Port( clk : in  STD_LOGIC;
          resetn : in  STD_LOGIC;
          rx : in  STD_LOGIC;
          data_out : out STD_LOGIC_VECTOR(7 downto 0);
          data_valid : out STD_LOGIC
    );
end UART_Receiver;
```

Listing 5.1: Entitatea modulului UART_Receiver

| Nume port | Direcție | Descriere |
|---|---|---|
| clk | in | Semnalul de ceas (100 MHz) |
| resetn | in | Reset activ pe '0' |
| rx | in | Linia de recepție UART |
| data_out | out | Byte-ul recepționat de la microcontroller |
| data_valid | out | Semnal de confirmare a datelor valide |

### 5.1.2 Modulul `Packet_Decoder`

Verifică structura pachetului UART primit și extrage valoarea pe 16 biți. Formatul transmis este:

`[START (0xAA)] [MSB] [LSB] [STOP (0x55)]`

```
entity Packet_Decoder is
    Port ( clk : in STD_LOGIC;
           resetn : in STD_LOGIC;
           data_in : in STD_LOGIC_VECTOR(7 downto 0);
           data_valid : in STD_LOGIC;
           packet_out : out STD_LOGIC_VECTOR(15 downto 0);
           packet_valid : out STD_LOGIC
    );
end Packet_Decoder;
```

Listing 5.2: Entitatea modulului Packet_Decoder

| Nume port | Direcție | Descriere |
|---|---|---|
| clk | in | Semnalul de ceas al sistemului |
| resetn | in | Reset activ pe '0' |
| data_in | in | Octetul recepționat de la UART |
| data_valid | in | Semnal de validare de la modulul UART_Receiver |
| packet_out | out | Valoarea formată din cei doi octeți de date |
| packet_valid | out | Semnal de confirmare a unui pachet |

### 5.1.3 Modulul `UART_to_AXI`

Modulul UART_to_AXI realizează conversia datelor pe 16 biți din formatul UART într-un cuvânt pe 32 biți compatibil cu AXI4-Stream. Acesta generează semnalele M_AXIS_TVALID și M_AXIS_TDATA pentru interconectarea cu FIFO.

```
entity UART_to_AXI is
    Port ( aclk : in STD_LOGIC;
           aresetn : in STD_LOGIC;
           packet_in : in STD_LOGIC_VECTOR(15 downto 0);
           packet_valid : in STD_LOGIC;
           m_axis_tvalid : out STD_LOGIC;
           m_axis_tready : in STD_LOGIC;
           m_axis_tdata : out STD_LOGIC_VECTOR(31 downto 0)
    );
end UART_to_AXI;
```

Listing 5.3: Entitatea modulului UART_to_AXI

| Nume port | Direcție | Descriere |
|---|---|---|
| aclk | in | Semnal de ceas global AXI |
| aresetn | in | Reset activ pe '0' (ARESETn conform standard AXI) |
| packet_in | in | Valoarea decodificată din pachetul UART |
| packet_valid | in | Confirmare pachet valid |
| m_axis_tdata | out | Cuvântul AXI transmis (date + zero padding) |
| m_axis_tvalid | out | Semnal TVALID - validare a datelor AXI |
| m_axis_tready | in | Semnal TREADY - confirmare de la modulul următor |

### 5.1.4 Modulul AXI_FIFO

Modulul AXI_FIFO stochează temporar pachetele AXI4-Stream pentru a menține fluxul de date stabil între blocuri care pot avea viteze diferite de procesare. Implementează atât interfață slave (intrare) cât și master (ieșire) conform protocolului AXI.

```
entity AXI_FIFO is
    Port ( clk : in STD_LOGIC;
           resetn : in STD_LOGIC;
           s_axis_tvalid : in STD_LOGIC;
           s_axis_tready : out STD_LOGIC;
           s_axis_tdata : in STD_LOGIC_VECTOR(31 downto 0);
           m_axis_tvalid : out STD_LOGIC;
           m_axis_tready : in STD_LOGIC;
           m_axis_tdata : out STD_LOGIC_VECTOR(31 downto 0);
           fifo_full : out STD_LOGIC;
           fifo_empty : out STD_LOGIC
    );
end AXI_FIFO;
```

Listing 5.4: Entitatea modulului AXI_FIFO

| Nume port | Direcție | Descriere |
|---|---|---|
| clk | in | Semnalul de ceas al sistemului |
| resetn | in | Reset activ pe '0' |
| s_axis_tvalid | in | Semnal TVALID de la modulul anterior |
| s_axis_tready | out | Indică faptul că FIFO poate primi date |
| s_axis_tdata | in | Datele de intrare în FIFO |
| m_axis_tvalid | out | Semnal TVALID către modulul următor |
| m_axis_tready | in | Confirmare de la modulul următor |
| m_axis_tdata | out | Datele furnizate la ieșirea FIFO |
| fifo_full | out | Indicator FIFO plin |
| fifo_empty | out | Indicator FIFO gol |

### 5.1.5 Modulul `Sliding_Average_AXI`

Acest modul implementează un filtru de medie mobilă cu o fereastră de 16 eșantioane, reducând zgomotul și variațiile rapide din semnalul de temperatură.

