Universitatea Națională de Știință și Tehnologie POLITEHNICA
București
Facultatea de Electronică, Telecomunicații și Tehnologia Informației

Sistem automat de control a energiei electrice

Lucrare de licență

Prezentată ca cerință parțială pentru obținerea
titlului de *Inginer*
în domeniul *Electronică, Telecomunicații și Tehnologia Informației*
programul de studii *Microelectronică, optoelectronică și nanotehnologii*

Conducător științific
Ș.L. dr. ing. Voichița DRAGOMIR
ing. Alexandru ULMĂMEI

Absolvent
Ștefania-Anca CIOCOIU

Anul 2024

Universitatea "Politehnica" din București
Facultatea de Electronică, Telecomunicații și Tehnologia Informației
Program de studiu

Anexa 1

TEMA PROIECTULUI DE DIPLOMĂ

a studentului **CIOCOIU A. Ștefania-Anca, 441E**

1. Titlul temei: Sistem automat de control a energiei electrice

2. Descrierea temei și a contribuției personale a studentului (în afara părții de documentare):
Proiectul are drept scop implementarea unui sistem automat de control a energiei electrice generate de panourile solare și testarea funcționării acestuia. Controllerul va fi implementat într-un limbaj de descriere hardware (Verilog/System Verilog) și va fi sintetizat pe o placă de dezvoltare PYNQ Z1/Z2, cu FPGA Xilinx. Se va proiecta un suport pentru panoul solar pentru a permite rotirea acestuia prin intermediul unor motoare pas cu pas, in urma analizei pozitiei curente a panoului acestea vor fi comandate prin FPGA. Convertorul A/D primeste semnalul analogic de la fotorezistentele plasate pe panou si transmite mai departe codul digital catre FPGA. Contributia studentului: realizarea unui convertor analogic-digital, implementarea modulului digital, simularea și sintetizarea acestuia.

3. Discipline necesare pt. proiect:
CID, MC, CEF, CIA

4. Data înregistrării temei: 2024-02-02 05:15:32

Conducător(i) lucrare,
Ș.L. dr. ing. Voichița DRAGOMIR

ing. Andrei-Alexandru Ulmămei

Student,
CIOCOIU A. Ștefania-Anca

Director departament,

Decan,
Prof. dr. ing. Mihnea UDREA

Cod Validare: **7ce852ae2d**

Declarație de onestitate academică

Prin prezenta declar că lucrarea cu titlul "Sistem automat de control a energiei electrice", prezentată în cadrul Facultății de Electronică, Telecomunicații și Tehnologia Informației a Universității Naționale de Știință și Tehnologie POLITEHNICA București ca cerință parțială pentru obținerea titlului de Inginer în domeniul Inginerie Eelctronică și Telecomunicații/Calculatoare și Tehnologia Informației, programul de studii domeniul Microelectronică, Optoelectronică și Nanotehnologii este scrisă de mine și nu a mai fost prezentată niciodată la o facultate sau instituție de învățământ superior din țară sau străinătate.

Declar că toate sursele utilizate, inclusiv cele de pe Internet, sunt indicate în lucrare, ca referințe bibliografice. Fragmentele de text din alte surse, reproduse exact, chiar și în traducere proprie din altă limbă, sunt scrise între ghilimele și fac referință la sursă. Reformularea în cuvinte proprii a textelor scrise de către alți autori face referință la sursă. Înțeleg că plagiatul constituie infracțiune și se sancționează conform legilor în vigoare.

Declar că toate rezultatele simulărilor, experimentelor și măsurătorilor pe care le prezint ca fiind făcute de mine, precum și metodele prin care au fost obținute, sunt reale și provin din respectivele simulări, experimente și măsurători. Înțeleg că falsificarea datelor și rezultatelor constituie fraudă și se sancționează conform regulamentelor în vigoare.

București, 27 Iunie 2024

Absolvent: Ștefania-Anca CIOCOIU

.....

Table of Contents

List of Figures

List of Acronyms

ADC - Analog-to-Digital Converter
ARM - Advanced RISC Machine
AXI - Advanced eXtensible Interface
BRAM - Block Random Access Memory
CLB - Configurable Logic Block
CMOS - Complementary Metal-Oxide-Semiconductor
DDR - Double Data Rate
DFF - D Flip-Flop
FPGA - Field-Programmable Gate Array
FSM - Finite State Machine
HDL - Hardware Description Language
HLS - High-Level Synthesis
HSTL - High-Speed Transceiver Logic
IDE - Integrated Development Environment
IP - Intellectual Property
KSPS - Kilosamples per Second
LSB - Least Significant Bit
LUT - Look-Up Table
LVCMOS - Low Voltage Complementary Metal-Oxide-Semiconductor
LVDS - Low-Voltage Differential Signaling
MOSFET - Metal-Oxide-Semiconductor Field-Effect Transistor
NMOS - N-channel Metal-Oxide-Semiconductor
PMOS - P-channel Metal-Oxide-Semiconductor
MSPS - Mega Samples per Second
MUX - Multiplexer
UART - Universal Asynchronous Receiver-Transmitter
PC - Personal Computer PL - Programmable Logic
PS - Processing Sytem
PWM - Pulse-Width Modulation
RAM - Random Access Memory
RTL - Register Transfer Level
SOC - System on Chip

TTL - Transistor-Transistor Logic
VDC - Voltage Direct Current
VDDL - Low Voltage Digital Supply
VDDH - High Voltage Digital Supply
VHDL - VHSIC Hardware Description Language

Introducere

Conceptul de optimizare a panourilor solare cu ajutorul tehnologiei avansate câștigă tot mai multă popularitate în industria energiei solare, acesta contribuind la maximizarea producției de energie solară, extinderea duratei de viață a echipamentelor și promovarea unei economii sustenabile și ecologice.

Motivație

Motivația acestei teze are la bază dorința proprie de a explora atât partea hardware, cât și cea software a electronicii. Am dorit că tema să reprezinte un model practic ce poate fi utilizat în viața de zi cu zi. Energia solară reprezintă una dintre cele mai puternice surse de energie regenerabilă, iar utilizarea tehnologiei FPGA permite crearea de soluții hardware personalizate în ceea ce privește eficiența panourilor solare. Avantajul aplicației este dat de viteza conversiei energetice în timp real, datorită convertorului analog-digital integrat cu o viteză de conversie de până la 1 MSPS, fiind potrivit pentru aplicații ce necesită achiziții de date în timp real și cu o rezoluție ridicată.

Aplicabilitate

Domeniile de aplicabilitate ale sistemului prezentat în această lucrare sunt variate și acoperă o gamă largă de utilizări, de la sisteme de alimentare pentru locuințe și dispozitive portabile la stații de încărcare solară, ferme solare și chiar vehicule electrice solare. Acest sistem a fost conceput inițial pentru a fi implementat într-un sistem de alimentare pentru locuințe, unde vă funcționa că sursă de energie pentru diverse dispozitive, inclusiv un ventilator solar, contribuind astfel la reducerea dependenței de sursele de energie tradiționale și la promovarea sustenabilității.

Industria optimizării panourilor solare

În prezent, există mai multe companii și organizații care utilizează tehnologii avansate pentru optimizarea eficienței panourilor solare prin automatizare. Aceste tehnologii pot include utilizarea algoritmilor de urmărire a soarelui, sisteme de monitorizare și control, precum și integrarea de soluții IoT pentru gestionarea eficientă a energiei solare. Iată câteva exemple de companii și organizații implicate în acest domeniu:

SunPower: SunPower este un producător de top de panouri solare și sisteme solare integrate. Compania utilizează tehnologii avansate pentru a îmbunătăți eficiența panourilor solare și a optimiza producția de energie electrică. SunPower integrează sisteme de monitorizare și control pentru a asigura performanța optimă a sistemelor solare instalate.

SolarEdge Technologies: SolarEdge este cunoscută pentru inovarea în domeniul optimizării performanței panourilor solare. Compania dezvoltă și produce sisteme avansate de optimizare a puterii solare, inclusiv convertoare de putere cu tehnologie optimizată pentru a maximiza producția de energie electrică din panourile solare.

Enphase Energy: Enphase Energy este specializată în sisteme de micro-invertoare pentru panouri solare. Aceste inovatoare micro-invertoare sunt concepute pentru a îmbunătăți eficiența, fiabilitatea și performanța sistemelor solare, prin monitorizare avansată și control individual al fiecărei unități de panou solar.

ABB: ABB este un lider în tehnologia de automatizare industrială și electrică. Compania dezvoltă soluții avansate pentru integrarea energiei solare în rețelele electrice, inclusiv tehnologii de control și monitorizare care optimizează eficiența și fiabilitatea sistemelor solare.

SMA Solar Technology: SMA este un alt jucător important în domeniul convertoarelor de putere pentru sisteme solare. Compania oferă soluții avansate pentru monitorizarea și controlul sistemelor fotovoltaice, contribuind la maximizarea performanței panourilor solare și a întregului sistem de energie solară.

Obiective

Obiectivele principale ale acestei lucrări includ:

- Proiectarea și implementarea unui sistem de alimentare solară eficient și fiabil.
- Optimizarea performanței sistemului pentru a asigura utilizarea maximă a

energiei solare disponibile.

- Asigurarea compatibilității și siguranței sistemului în utilizarea sa practică.

Capitolul 1

Lucrări în literatură (State of the art)

În literatura de specialitate, au fost dezvoltate numeroase implementări ale sistemului automat de control al energiei electrice utilizând FPGA-uri. În lucrarea [13], se implementează un modul de top extrem de complex, bazat pe un modul FSM (Finite State Machine), alături de convertoare analog-digital, comparatoare, registre, numărători verticali și orizontali, numărător maxim și un modul de servomotor care include PWM și un modul de divizare a ceasului. Implementarea acestui modul se realizează utilizând placa Basys3, oferind un avantaj major în ceea ce privește precizia poziționării panoului solar, datorită numărătorilor care decid poziția optimă și exactă.

În lucrarea [14], controlul panoului solar a fost realizat folosind un microcontroler și sisteme fuzzy, care controlează servomotoarele. Acest proiect se remarcă prin eficiența ridicată și consumul redus de energie.

În lucrarea [15], proiectul este implementat utilizând placa Altera Quartus-II. Aceasta nu dispune de un XADC integrat pe placă, motiv pentru care a fost construit un ADC de la zero. Tensiunea de intrare pentru ADC a fost preluată de la un circuit format din fotorezistențe și rezistențe atașate pe panoul solar.

În această lucrare, schema principală și logica ce controlează sistemul au fost realizate având ca inspirație conceptele enunțate în [14], [15], si [13] la care au fost adăugate contribuții unice și personale, cu scopul de a realiza o implementare hardware în FPGA.