**Algoritm:**

$$S_{nou} = S_{vechi} - x_{vechi} + x_{nou}, \qquad \text{Media} = S_{nou} \gg 4$$

```
entity Sliding_Average_AXI is
    Generic ( WINDOW_SIZE : integer := 16 );
    Port ( aclk : in STD_LOGIC;
           aresetn : in STD_LOGIC;
           s_axis_tvalid : in STD_LOGIC;
           s_axis_tready : out STD_LOGIC;
           s_axis_tdata  : in STD_LOGIC_VECTOR(31 downto 0);
           m_axis_tvalid : out STD_LOGIC;
           m_axis_tready : in STD_LOGIC;
           m_axis_tdata  : out STD_LOGIC_VECTOR(31 downto 0)
    );
end Sliding_Average_AXI;
```

Listing 5.5: Entitatea modulului Sliding_Average_AXI

| Nume port | Direcție | Descriere |
|---|---|---|
| `aclk` | in | Semnalul de ceas global (ACLK) |
| `aresetn` | in | Reset activ pe '0' (ARESETn) |
| `s_axis_tvalid` | in | TVALID - validare intrare date |
| `s_axis_tready` | out | TREADY - confirmare disponibilitate de recepție |
| `s_axis_tdata` | in | TDATA - valoarea de intrare pentru filtrare |
| `m_axis_tvalid` | out | TVALID - semnal de validare a ieșirii |
| `m_axis_tready` | in | TREADY - confirmare de la modulul următor |
| `m_axis_tdata` | out | TDATA - rezultatul mediei mobile |

### 5.1.6 Modulul `Display_7SEG`

Modulul final realizează conversia valorii numerice rezultate în format BCD și afișarea acesteia pe cele patru cifre ale display-ului cu 7 segmente al plăcii Basys 3.

```
entity display_7seg is
    Port ( digit0 : in  STD_LOGIC_VECTOR (3 downto 0);
           digit1 : in  STD_LOGIC_VECTOR (3 downto 0);
           digit2 : in  STD_LOGIC_VECTOR (3 downto 0);
           digit3 : in  STD_LOGIC_VECTOR (3 downto 0);
           clk : in  STD_LOGIC;
           cat : out STD_LOGIC_VECTOR (6 downto 0);
           an : out STD_LOGIC_VECTOR (3 downto 0);
           dp : out STD_LOGIC
    );
end display_7seg;
```

Listing 5.6: Entitatea modulului display_7seg

| Nume port | Direcție | Descriere |
|---|---|---|
| digit 0..3 | in | Cifrele de afișat (format BCD, câte 4 biți per cifră) |
| clk | in | Semnalul de ceas pentru multiplexare temporală |
| cat | out | Linii catod pentru segmentele a-g |
| an | out | Semnale de selecție a cifrelor (anod comun) |
| dp | out | Punct zecimal al afișajului |

## 5.2 Integrarea modulelor

Modulele au fost interconectate într-un nivel superior denumit Top_Basys3. Sincronizarea sistemului este realizată cu semnalul comun de ceas de **100 MHz** (ACLK), iar resetarea globală este controlată prin semnalul ARESETn (activ pe '0').