Capitolul 2

Aspecte teoretice

2.1 Optimizarea sistemelor fotovoltaice

Optimizarea sistemelor fotovoltaice prin utilizarea FPGA-urilor reprezintă o abordare inovatoare și eficientă în domeniul energiei regenerabile. Prin integrarea unui FPGA în sistemul de control al panourilor solare, avem posibilitatea de a implementa un set diversificat de funcționalități avansate, adaptabile și optimizate. Alegerea FPGA-ului este motivată de capacitatea sa de a oferi flexibilitate și putere computațională remarcabilă, permițând implementarea algoritmilor complexi și a logicii de control într-un mediu programabil și reconfigurabil. Utilizarea unui Finite State Machine (FSM) pentru controlul panoului solar aduce beneficii semnificative în eficiența și performanța sistemului. Prin FSM, putem gestiona în mod inteligent poziționarea și orientarea panoului solar în funcție de parametrii de mediu, precum poziția soarelui, condițiile meteorologice și cerințele de energie. Astfel, optimizăm producția de energie solara și creștem eficiența sistemului în ansamblu, contribuind la utilizarea durabilă și responsabilă a resurselor regenerabile.[16]

2.2 Descrierea instrumentelor software utilizate

2.2.1 Xilinix Vivado 2021.2

Xilinx Vivado este un mediu de proiectare integrat (IDE) dezvoltat de compania Xilinx pentru proiectarea și implementarea circuitelor digitale pe dispozitive FPGA. Vivado permite realizarea modulelor digitale în limbajele Verilog sau VHDL, însă acesta dispune și de o unealtă numită IP Integrator, prin care utilizatorul poate interconecta modulele realizate manual și modulele create de Xilinx, în vederea realizării proiectelor digitale complexe. IP Integrator este un mediu grafic și interactiv care generează automat interfețele grafice ale modulelor, precum procesoare,

convertoare sau registre. [17]

Procesul de programare a FPGA-ului include mai mulți pași:

- Scrierea codului HDL în limbajul de descriere hardware VHDL sau Verilog.
- Sinteza codului folosind secțiunea RTL ANALYSES.
- Plasarea și rutarea codului RTL pe FPGA.
- Generarea fișierului binar, denumit Bitstream.
- Programarea FPGA-ului: încărcarea fișierului Bitstream în FPGA.

2.2.2 Vitis HLS 2021.2

Vitis HLS este un instrument software de proiectare dezvoltat de Xilinix cu scopul de converti codul scris in limabje de programare de nivel inalt, precum C/C++ sau OpenCL in cod RTL pentru a putea fi apoi implementat pe FPGA.[18]

Procesul de programare a FPGA-ului include mai mulți pași:

- Scrierea codului HDL în limbajul de descriere hardware VHDL sau Verilog.
- Sinteza codului folosind secțiunea RTL ANALYSES.
- Plasarea și rutarea codului RTL pe FPGA.
- Generarea fișierului binar, denumit Bitstream.
- Programarea FPGA-ului: încărcarea fișierului Bitstream în FPGA.

2.2.3 Vitis HLS 2021.2

Vitis HLS este un instrument software de proiectare dezvoltat de Xilinix cu scopul de converti codul scris în limabje de programare de nivel înalt, precum C/C++ sau OpenCL în cod RTL pentru a putea fi apoi implementat pe FPGA.[18] Procesul de programare a FPGA-ului include mai mulți pași:

- Compilarea, simularea și depanarea algoritmului C/C++.

- Analiza și optimizarea algoritmului C/C++.
- Sinteza algoritmului într-o descriere RTL.
- Verificarea implementării RTL prin co-simulare.
- Exportarea modelului RTL pentru utilizare în instrumente de proiectare hardware.

În cadrul Vitis HLS, sunt introduse noi tipuri de date utile pentru lucrul cu precizie arbitrară și pentru a realiza o trecere de la algoritmii de nivel înalt la o implementare mai apropiată de nivelul RTL. Aceste tipuri de date includ *ap_int* și *ap_uint*, care permit specificarea precisă a dimensiunii în biți a variabilelor utilizate în funcțiile definite în HLS. De exemplu, o variabilă de tip $ap\_uint < 32 >$ vă avea o dimensiune de 32 de biți (sau 4 octeți) și vă putea stoca doar valori naturale (întregi pozitive).

În plus, în HLS se utilizează tipul de date $hls :: stream$ pentru implementarea fluxului de date. Acest tip de date implementează o memorie tip FIFO în hardware, realizată în general cu registre de deplasare. Stream-urile de date sunt extrem de utile în aplicații precum prelucrarea de imagini, unde stocarea completă a acestora în memoria dedicată a FPGA-ului nu este eficientă.

Astfel, utilizarea acestor tipuri de date în Vitis HLS permite programatorilor să specifice și să optimizeze implementările hardware pentru a îndeplini cerințele de performanță și de resurse ale aplicațiilor lor.[18]

2.3 Placa de dezvoltare

Placa de dezvoltare utilizată în aceste proiecte este PYNQ-Z2 și face parte din seria Xilinx Zynq-7000 SoC. Aceasta este echipată cu un FPGA Xilinx Zynq-7000 SoC, care integrează un procesor ARM Cortex-A9 dual-core împreună cu o logică FPGA programabilă. Configurația descrisă oferă plăcii versatilitate și o putere de procesare remarcabilă, fiind potrivită pentru o varietate de aplicații din domeniul embedded. PYNQ-Z2 este compatibilă cu framework-ul PYNQ (Python Productivity for ZYNQ), care permite dezvoltatorilor să utilizeze Python și biblioteci specifice pentru a accelera dezvoltarea aplicațiilor FPGA-based. Acest aspect face placa accesibilă și pentru cei care sunt mai familiarizați cu programarea în Python decât cu programarea FPGA tradițională.[1]

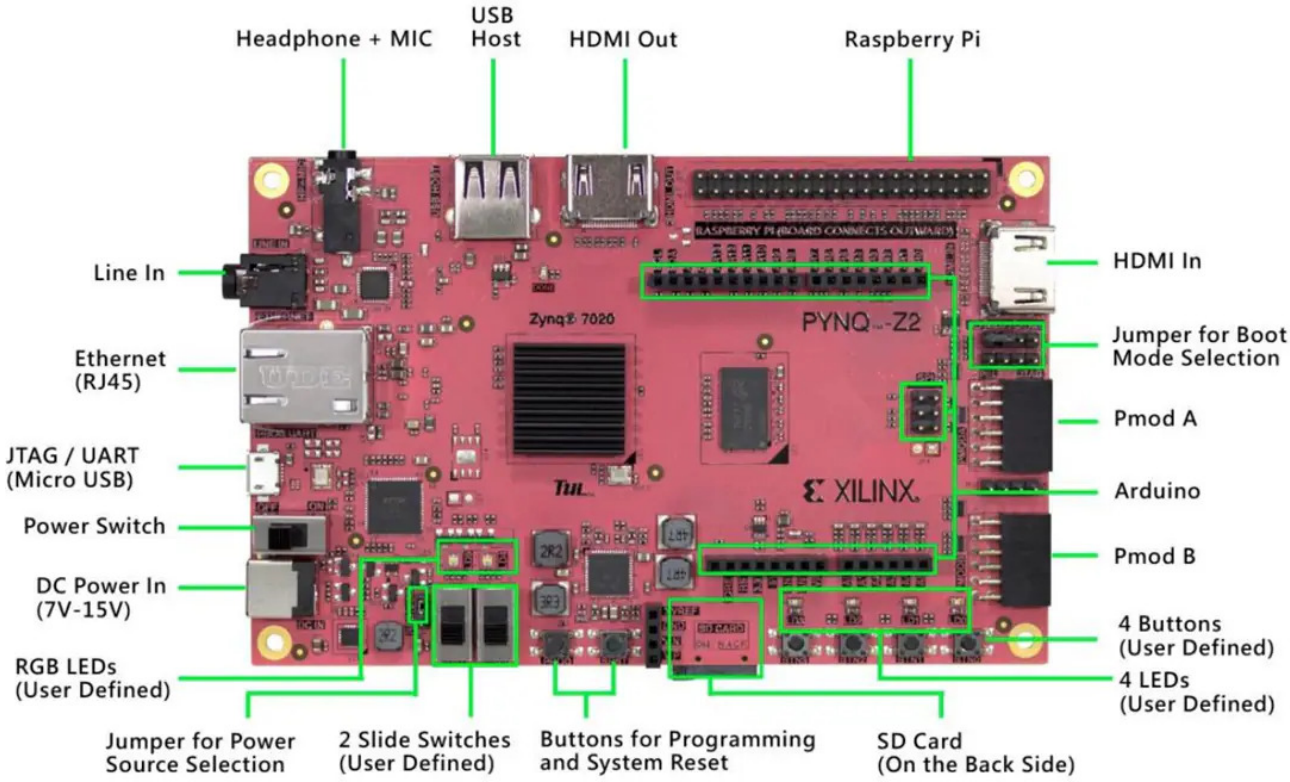


Figura 2.1: Pinout PYNQ-Z2 [1]

2.4 Descrierea componentelor hardware ale plăcii de dezvoltare

Un FPGA este format din:

- Blocuri logice configurabile
- Matrice de interconectare
- Memorii RAM
- Blocuri I/O
- Transcievere

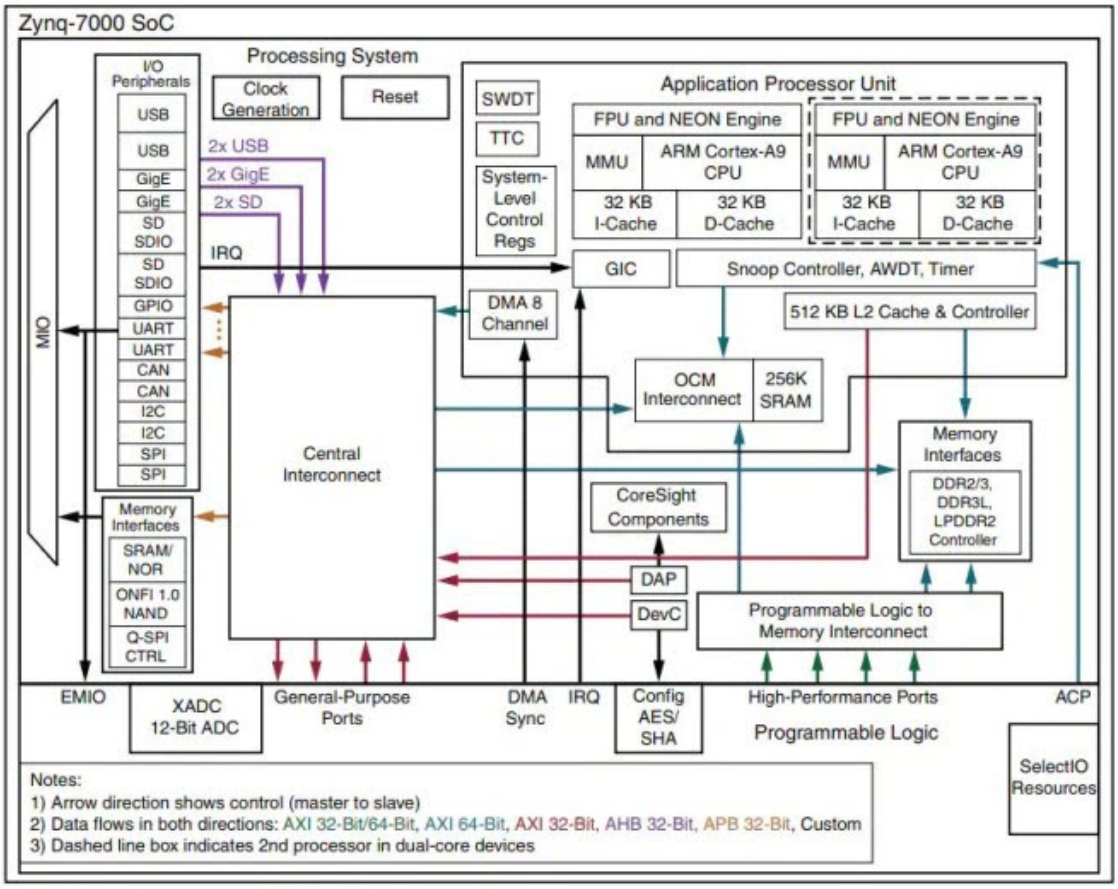


Figura 2.2: Arhitectură internă PYNQ-Z2[2]

2.4.1 Blocuri logice programabile(Configurable Logic Blocks)

Blocurile logice programabile sunt blocurile fundamentale ale unui FPGA, implementarea funcțiilor logice fiind realizată în CLB. Acestea sunt alcătuite dintr-o matrice de LUT-uri, DFF-uri și MUX-uri. LUT-urile pot fi programate pentru a implementa orice funcție logică, iar DFF-urile sunt utilizate pentru a produce o versiune întârziată a ieșirii LUT-ului. MUX-ul selectează una dintre cele două intrări: ieșirea directă din LUT și versiunea întârziată din DFF.[19] [3]

Fiecare funcție logică configurată cu maxim patru întrări poate folosi un singur LUT. În cazul în care funcția logică prezintă mai mult de șase intrări, de exemplu

opt intrări, va trebui să utilizăm încă un LUT.[19] [3]

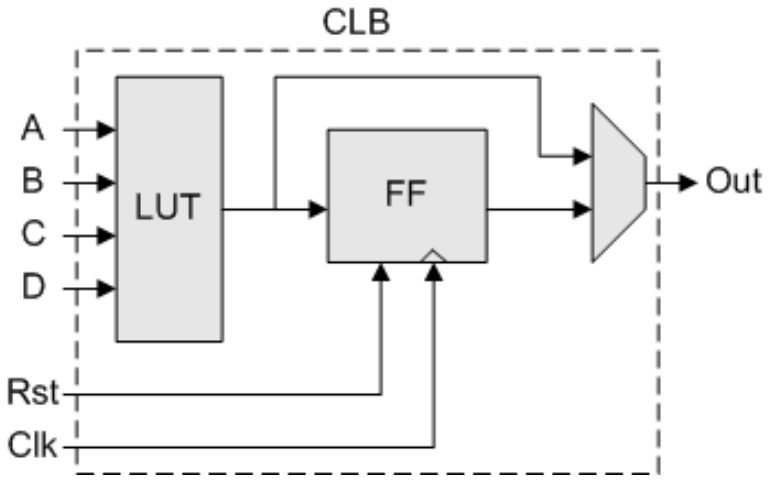


Figura 2.3: Arhitectura interna a unui BLC [3]

2.4.2 Matrice de interconectare (Interconnection Matrix)

Matricea de interconexiuni prezintă o topologie de tip grilă (grid-based), în care interconexiunile sunt organizate într-o rețea regulată de linii orizontale și verticale. Cu cât matricea de interconectare este mai largă și mai complexă, aceasta poate permite implementarea unor circuite mai mari și mai complexe în FPGA. Scopul său este acela de realiza transferul datele printre CLB-uri.[20] [21]

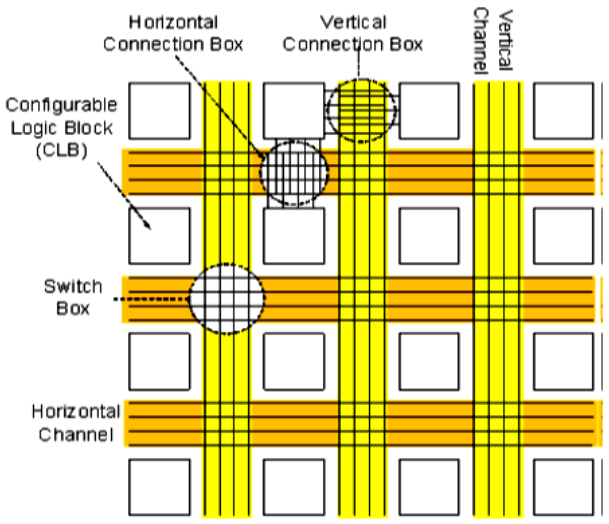


Figura 2.4: Configurație Dual Port BRAM [4]

2.4.3 Memorii RAM

FPGA-urile pot include și blocuri de memorie RAM configurabile, care pot fi utilizate pentru a stoca date temporare sau pentru a implementa memorie pentru aplicații specifice.[4] BRAM este o resursă importantă în FPGA-uri, oferind memorie cu acces rapid pentru stocarea și manipularea datelor în cadrul proiectelor digitale. Conceptul de single și dual port BRAM configuration se referă la modul în care această memorie este organizată și utilizată în cadrul unui FPGA.[4]

- Configurație Single Port BRAM: Într-o configurație de tip single port BRAM, memoria poate fi accesată (citită sau scrisă) printr-un singur port. Acest lucru înseamnă că, în orice moment, o singură operație de citire sau scriere poate avea loc în memorie. Această configurație este adecvată pentru aplicații în care este necesar un singur flux de date sau acces simultan la memoria BRAM.[4]

- Configurație Dual Port BRAM: Într-o configurație de tip dual port BRAM, memoria este echipată cu două porturi separate care permit accesul simultan la aceeași memorie din două surse diferite. Acest lucru înseamnă că, în același ciclu de ceas, două operații de citire sau scriere pot avea loc simultan în memorie, fiecare accesând o zonă specifică a acesteia. Dual port BRAM

este ideal pentru aplicații care necesită acces concurent la memorie din două fluxuri de date independente sau pentru implementarea de structuri de date complexe, cum ar fi cozi FIFO (First-In, First-Out) sau buffer-e circulare.[4]

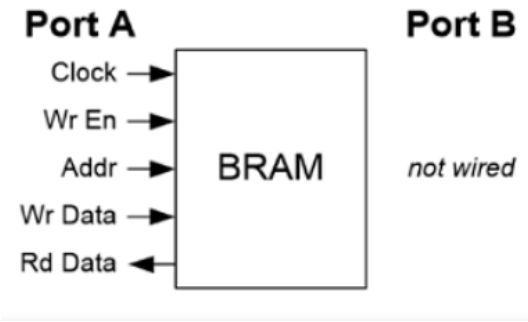


Figura 2.5: Configurație Single Port BRAM [4]

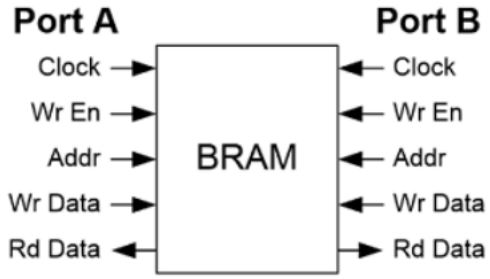


Figura 2.6: Configurație Dual Port BRAM [4]

În ambele configurații, BRAM-ul poate fi configurat în mod flexibil pentru a satisface nevoile specifice ale aplicației, inclusiv dimensiunea, lățimea și modul de organizare a datelor. Utilizarea corectă a configurațiilor single și dual port BRAM poate contribui la optimizarea performanței și eficienței în proiectele FPGA.[4]

2.4.4 Blocuri I/O

Blocurile I/O sunt circuite specializate cu programarea intrărilor și ieșirilor. Acestea pot include diferite tipuri de interfețe electrice, cum ar fi TTL, LVCMOS,

LVDS, HSTL. Aceste interfețe permit conectarea și comunicarea cu o varietate de dispozitive externe. Unele FPGA-uri includ blocuri I/O cu caracteristici speciale, cum ar fi suport pentru alimentare separată pentru interfețe sensibile la zgomot, funcții de protecție împotriva supratensiunilor și subtensiunilor. Performanța și lățimea benzii a blocurilor I/O variază în funcție de arhitectura FPGA-ului și de specificațiile dispozitivului. Aceste caracteristici sunt importante în proiectarea sistemelor în timp real sau a altor aplicații care necesită interacțiune rapidă cu mediul extern. [19]

2.4.5 Transcievere

Transceiver-urile sunt componente esențiale în FPGA-uri și alte dispozitive de semnal digital, care permit transmiterea și recepția datelor între dispozitivul FPGA și alte dispozitive sau sisteme externe. Acestea preiau datele digitale din interiorul FPGA-ului, le convertește în semnale electrice sau optice pentru transmisie și le trimite către dispozitivul extern. De asemenea, ele pot recepționa semnalele de la dispozitivul extern, le convertește în format digital și le trimite înapoi către FPGA. Această funcționalitate bidirecțională face transceiver-ele ideale pentru comunicația în ambele direcții între FPGA-uri sau între FPGA-uri și alte componente. Transceiver-ele pot avea o serie de caracteristici și specificații, cum ar fi viteza de transmitere a datelor, lățimea de bandă, nivelurile de tensiune suportate, protocolurile de comunicație compatibile. Aceste caracteristici sunt importante în proiectarea sistemelor în care comunicarea rapidă și eficientă a datelor este importantă. [22]

2.5 Convertorul analog-digital: XADC Dual 12-Bit 1 MSPS

- XADC-ul este integrat în FPGA-ul Zynq-7000 și oferă 17 canale de intrare analogică, cu o rezoluție de 12 biți. Aceste canale pot fi configurate pentru a măsura tensiuni analogice într-un interval cuprins între 0 și 1 volți, sau între -1 și 1 volți cu un singur circuit de referință, sau între alte limite utilizând două circuite de referință. De asemenea, XADC-ul poate efectua măsurători de temperatură.[5]
- XADC-ul este accesibil prin intermediul interfeței Xilinx AXI (Advanced eXtensible Interface), permițând comunicarea cu alte blocuri din FPGA prin intermediul unui bus de date. Acest lucru facilitează integrarea XADC-ului în designul dvs. FPGA și permite citirea datelor măsurate de către alte module

sau componente.[5]

- XADC-ul poate fi configurat și controlat folosind interfețe software precum Xilinx Vivado sau PYNQ Python. Aceasta permite setarea parametrilor de măsurare, cum ar fi intervalul de tensiune, modul de operare și ratele de eșantionare, și citirea datelor măsurate pentru a fi utilizate în aplicații. [5]

În modul unipolar, cu o gamă nominală de intrare analogică de la 0V la 1V, ADC-urile produc un cod la scară completă de FFFh (12 biți) atunci când intrarea este de 1V. Prin urmare, dacă un semnal de intrare analogică are o valoare de 200 mV, acesta va produce și scoate un cod proporțional cu această valoare analogică. Pentru a determina codul specific generat pentru o intrare de 200 mV, putem utiliza raportul dintre tensiunea maximă de intrare și numărul maxim de biți (12 biți), astfel: [5]

Pentru calculul codului pentru o intrare analogică de 200 mV:

$$Cod_{200mV} = \frac{200mV}{1V} \times FFFh$$

$$Cod_{200mV} = \frac{200}{1000} \times FFFh$$

$$Cod_{200mV} = \frac{1}{5} \times FFFh$$

$$Cod_{200mV} = \frac{4095}{5}$$

$$Cod_{200mV} = 819.0$$

Calculul LSB-ului LSB (Least Significant Bit) în contextul unui XADC (Xilinx Analog-to-Digital Converter) se referă la cea mai mică unitate de măsură a ieșirii ADC-ului, adică schimbarea minimă în tensiune pe care ADC-ul o poate detecta. XADC-ul Xilinx are o rezoluție de 12 biți, ceea ce înseamnă că poate distinge 4096 (2^{12}) nivele diferite de tensiune. LSB este dat de intervalul de tensiune împărțit la numărul de nivele (2^{12}).[5]

$$LSB = \frac{Intervalul\_de\_tensiune}{2^{Rezoluție}}$$

Pentru o referință de 1V și o rezoluție de 12 biți:

$$LSB = \frac{1V}{2^{12}} = \frac{1V}{4096} \approx 0.00024414V \quad sau \quad 244.14\mu V$$