```
entity Top_Basys3 is
    Port ( clk : in STD_LOGIC;
           reset_btn : in STD_LOGIC;
           rx : in STD_LOGIC;
           sw : in STD_LOGIC_VECTOR(0 downto 0);
           led : out STD_LOGIC_VECTOR(15 downto 0);
           cat : out STD_LOGIC_VECTOR(6 downto 0);
           an : out STD_LOGIC_VECTOR(3 downto 0);
           dp : out STD_LOGIC);
end Top_Basys3;
```

Listing 5.7: Entitatea modulului Top_Basys3

| Nume port | Direcție | Descriere |
|---|---|---|
| clk | in | Semnal de ceas de 100 MHz al plăcii Basys 3 |
| reset_btn | in | Buton de reset global al sistemului |
| rx | in | Linie de recepție UART de la Arduino |
| sw | in | Switch utilizat pentru control sau testare |
| led | out | LED-uri pentru afișarea stărilor interne |
| cat | out | Semnale pentru segmentele display-ului |
| an | out | Semnale de activare a cifrelor display-ului |
| dp | out | Punct zecimal al afișajului |

### 5.2.1 Respectarea protocolului AXI4-Stream

Conform standardului AXI4-Stream, semnalele de handshake sunt organizate astfel:

- **TDATA** și **TVALID** se propagă în sensul fluxului de date (de la Master către Slave);
- **TREADY** se propagă în sens invers, indicând disponibilitatea următorului modul de a accepta date;
- **Transferul de date** are loc doar când TVALID = '1' și TREADY = '1' simultan la frontul crescător al ceasului ACLK.

# Capitol 6

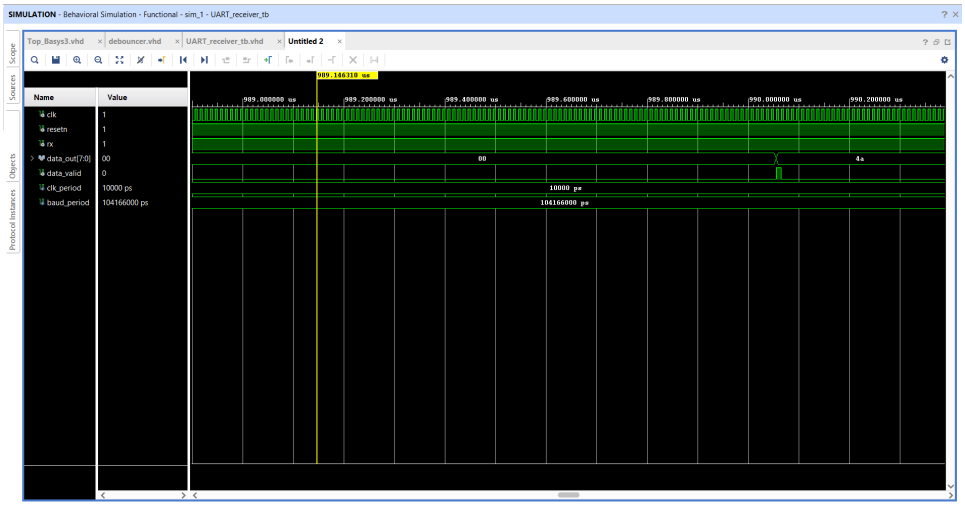
# Simulare și testare

Simulările au fost realizate în **Vivado Simulator (XSim)**, utilizând *testbench-uri* pentru fiecare modul principal:

- `UART_RECEIVER_tb.vhd` – verificarea recepției octeților seriali și validarea semnalului `data_valid`;
- `UART_TO_AXI_tb.vhd` – testarea conversiei pachetului serial în cuvânt AXI4-Stream;
- `Packet_Decoder_tb.vhd` – verificarea reconstrucției pachetului de 16 biți și a detecției corecte a secvenței;
- `Sliding_Window_Average_tb.vhd` – verificarea calculului mediei;
- `AXI_FIFO_tb.vhd` – testarea mecanismului AXI (TVALID/TREADY);

## 6.1 Simularea recepției UART

Modulul `UART_RECEIVER` a fost testat separat pentru a verifica funcționarea mecanismului de detecție a bitului de start, eșantionarea datelor și generarea semnalului `data_valid`. Am folosit o secvență de impulsuri care simulează transmisia serială a unui octet la 9600 bps.