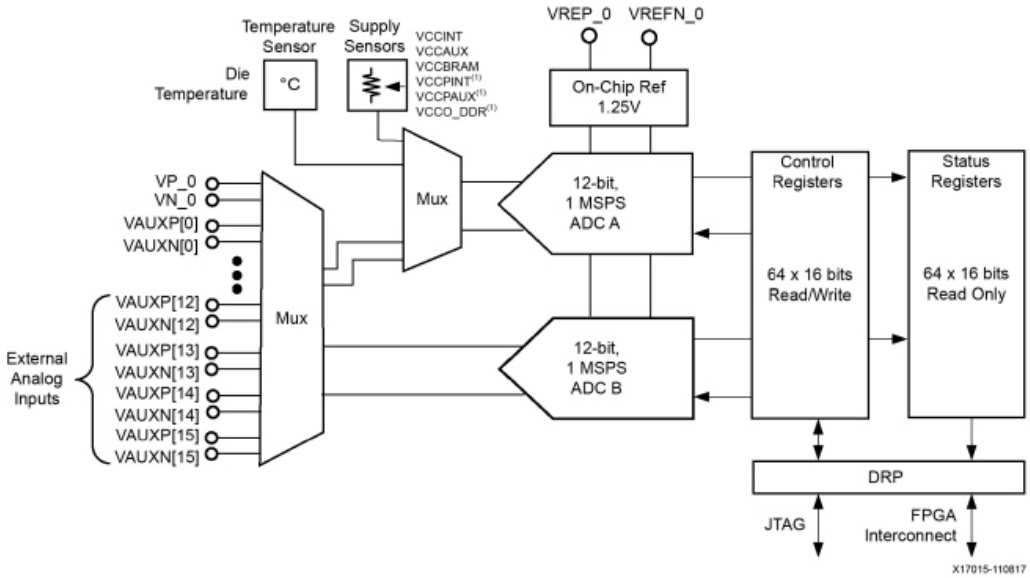


Figura 2.7: Diagramă bloc XADC Zynq-7000 [5]

2.6 Protocolul de comunicare serială UART

UART (Universal Asynchronous Receiver-Transmitter) este un protocol de comunicare serială asincronă care permite transferul de date între dispozitive. Este utilizat frecvent în sistemele integrate și în calculatoare pentru transmiterea și recepționarea datelor în mod serial. [23]

Utilizări ale protocolului de comunicare UART:

- IoT: Este utilizat în sisteme înglobate pentru comunicarea între microcontrolere, senzori și alte dispozitive periferice.
- Conectivitate PC: UART este folosit pentru a permite comunicarea între

PC-uri și dispozitive periferice, cum ar fi modulele Bluetooth, modemele, și alte dispozitive seriale.

- Debugging și diagnosticare: UART este adesea utilizat pentru a trimite mesaje de debug și pentru a monitoriza starea dispozitivelor în timpul dezvoltării și testării software.

Configurare hardware: pentru a implementa UART, este necesară configurarea adecvată a liniei TX și RX, utilizând adesea nivelele de tensiune specificate (de exemplu, TTL sau RS-232).

Software: programarea UART necesită adesea configurarea parametrilor cum ar fi viteza de transmisie (baud rate), numărul de biți de date, paritatea și numărul de biți de stop. Aceste setări trebuie să fie consistente între dispozitivele de comunicație pentru a asigura o comunicare corectă.[23]

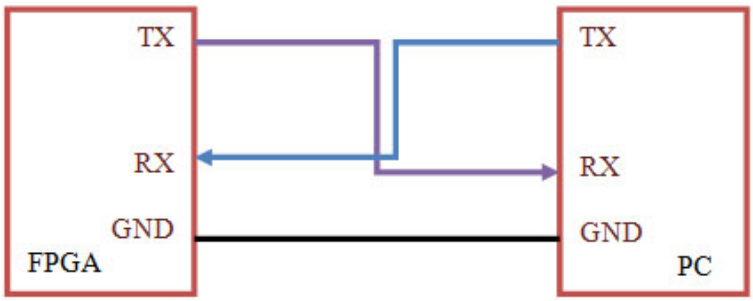


Figura 2.8: Interfata UART - PC [6]

2.6.1 Terminalul Putty

PuTTY este un emulator de terminal open-source care poate fi utilizat pentru a stabili conexiuni cu diverse protocoale de rețea, inclusiv serial (folosit pentru UART), SSH, Telnet și altele. În contextul UART, PuTTY este folosit pentru a afișa datele recepționate de la dispozitivul care trimite date prin portul serial. [24]

Viteza de baud (baud rate) se măsoară în unitatea de măsură "baud". Termenul "baud" este folosit pentru a descrie numărul de semnale sau schimbări de stare ale unei linii de comunicație într-o secundă. În general, un semnal de baud reprezintă o tranziție de la un nivel de tensiune la altul pe linia de comunicație.[25]

Exemple de valori comune de baud rate:

- 9600 baud: Foarte des întâlnit în aplicațiile de comunicație serială, inclusiv pentru conectarea dispozitivelor la PC-uri sau la echipamente de rețea.
- 115200 baud: Utilizat pentru comunicarea rapidă între dispozitive care necesită o rată de transfer mai mare, cum ar fi în cazul consolelor de debug sau în controlul automatizărilor industriale.
- 57600 baud: O altă valoare comună, folosită în unele aplicații industriale sau în controlul echipamentelor de laborator.

2.7 Modelul matematic Finite State Machine(FSM)

Un FSM, cunoscut și sub denumirea de automat finit, este un model matematic și conceptual al unui sistem de calcul care poate fi într-un număr finit de stări discrete la un moment dat. Acesta poate trece de la o stare la alta în funcție de intrările primite și starea curentă a mașinii.[7]

Caracteristicile unui automat finit:

- Stări: FSM-ul poate fi într-o singură stare la un moment dat. Aceste stări reprezintă condițiile interne ale sistemului și reflectă comportamentul său la un moment dat.
- Tranziții: Tranziția reprezintă schimbarea de la o stare la alta în funcție de intrările primite și starea curentă a mașinii. Aceste tranziții sunt definite de o funcție de tranziție care mapează o stare și o intrare la o altă stare.
- Intrări: FSM-ul primește intrări din mediul său și poate reacționa la aceste intrări prin schimbarea stării sale. Intrările pot influența comportamentul și tranzițiile mașinii.
- Ieșiri: În unele cazuri, FSM-ul poate produce și ieșiri pe baza stării și a intrărilor sale. Aceste ieșiri pot influența mediul exterior sau pot fi utilizate pentru a comunica informații.

Automatele Mealy și Moore sunt două tipuri de Finite State Machines (FSM-uri), modele matematice utilizate pentru a descrie și analiza comportamentul sistemelor discrete. Aceste două tipuri de automate diferă în modul în care produc ieșiri în funcție de starea lor curentă și de intrările primite.[7]

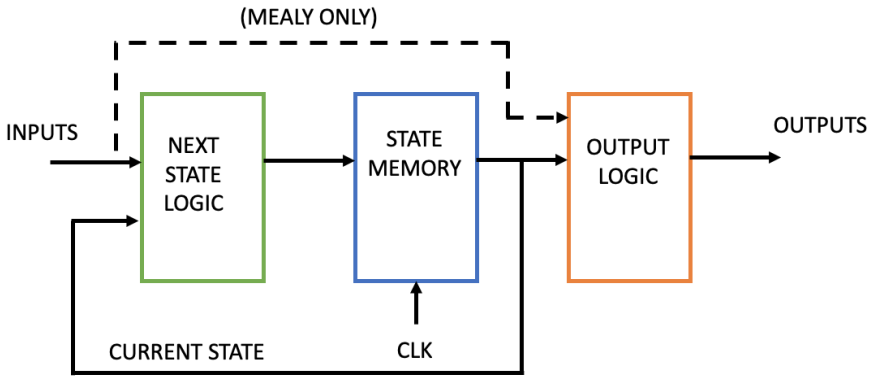


Figura 2.9: Diagrama bloc FSM [7]

2.7.1 Automatul Mealy

Într-un automat Mealy, ieșirile sunt asociate cu tranzitiile între stările automatei și cu intrările primite în acel moment. Astfel, ieșirile sunt generate nu numai pe baza stării curente, ci și pe baza intrărilor primite în momentul respectiv. Producerea ieșirilor este determinată de o funcție de ieșire, care mapează o pereche formată din starea curentă și intrarea curentă la o ieșire specifică. Avantajul principal al automatelor Mealy este că pot fi mai compacte decât echivalentele lor Moore, deoarece unele informații pot fi codificate în intrări și ieșiri, reducând astfel numărul de stări necesare. Un exemplu clasic de aplicație a automatelor Mealy este în designul de circuite secvențiale și în protocoalele de comunicație.[7]

2.7.2 Automatul Moore

Într-un automat Moore, ieșirile sunt asociate cu stările automatei și nu depind direct de intrările primite în momentul respectiv. Ieșirile sunt generate numai pe baza stării curente, fără a lua în considerare intrările curente. Producerea ieșirilor este determinată de o funcție de ieșire, care mapează fiecare stare a automatei la o ieșire specifică. Avantajul principal al automatelor Moore este că sunt mai ușor de proiectat și de înțeles, deoarece ieșirile sunt determinate exclusiv de starea curentă a automatei. Un exemplu obișnuit de aplicație a automatelor Moore este în sistemele de control și în modelarea comportamentului secvențial simplu.[7]

2.8 Descrierea componentelor hardware adiacente utilizate în proiect

2.8.1 Panoul solar 12VDC

Un panou solar funcționează pe baza efectului fotovoltaic, în care celulele solare din panou convertește energia luminii solare în electricitate. Această conversie are loc prin interacțiunea fotonilor de lumină solară cu semiconductorii din celulele solare, generând astfel o diferență de potențial și producând curent electric.[26]Panourile solare pot avea diferite caracteristici tehnice, cum ar fi puterea nominală, tensiunea de ieșire, curentul de ieșire, eficiența conversiei, dimensiunile fizice și greutatea. Un panou solar cu o tensiune nominală de 12V DC este proiectat să furnizeze o tensiune stabilă de 12 volți în condiții nominale de lumină solară.[26]

Figura 2.10: Panou solar [8]

2.8.2 Servomotoare de rotație continuă bidirecțională

Servomotoarele de rotație continuă bidirsecțională sunt dispozitive electromecanice utilizate pentru controlul rotației unui obiect în ambele direcții, cu o viteză variabilă și control precis. Servomotoarele de rotație continuă bidirecțională funcționează în principal pe baza unui servomecanism care convertește semnalele de control în mișcarea rotațională a unui arbore de ieșire. Ele sunt similare cu servomotoarele standard, dar au fost modificat pentru a permite rotația continuă în ambele direcții,

în loc de mișcarea la un unghi specific. Aceste semnale sunt de obicei pulsuri modulate în lățime (PWM), care indică viteza și direcția dorită. PWM ste o tehnică utilizată pentru a controla puterea sau viteza unui dispozitiv electric prin variația duratei impulsurilor de semnal electric, numite și pulsuri. [27] [28]

Figura 2.11: Servomotor [9]

2.8.3 Potentiometrul liniar

Potentiometrele sunt dispozitive pasive utilizate pentru a varia rezistența într-un circuit electric, fiind esențiale în multe aplicații electronice și electrice. Un potentiometru este un tip de rezistor variabil care are trei terminale: două fixe și unul mobil (numit wiper). Funcționarea sa se bazează pe principiul divizării tensiunii.

Cele trei terminale sunt conectate astfel:

- Două terminale sunt conectate la capetele unei rezistențe fixe.
- Al treilea terminal, wiper-ul, este conectat la un punct intermediar pe rezistență, al cărui loc poate fi schimbat mecanic.
- Prin deplasarea wiper-ului, se modifică raportul dintre cele două rezistențe parțiale create de acesta, permițând ajustarea tensiunii de ieșire sau curentului în circuit.

Tipuri de potentiometre:

- Liniare: rezistența variază proporțional cu deplasarea wiper-ului.
- Logaritmice (audio): rezistența variază în mod logaritmic, fiind adesea utilizate în aplicații audio pentru controlul volumului.

Figura 2.12: Potentiometru

2.8.4 Rezistențe

Rezistențele sunt componente electronice pasive care sunt folosite pentru a controla fluxul de curent într-un circuit electric. Acestea sunt utilizate pentru realizarea divizorului de tensiune conectat la intrarea XADC-ului. Un divizor de tensiune este un circuit simplu, format din două sau mai multe rezistențe conectate în serie, care împarte tensiunea de intrare în mai multe tensiuni mai mici.

Figura 2.13: Rezistențe [10]

2.8.5 Convertor nivel logic 3.3V 5V

Diferite componente și microcontrolere operează la diferite niveluri de tensiune logică. De exemplu, multe microcontrolere moderne și senzori funcționează la 3.3V, în timp ce multe alte periferice și dispozitive mai vechi funcționează la 5V. Comunicarea directă între aceste dispozitive fără un convertor de nivel logic poate duce la daune hardware sau la o funcționare incorectă, deoarece semnalele de 5V pot fi prea mari pentru dispozitivele de 3.3V, iar semnalele de 3.3V pot fi prea mici pentru a fi recunoscute corect de dispozitivele de 5V. MOSFET-urile pot fi utilizate pentru a crea un convertor de nivel logic bidirecțional prin utilizarea unui tranzistor N-channel și a unei rezistențe pull-up. Modulul utilizat este compus din 4 canale uni-direcționale, la HV se conectează sursa de alimentare de 5V, iar la LV se conectează sursa de alimentare de 3.3 V.

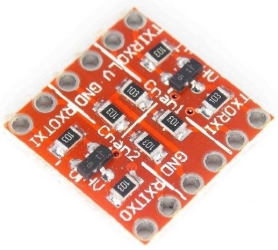


Figura 2.14: Convertor nivel logic 3.3V - 5V [10]

2.8.5.1 Convertorul CMOS cross-coupled unidirectional

Un circuit CMOS încrucișat pentru convertirea nivelului logic este unul dintre cele mai preferate și robuste designuri pentru conversia nivelurilor logice digitale. După cum se arată în figura, un convertor de nivel CMOS simplu unidirecțional încrucișat (cross-coupled) constă din șase tranzistori, M1-M6, care permit transferul tensiunii semnalului de intrare A, alimentat la tensiunea VDDL, către semnalul de ieșire B, alimentat la tensiunea VDDH. Circuitul poate realiza atât conversia de la tensiune înaltă la joasă, cât și de la joasă la înaltă, cu condiția ca VDDH și VDDL să fie în intervalul de operare.

Principiul de funcționare al circuitului:

- Când semnalul A ajunge la nivelul logic „1” (tensiunea la A = VDDL), tranzistorii NMOS M3 și M6 se deschid. M6 trage tensiunea porții tranzistorului NMOS M4 la VSS (masa comună), închizându-l și astfel izolând propagarea masei către ieșirea B. Similar, M3 trage tensiunea porții tranzistorului PMOS M2, deschizându-l și permițând propagarea tensiunii de alimentare VDDH către ieșirea B, ridicând-o la nivelul logic „1” (tensiunea la B = VDDH).
- Când semnalul A ajunge la nivelul logic „0” (tensiunea la A = VSS), tranzistorul PMOS M5 se deschide, propagând VDDL către poarta tranzistorului NMOS M4. M4 se deschide apoi și propagă tensiunea VSS (nivel logic „0”) la B. Astfel, traducerea tensiunii de la VDDL la VDDH este asigurată de la A la B. Tranzistorul suplimentar M1 este utilizat ca restaurator de nivel pentru poarta lui M2, asigurând funcționarea corectă în timpul tranzițiilor de semnal.

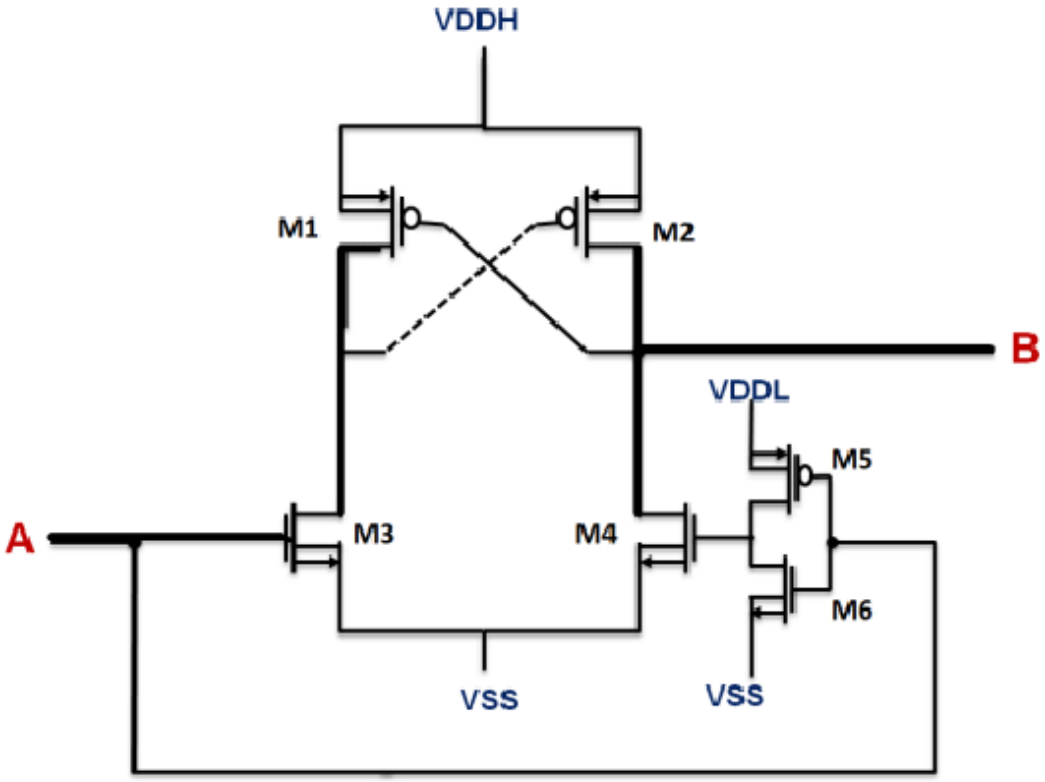


Figura 2.15: Convertor logic CMOS cross-coupled unidirecțional[10]

Capitolul 3

Proiectare și implementare

3.1 Specificații

Acest capitol se concentrează pe implementarea practică a proiectului propus, descriind în detaliu blocurile componente, legăturile dintre ele și conceptele fundamentale ale funcționării sistemului automat de control al energiei electrice. Informațiile sunt prezentate într-o ordine care respectă designul proiectului, menținând coerența între subcapitole. Argumentele sunt susținute de scheme bloc și de o simulare exemplificativă, care facilitează înțelegerea modului de operare a fiecărui bloc. Toate simulările au fost realizate folosind Vivado Xilinix 2021.2 si placa a fost programata prin Vitis HLS 2021.2.

3.1.1 Schema de principiu

În cadrul sistemului descris, sunt implicate mai multe componente și interconexiuni esențiale care permit funcționarea corectă și eficientă a dispozitivului bazat pe placa PYNQ-Z2. Voi detalia fiecare componentă și legătură principală:

- Panoul solar este sursa principală de energie pentru sistem. Acesta convertește energia solară în energie electrică continuă (DC) la o tensiune specifică, de obicei în jurul valorii de 12V, în funcție de specificațiile panoului solar utilizat.
- Divizorul de tensiune este o componentă pasivă utilizată pentru a reduce tensiunea de la panoul solar la o valoare mai mică, adecvată pentru măsurarea și procesarea cu XADC-ul (Xilinx Analog-to-Digital Converter) integrat pe placa PYNQ-Z2. În cazul tău, tensiunea este redusă la un interval de [0-1]V, deoarece acesta este maximul admis de XADC pentru a asigura o măsurare precisă și sigură.

- ADC-ul integrat pe placa PYNQ-Z2 este utilizat pentru a converti semnalele analogice provenite de la divizorul de tensiune în valori digitale. Acesta oferă o rezoluție de 12 biți și poate efectua măsurători precise la o rată de eșantionare configurabilă. XADC-ul este esențial în monitorizarea și controlul parametrilor de intrare analogici în aplicații FPGA.
- FPGA-ul din placa PYNQ-Z2 este componenta centrală care gestionează toate operațiunile de control și procesare. Este un FPGA Xilinx Zynq-7000 SoC, care integrează un procesor ARM Cortex-A9 dual-core și o logică FPGA programabilă. FPGA-ul poate implementa algoritmi de control și logici de interfață pentru a gestiona diverse funcționalități ale sistemului, inclusiv comunicarea cu XADC-ul și cu alte dispozitive periferice.
- Între FPGA și servomotoare este folosit un convertor logic de nivel. Acesta este necesar deoarece FPGA-ul și servomotoarele pot avea niveluri de tensiune de operare diferite (de exemplu, FPGA-ul poate opera la 3.3V, în timp ce servomotoarele pot necesita 5V). Convertorul logic de nivel asigură o interfață corespunzătoare între cele două dispozitive, protejând FPGA-ul de tensiuni incorecte și asigurând comunicarea corectă între ele.

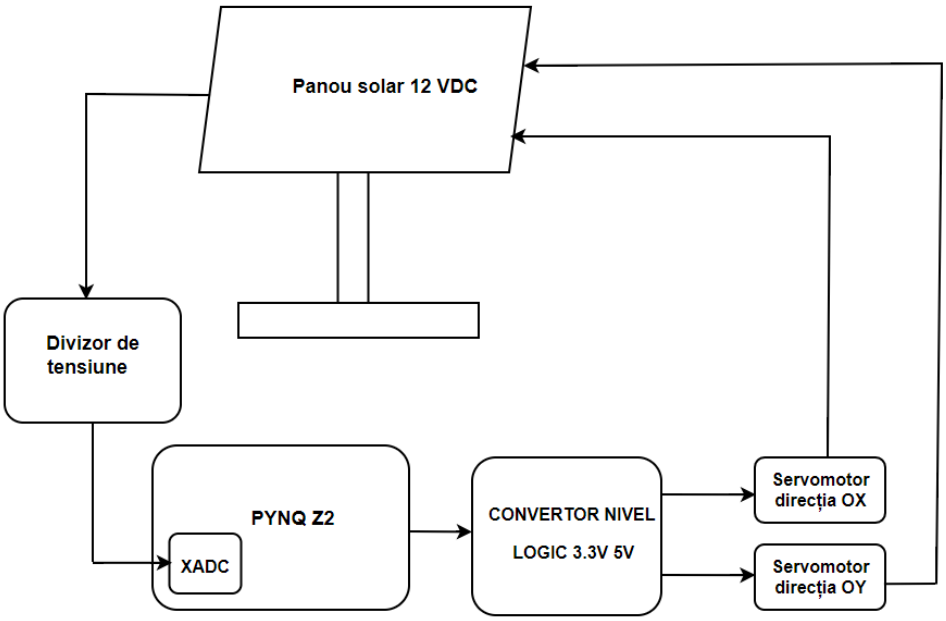


Figura 3.1: Schema de principiu a sistemului automat de control al energiei electrice

3.2 Implementarea digitală a sistemului

Implementarea digitală a sistemului are la bază logica servo controller-ului, care reprezintă creierul sistemului. Acesta primește tensiunea citita de XADC și generează semnalul de control care este trimis la fiecare servomotor. [13]