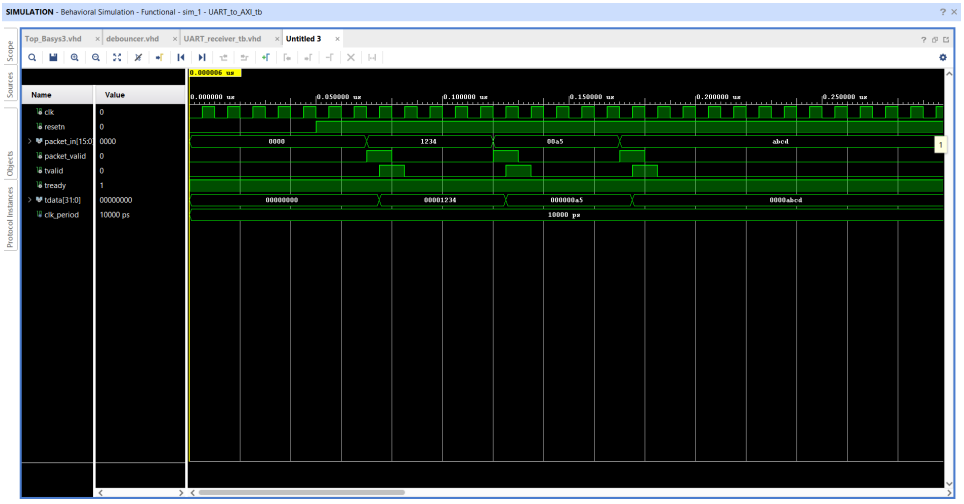

După finalizarea transmisiei, semnalul `data_valid` s-a activat pentru un singur ciclu de ceas, indicând recepția completă a octetului.

## 6.2 Testarea conversiei UART → AXI4-Stream

Modulul `UART_TO_AXI` a fost verificat cu o secvența de pachete UART de forma:

`0xAA [MSB] [LSB] 0x55`

Rezultatul a fost un flux AXI4-Stream de 32 de biți la ieșire, cu semnalele `TVALID` și `TREADY`.




## 6.3 Simularea modulului Packet_Decoder

`Packet_Decoder` are rolul de a reconstrui un pachet de 16 biți pe baza fluxului de octeți primit de la receptorul UART. Structura pachetului este:

`0xAA [MSB] [LSB] 0x55`

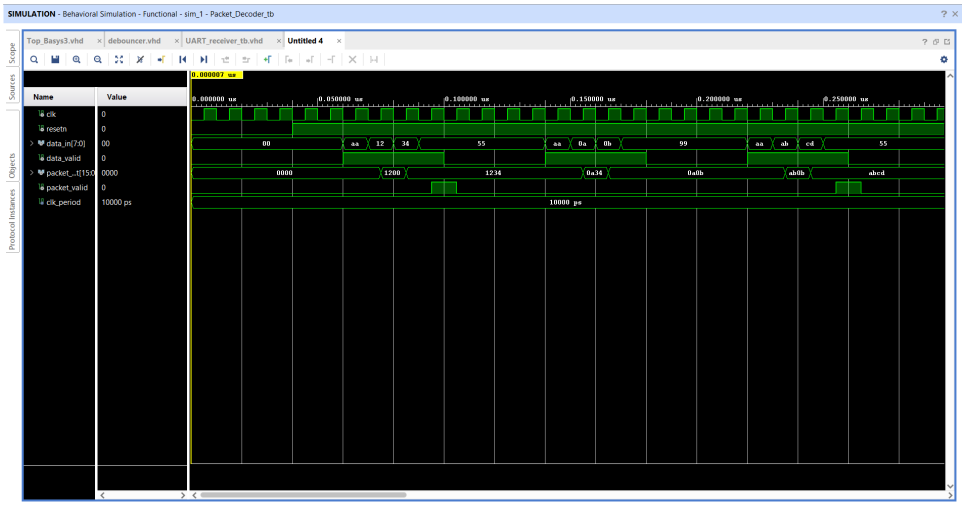
unde:

- `0xAA` reprezintă octetul de start,
- `MSB` este octetul superior al datelor,
- `LSB` este octetul inferior al datelor,
- `0x55` este octetul de stop.

Pentru testare, a fost creat un *testbench* dedicat (`Packet_Decoder_tb.vhd`), care introduce la intrare secvențe UART, cu semnalul `data_valid` activat la fiecare octet. S-au transmis atat pachete valide, precum și secvențe invalide pentru verificarea modulului.

În figura se observă succesiunea stărilor:

- după detectarea secvenței de start `0xAA`, modulul intră în starea `READ_MSB`;
- următorul octet este salvat ca `MSB`;
- apoi se citește `LSB`;
- dacă următorul octet este `0x55`, atunci `packet_valid` se activează pentru un singur ciclu.