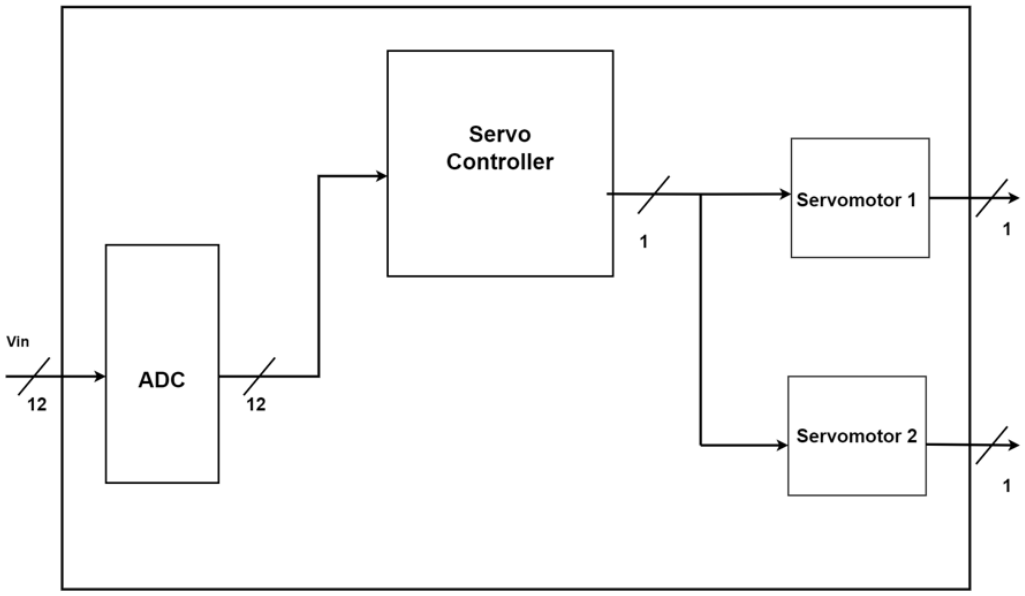


Figura 3.2: Implementarea la nivel logic a sistemului automat de control al energiei electrice

3.2.1 Convertor analog digital (ADC)

3.2.1.1 Configurarea XADC-ului in Xilinix Vivado

Convertorul analog-digital este implementat cu ajutorul IP Catalog din Vivado, selectând pinii(VP/VN)care citesc valoarea analogică și modul de operare a acestuia. Tensiunea analogică pe care ADC-ul o poate citi este de maxim 1V, astfel că trebuie conectat divizorul de tensiune. IP-ul XADC poate fi configurat pentru a opera în diferite moduri, cum ar fi single-ended sau diferențial, în funcție de modul în care semnalul analogic este prezentat. După adăugarea IP-ului ADC în

diagrama Vivado, vor fi selectati pinii FPGA-ului care vor fi conectați la semnalul analogic provenit de la divizorul de tensiune.

Pentru a funcționa, XADC-ul a fost conectat la încă trei module IP: ZYNQ7 Processing System, Processor System Reset și AXI Interconnect, în felul următor:

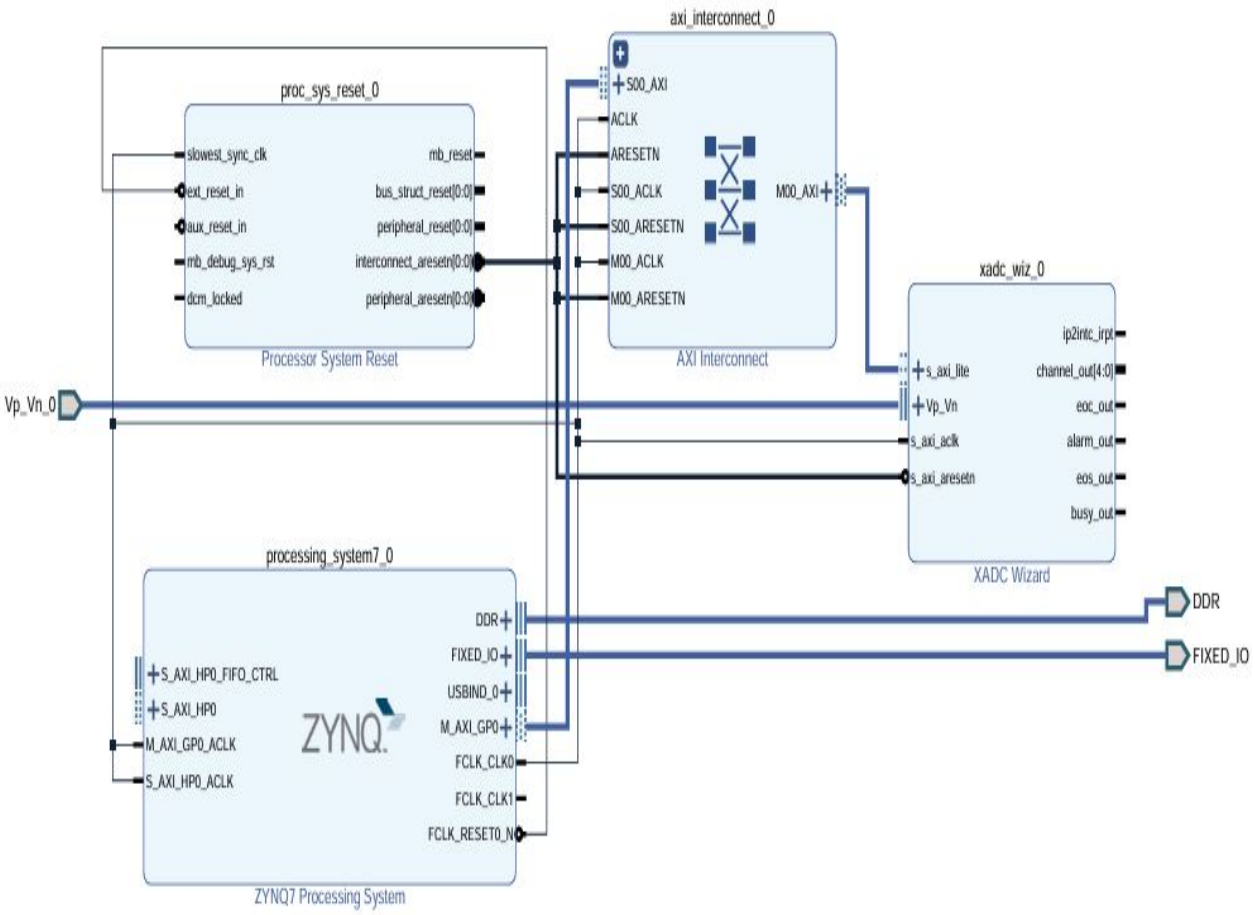


Figura 3.3: Configurarea XADC-ului în Xilinix Vivado IP Catalog

Pașii realizați în instanțierea XADC-ului:

- Citirea se realizează prin intermediul interfeței AXI4Lite.
- Pentru acest proiect avem nevoie de un singur canal, deci modul de operare va fi Single Channel. Se vă utiliza perechea de pini dedicați, nu pinii auxiliari.
- Dorim că XADC-ul să citească tensiunea analogică continuu, de aceea am ales Continuous Mode.
- Alarmele vor fi complet dezactivate pentru a reduce gradul de complexitate.
- Frecvența ceasului, rata de conversie și timpul de achiziție a datelor râm nemodificate: 100MHz, 1000KSPS, respectiv 4 perioade de ceas.

AXI Interconnect este un modul IP disponibil în Vivado IP Catalog, care servește că o interfață centrala de comunicație intre diverse module IP pe platformele FPGA și permite trasferuri de date eficiente intre aceste module. Acesta este configurabil în funcție de nevoile specifice ale proiectului, utilizatorul putând modifica numărul de componente master sau slave, dimensiunea datelor, priorități, optimizând astfel performata și utilizarea resurselor. Suporta diferite versiuni ale protocolului AXI, orecum AXI4 sau AXI4-Lite, integrand astfel diverse module IP disponibile în Vivado. Scopul acestui modul este de a optimiza latenta în sistemele FPGA complexe, reducând astfel și din timpul de dezvoltare și complexitatea proiectului.

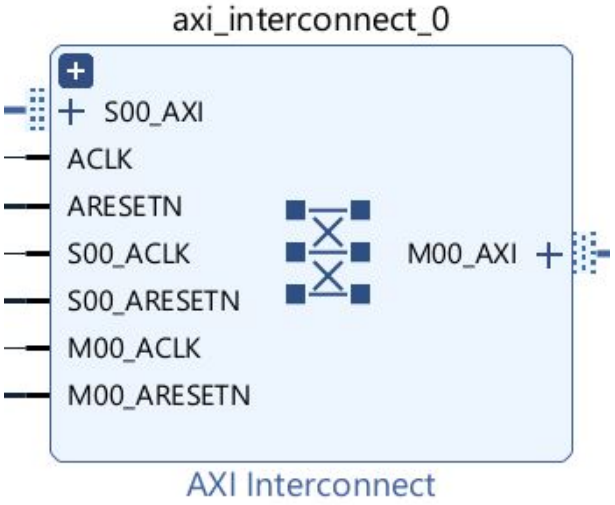


Figura 3.4: Bloc AXI Interconnect

Processing System Reset este un modul IP care gestionează resetarea diferitelor componente ale sistemului, în special în contextul utlizarii unui SoC, precum Zynq-7000 sau Zynq UltraScale++. Acesta este esențial pentru asigurarea unei secvențe corecte de resetare a întregului sistem, astfel încât toate componentele să pornească într-o stare cunoscuta și stabila. Modulul poate genera semnale de resetare atât sincrone, cât și asincrone pentru diverse componente ale sistemului. Configurarea tipica include: selectarea surselor de reset(reset extern, reset software, reset generat de watchdog timer), setarea domeniilor de ceas, setarea parametrilor de debouncing și filtrare pentru semnalele de reset.

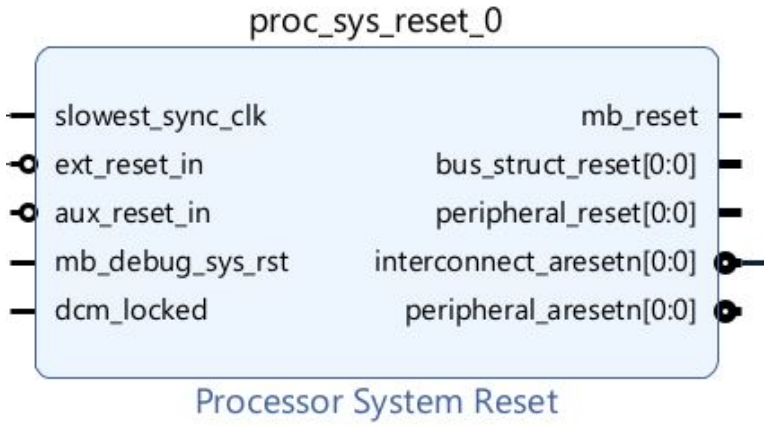


Figura 3.5: Bloc Processing System Reset

Zynq-7000 Processing System (PS) este o componenta integrata a familiei de SoC-uri Zynq-7000 de la Xilinix, care combină un procesor ARM dual-core Cortex-A9 cu un FPGA programabil pe același cip. Aceasta arhitectura hibridă prezintă următoarele caracteristici:

- Fiecare nucleu Cortex-A9 poate rula până la 1 GHz, oferind performanță ridicată pentru sarcinile de procesare generală.
- Include un cache L1 de 32KB pentru instrucțiuni și date pentru fiecare nucleu, plus un cache de 512 KB, partajat.
- Include multiple interfețe de comunicație precum USB, Ethernet, UART, SPI, I2C, și CAN.
- Utilizat pentru proiectarea logicii programabile (PL) și pentru configurarea interfețelor dintre PS și PL.

- AXI Master și Slave: Permite procesorului să comunice cu logica programabilă FPGA prin interfețele AXI. Acestea sunt utilizate pentru transferuri de date de înaltă performanță între PS și logica programabilă (PL - Programmable Logic).
- Utilizat pentru dezvoltarea aplicațiilor software care rulează pe procesorul ARM (Xilinix Vitis HLS)
- Controler de memorie DDR: Gestionarea accesului la memoria DDR3/DDR2 pentru stocarea și accesul rapid la date.
- Controler de memorie Flash: Pentru încărcarea și executarea de cod de boot și pentru stocarea permanentă a datelor.