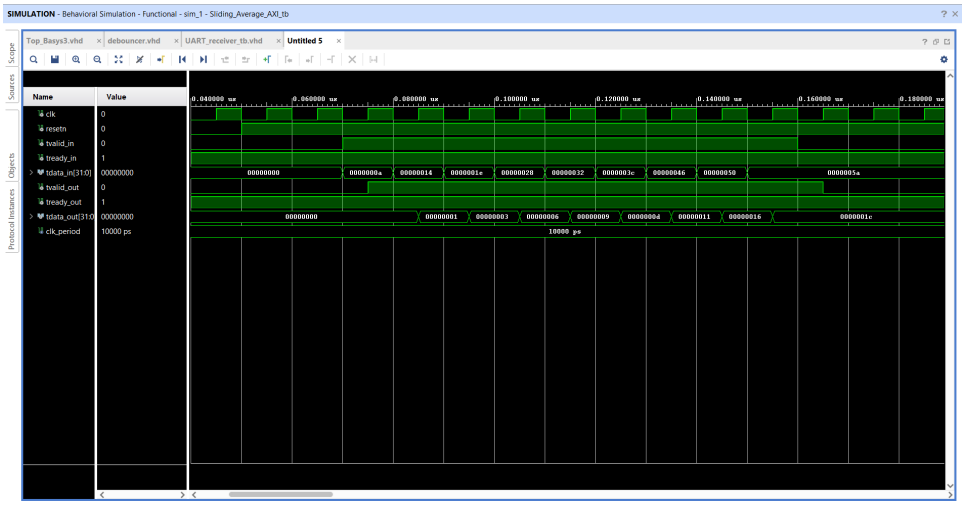

La ieșire, pachetul are forma:

$$\texttt{packet\_out} = \texttt{MSB} \,\|\, \texttt{LSB}$$

Semnalul `packet_valid` confirmă sincronizarea corectă a celor patru octeți.

## 6.4 Simularea filtrului de medie mobilă

Primul modul testat a fost **Sliding_Window_Average**, care implementează logica de filtrare a semnalului. În cadrul testbench-ului, au fost generate mai multe valori de intrare cu semnalul `s_axis_val_tvalid` activ. Pe măsură ce fereastra se umple, media calculată la ieșire crește gradual și se stabilizează.

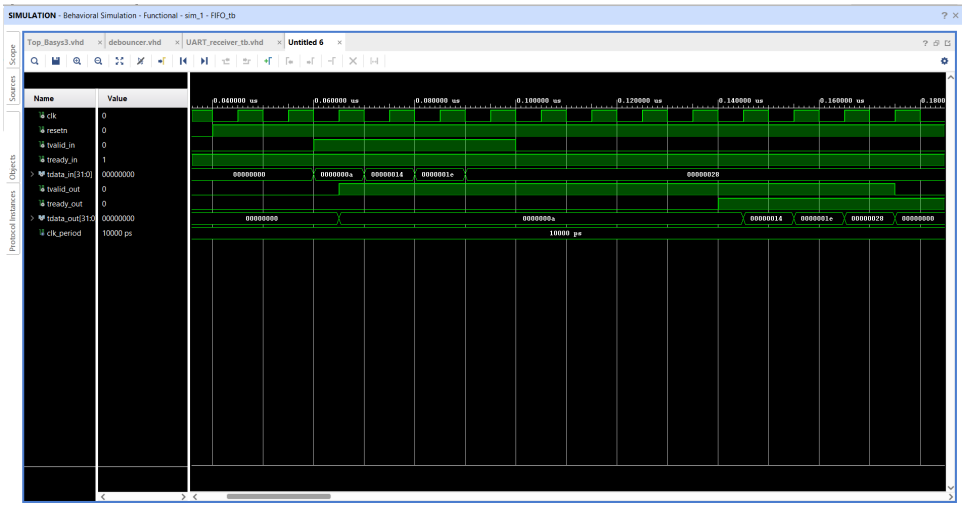



Pentru o fereastră de 5 valori consecutive (10, 20, 30, 40, 50), media rezultată este:

$$M = \frac{10 + 20 + 30 + 40 + 50}{5} = 30$$

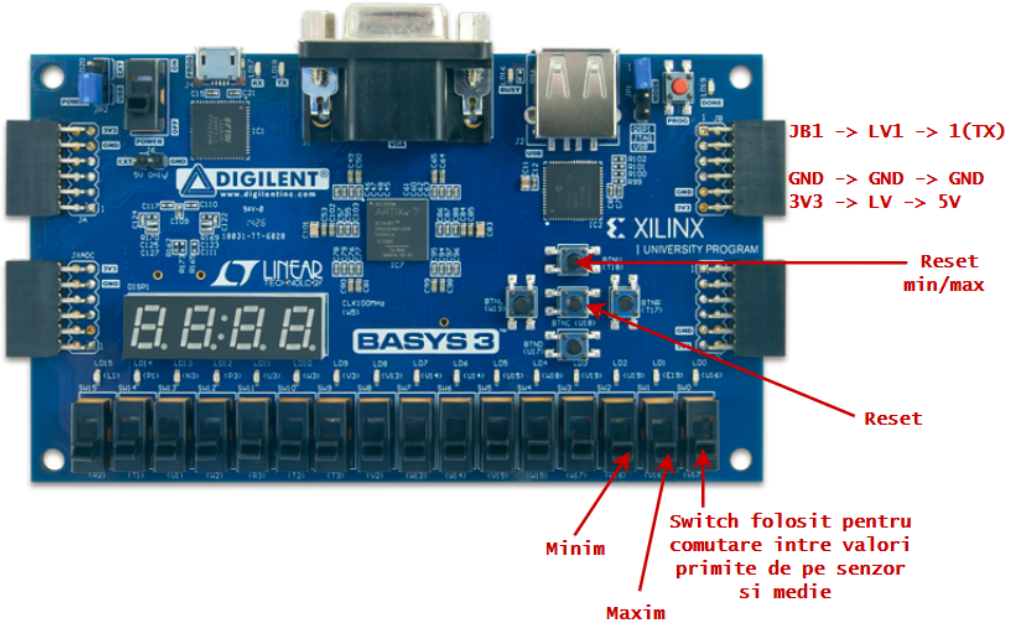
## 6.5 Simularea memoriei FIFO

FIFO stochează temporar datele primite până când semnalul `TREADY` devine activ, prevenind pierderea informației.




## 6.6 Testarea pe placa Basys3

După validarea funcționării fiecărui modul prin simulare, sistemul a fost implementat și testat pe placa de dezvoltare **Basys 3**. Testarea hardware a avut ca scop verificarea funcționării corecte a recepției UART, a procesării datelor și a afișării rezultatelor pe LED-uri și pe display-ul cu 7 segmente (SSD).



### 6.6.1 Recepția datelor de la Arduino

Placa Basys3 primește datele de temperatură de la Arduino prin interfața **UART**, la viteza de **9600 bps**. Datele sunt transmise sub formă de pachete, iar fiecare pachet valid este decodat și procesat în FPGA.

La recepția unui pachet corect:

- valoarea instantanee este salvată;
- valoarea este introdusă în FIFO;
- se actualizează valorile maxime și minime;
- filtrul de medie mobilă calculează o nouă valoare mediată.

### 6.6.2 Selectarea modului de afișare cu switch-uri

Modul de afișare pe SSD este controlat prin switch-urile plăcii, după cum urmează:

- **SW0 = 0**: afișarea valorii instantanee primite de la Arduino;
- **SW0 = 1**: afișarea valorii mediate (filtru de medie mobilă);
- **SW1 = 1**: afișarea valorii maxime (**MAX**);
- **SW2 = 1**: afișarea valorii minime (**MIN**).

Valorile **MAX** și **MIN** sunt calculate exclusiv pe baza valorilor brute primite de la Arduino, nu pe baza mediei.

### 6.6.3 Resetarea valorilor MAX și MIN

Pentru resetarea valorilor maxime și minime a fost utilizat un buton dedicat. La apăsarea acestuia:

- valoarea maximă este resetată la valoarea minimă posibilă;
- valoarea minimă este resetată la valoarea maximă posibilă;
- restul sistemului continuă să funcționeze normal.

Această funcționalitate permite reluarea măsurătorilor extreme fără resetarea completă a sistemului.

### 6.6.4 Afișarea mesajelor de eroare pe SSD

Display-ul cu 7 segmente este utilizat și pentru afișarea mesajelor de stare ale sistemului.

#### Eroare de pachet – mesaj `Err`

În cazul în care modulul `Packet_Decoder` detectează o eroare de pachet (secvență invalidă sau pachet incomplet), pe SSD este afișat mesajul:

```
Err
```

Acest mesaj are prioritate față de afișarea valorilor numerice.