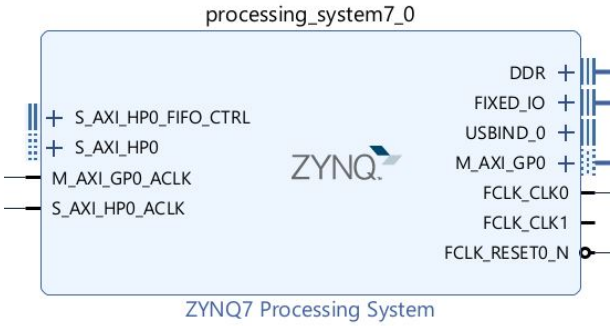


Figura 3.6: Bloc Zynq-7000 Processing System

3.2.1.2 Configurarea XADC-ului in Xilinix Vitis HLS

Configurarea XADC-ului în Xilinx Vitis HLS implică scrierea de cod C/C++ pentru a configura și utiliza XADC-ul în aplicații FPGA. Vitis HLS permite descrierea comportamentală a designurilor hardware, transformând codul de înalt nivel în HDL.

Importarea bibliotecilor necesare și rolul acestora:

- #include "xsysmon.h": citirea valorilor de la diverse canale ADC, monitorizarea parametrilor critici ai sistemului, precum temperatura
- #include "xparameters.h": definește adresele de baza pentru perifice
- #include "xstatus.h": definirea codurilor de stare (de exemplu, XST_SUCCESS, XST_FAILURE)

- #include "xil_exception.h": oferă suport pentru gestionarea excepțiilor și întreruperilor în sistemele Xilinx
- #include "xil_printf.h": furnizează funcționalități de tipărire și debug pentru platformele Xilinx, similar cu biblioteca standard stdio.h, dar optimizată pentru sistemele încorporate
- #include "sleep.h": Include funcții pentru a implementa întârzieri în aplicațiile încorporate
- #define C_BASEADDR 0x43C00000: este o macrodefiniție, reprezintă o constantă

După scrierea completă a codului, pentru a programa FPGA-ul au fost realizați următorii pași:

- Rularea și validarea cu succes a codului(Build Finished)
- Din meniul Vitis, se alege Xilinix, apoi Program FPGA, Program
- Din fereastra Explorer, se apasă click dreapta pe fișierul cu iconiță sub forma de chip integrat → Run as → Launch on Hardware (Single Application Debug(GDB))

Utilizarea documentației PG091 este esențială pentru configurarea corectă a XADC-ului și pentru accesarea registrelor corespunzătoare. Aceasta conține pașii referitori la configurarea de baza a XADC-ului și adresele registrelor responsabile pentru citirea valorilor analogice, dar și a celor pentru măsurarea temperaturii sau tensiunii de pe anumiți senzori. Documentația PG091 furnizează informațiile necesare pentru configurarea corectă a XADC-ului în FPGA-ul Xilinx, inclusiv:

- Configurarea de bază a XADC-ului pentru măsurarea tensiunilor analogice și a altor parametri.
- Instrucțiuni detaliate despre cum să accesezi și să utilizezi registrele XADC-ului pentru citirea datelor convertite.
- Informații despre modurile de funcționare și opțiunile disponibile pentru a adapta XADC-ul nevoilor aplicației tale.

C_BASEADDR + 0x240	$V_{AUXP}[0]$/ $V_{AUXN}[0]$	0x0	R[6]	The 12-bit MSB justified result of A/D conversion on the auxiliary analog input 0 is stored in this register.
C_BASEADDR + 0x244	$V_{AUXP}[1]$/ $V_{AUXN}[1]$	0x0	R[6]	The 12-bit MSB justified result of A/D conversion on the auxiliary analog input 1 is stored in this register.
C_BASEADDR + 0x248	$V_{AUXP}[2]$/ $V_{AUXN}[2]$	0x0	R[6]	The 12-bit MSB justified result of A/D conversion on the auxiliary analog input 2 is stored in this register.
C_BASEADDR + 0x24C	$V_{AUXP}[3]$/ $V_{AUXN}[3]$	0x0	R[6]	The 12-bit MSB justified result of A/D conversion on the auxiliary analog input 3 is stored in this register.
C_BASEADDR + 0x250	$V_{AUXP}[4]$/ $V_{AUXN}[4]$	0x0	R[6]	The 12-bit MSB justified result of A/D conversion on the auxiliary analog input 4 is stored in this register.
C_BASEADDR + 0x254	$V_{AUXP}[5]$/ $V_{AUXN}[5]$	0x0	R[6]	The 12-bit MSB justified result of A/D conversion on the auxiliary analog input 5 is stored in this register.
C_BASEADDR + 0x258	$V_{AUXP}[6]$/ $V_{AUXN}[6]$	0x0	R[6]	The 12-bit MSB justified result of A/D conversion on the auxiliary analog input 6 is stored in this register.

Figura 3.7: Adresele registrelor de citire pentru XADC[11]

XADC Wizard Hard Macro Register Grouping[5]				
C_BASEADDR + 0x200	Temperature	N/A	R[6]	The 12-bit Most Significant Bit (MSB) justified result of on-device temperature measurement is stored in this register.
C_BASEADDR + 0x204	V_{CCINT}	N/A	R[6]	The 12-bit MSB justified result of on-device V_{CCINT} supply monitor measurement is stored in this register.
C_BASEADDR + 0x208	V_{CCAUX}	N/A	R[6]	The 12-bit MSB justified result of on-device V_{CCAUX} Data supply monitor measurement is stored in this register.
C_BASEADDR + 0x20C	V_P/V_N	0x0	R/W[7]	When read: The 12-bit MSB justified result of A/D conversion on the dedicated analog input channel (Vp/Vn) is stored in this register. When written: Write to this register resets the XADC hard macro. No specific data is required.
C_BASEADDR + 0x210	V_{REFP}	0x0	R[6]	The 12-bit MSB justified result of A/D conversion on the reference input V_{REFP} is stored in this register.
C_BASEADDR + 0x214	V_{REFN}	0x0	R[6]	The 12-bit MSB justified result of A/D conversion on the reference input V_{REFN} is stored in this register.
C_BASEADDR + 0x218	V_{BRAM}	0x0	R[6]	The 12-bit MSB justified result of A/D conversion on the reference input V_{BRAM} is stored in this register.

Figura 3.8: Adresele registrelor de configurare pentru XADC[11]

3.2.1.3 Implementarea comunicației UART între FPGA și PC: afișarea valorii citite de XADC

În cazul plăcii de dezvoltare PYNQ-Z2, conexiunea UART se realizează prin portul Micro USB, același port fiind respinsabil și pentru alimentarea FPGA-ului. Configurarea conexiunii în PuTTY a constat în alegerea unei conexiuni de tip seriale. Viteza de baud a fost setată astfel încât să se potrivească cu configurarea dispozitivului, 115200 semnale pe secunda.

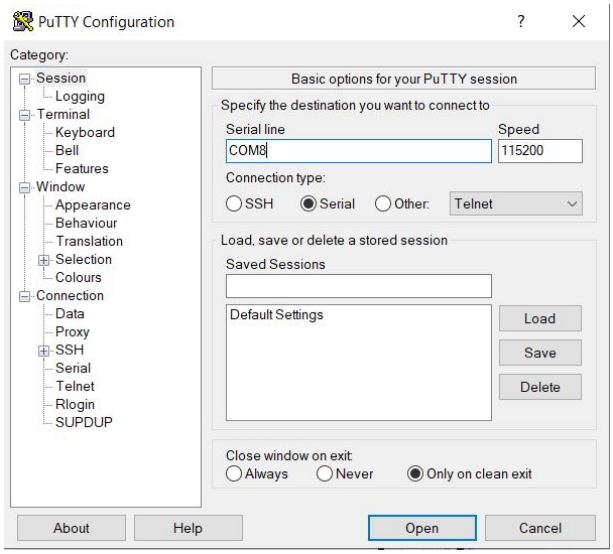


Figura 3.9: Configurare terminal Putty

3.2.2 Servo controller

În procesul de dezvoltare a controlerului pentru servomotoare, inițial a fost scris codul de bază care definește funcționalitatea de control pentru servomotoare. Acest cod a fost apoi integrat în cadrul unui IP (Intellectual Property) în mediul de proiectare Vivado. IP-ul integrat permite controlul precis al servomotoarelor utilizând FPGA-ul disponibil pe placa PYNQ-Z2.

Pașii realizați pentru a realiza IP-ul pentru Servo_Controller:

Tools → Create and Package New IP → Create AXI4 Peripheral → Configurare Add Interfaces → Edit IP → Adăugarea codului în fișierul S00_AXI → Instantierea și modificarea registrelor.

În aceasta etapa, a fost ales numele IP-ului (S00_AXI), tipul interfeței AXI Lite și interfața modulului de tip Slave(procesorul este master, iar servo controller-ul este slave).

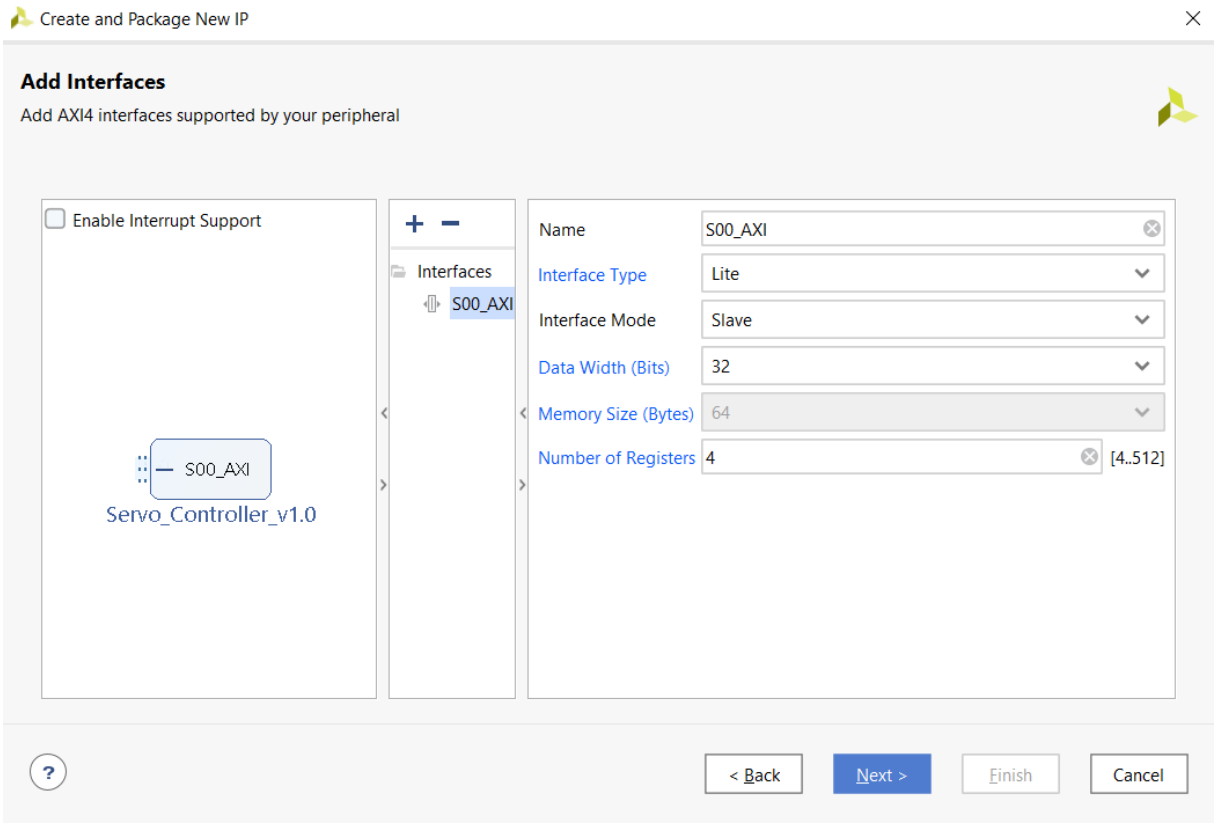


Figura 3.10: Configurarea etapei Add Interfaces

Semnalul digital convertit de XADC este asignat registrului slv_reg0, acesta intra în modulul Servo_Controller. Ieșirea de control semnalul provenit de la XADC sunt trimise mai departe într-un MUX și sunt cititie de pe ieșirea registrului axi_data_reg. Astfel, am determinat ieșirea semnalului de control și îi vom atribui corect pinul: S_AXI__RDATA.