**FIFO plin – mesaj `FULL`**

Dacă memoria FIFO devine plină, sistemul afișează pe SSD mesajul:

`FULL`

Acest mesaj avertizează utilizatorul asupra riscului de pierdere a datelor și indică necesitatea procesării acestora.

# Capitol 7

# Concluzii

## 7.1 Concluzii generale

Realizarea acestui proiect a ajutat in înțelegerea modului în care datele sunt prelucrate și afișate într-un sistem bazat pe FPGA. Prin implementarea folosirea interfetei de comunicare**AXI4-Stream**, am reușit să integrez conceptele teoretice studiate la curs cu partea practică de laborator. Totodată, am înțeles mai bine rolul semnalelor de control din protocolul AXI4-Stream (`TVALID`, `TREADY`, `TDATA`) și cum acestea asigură sincronizarea corectă între module.

## 7.2 Posibile îmbunătățiri și implementări viitoare

- **Filtrare în timp real:** în locul unei ferestre fixe, se poate implementa un filtru cu fereastră variabilă, ajustată automat în funcție de variația temperaturii. De exemplu, o fereastră mai mică atunci când temperatura se schimbă rapid și o fereastră mai mare cand temperaturile sunt stabile.
- **Logare și monitorizare externă:** valorile filtrate ar putea fi trimise către un PC sau o aplicație mobilă si o interfață grafică ar permite vizualizarea în timp real a semnalului.
- **Suport pentru mai mulți senzori:** FPGA-ul ar putea procesa în paralel temperaturile provenite de la mai multe dispozitive, fiecare cu propriul flux AXI.

# Bibliografie


[1] R. Woods, J. McAllister, G. Peterson, and A. G. Vladimirova, *FPGA-Based Implementation of Signal Processing Systems*, 2nd ed. Wiley, 2017, [Accesat Octombrie 2025]. [Online]. Available: https://rfsoc.mit.edu/6S965/_static/F24/textbooks/fpga_signal_processing.pdf

[2] R. Sass and A. G. Schmidt, *Embedded Systems Design with Platform FPGAs: Principles and Practices*. Morgan Kaufmann, 2010, [Accesat Octombrie 2025]. [Online]. Available: https://www.r-5.org/files/books/computers/hw-layers/hardware/digital-desigh/Ronald_Sass_Andrew_G_Schmidt-Embedded_Systems_Design_with_Platform_FPGAs-EN.pdf

[3] AMD Xilinx, *AXI4-Stream Interface Protocol Specification*, 2023, [Accesat Octombrie 2025]. [Online]. Available: https://docs.amd.com/r/en-US/pg256-sdfec-integrated-block/AXI4-Stream-Interface

[4] Xilinx Inc., *UART 16550 Compatible IP Core Product Guide (PG143)*, 2022, [Accesat Octombrie 2025]. [Online]. Available: https://docs.xilinx.com/r/en-US/pg143-axi-uart16550

[5] S. Brown and Z. Vranesic, *Fundamentals of Digital Logic with VHDL Design*, 3rd ed. McGraw-Hill, 2019, [Accesat Octombrie 2025]. [Online]. Available: https://dpvipracollege.ac.in/wp-content/uploads/2023/01/Fundamentals-of-Digital-Logic-with-VHDL-Design-3rd-edition.pdf

[6] A. D. Inc. (2019) Uart, hardware-communication protocol. [Accesat Octombrie 2025]. [Online]. Available: https://www.analog.com/en/resources/analog-dialogue/articles/uart-a-hardware-communication-protocol.html

[7] Advanced SoC Design Education Kit, *AXI UART and AXI4-Stream Peripherals*, Xilinx / ARM Education Kit, 2018, [Accesat Noiembrie 2025]. [Online]. Available: https://kolegite.com/EE_library/books_and_lectures/%D0%9C%D0%B8%D0%BA%D1%80%D0%BE%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D0%BE%D1%80%D0%BD%D0%B0%20%D1%81%D1%85%D0%B5%D0%BC%D0%BE%D1%82%D0%B5%D1%85%D0%BD%D0%B8%D0%BA%D0%B0/PVSAE_Lekcii/02_Advanced-System-on-Chip-Design-Education-Kit/08_AXI_UART_and_AXI4_Stream_Peripherals.pdf

[8] Digilent Inc., *Basys 3 Reference Manual*, 2024, [Accesat Noiembrie 2025]. [Online]. Available: https://digilent.com/reference/_media/basys3:basys3_rm.pdf