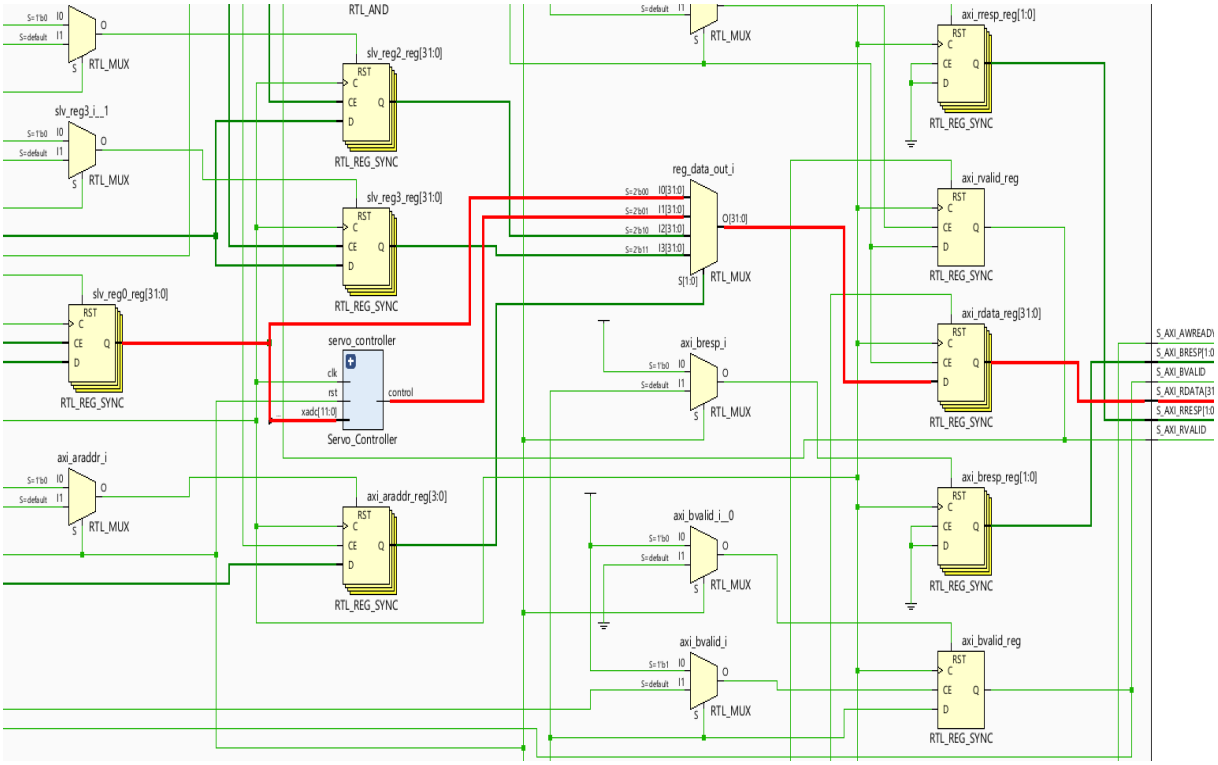


Figura 3.11: Conectarea servo controller-ului la nivelul registrelor din procesor

Perioada semnalului de control și lățimea pulsurilor sunt adaptate specificațiilor din documentația servomotoarelor MG996R. În esență, controller-ul utilizează o tehnică de modulare în lățime de impulsuri (PWM). PWM-ul generează un semnal cu o anumită proporție de umplere, cunoscută și sub denumirea de factor de umplere. Acest factor de umplere este determinat prin modul în care semnalul de ceas și un numărător sunt folosite în circuitul de control. De aceea, în implementarea codului este inclus și un numărător care monitorizează și reglează durata impulsurilor. În paralel cu numărătorul, este necesar și un comparator. Acesta este utilizat pentru a compara valoarea contorului cu valoarea setată pentru factorul de umplere. Astfel, PWM-ul poate ajusta durata pulsului generat în funcție de parametrii specificați, asigurând astfel controlul precis al semnalului de ieșire.

Această abordare permite adaptabilitate și precizie în generarea semnalului PWM, esențială în aplicațiile care necesită control fin al poziției sau al altor caracteristici, cum ar fi în cazul controlului de motoare sau de alte dispozitive care răspund la semnale precise de control.

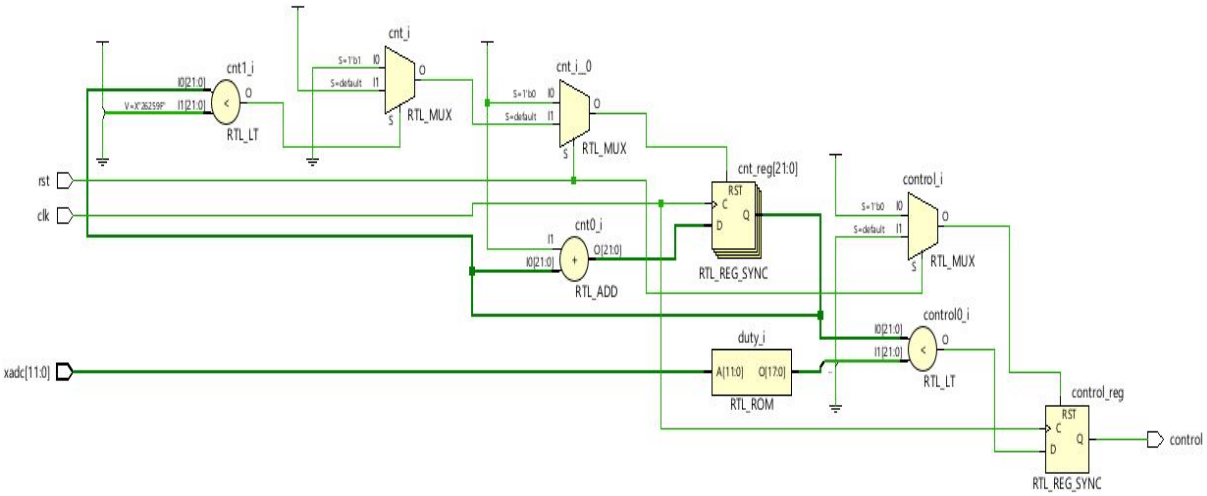
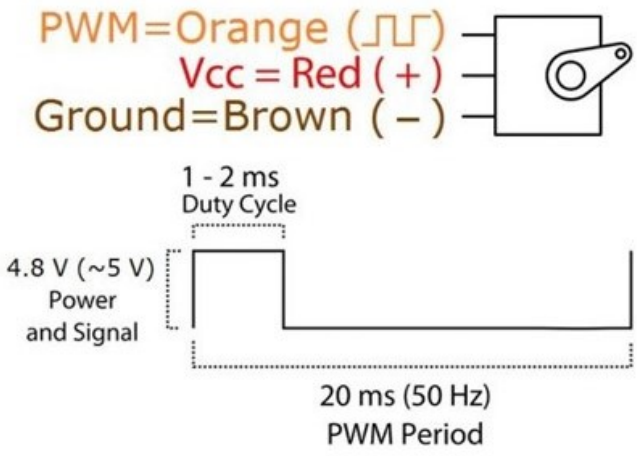


Figura 3.12: Schemă servo controller



Position "0" (1.5ms pulse) is middle, "90" (~2ms pulse) is middle, is all the way to the right, "-90" (~1ms pulse) is all the way to the left.

Figura 3.13: Detalii documentație MG996R [12]

3.3 Modulul de top

Procesorul preia semnalul digital generat de XADC, trimite semnalul către servo controller, preia semnalul de control de la servo controller și îl trimite către servomotoare.

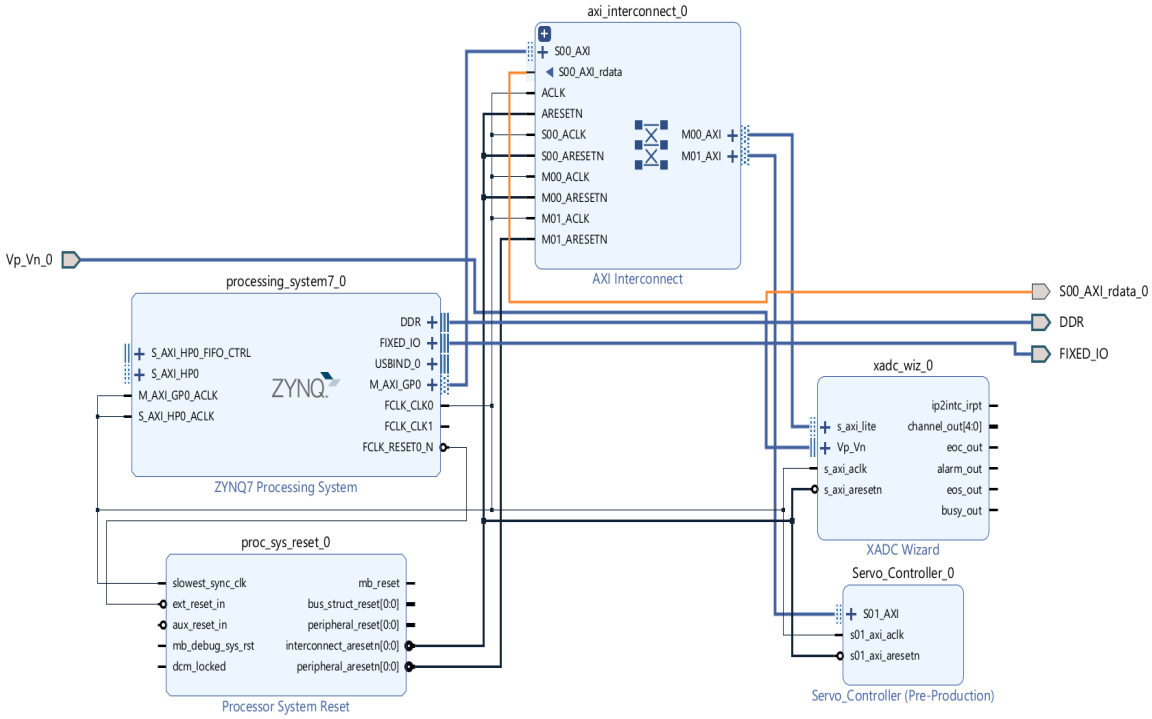


Figura 3.14: Schemă modul de top

3.4 Implementarea fizică a sistemului

Montajul fizic a acestui sistem este alcătuit din următoarele componente:

- Configurația fizică este compusă din panoul solar și servomotoarele fixate cu colțare metalice.
- Divizor de tensiune 12V - 1V.
- Convertor nivel logic 3.3V - 5V.

- FPGA Pynq Z2.

3.4.1 Configurația fizică compusă din panoul solar și servo motoarele fixate cu colțare metalice

Pentru a realiza poziționarea panoului solar de către servomotoare, a fost realizat un sistem cu ajutorul unor colțare metalice. Conform montajului de mai jos, servomotorul responsabil pentru poziționarea pe orizontala este lipit de suportul de plexiglas, iar colțarul metalic este înșurubat în servomotor. Pentru pozionarea pe verticala, al doilea servomotor a fost lipit de colțarul metalic și a fost atașat un al doilea colțar metalic la servomotor. Cel din urmă colțar metalic a fost lipit pe spatele panoului solar.

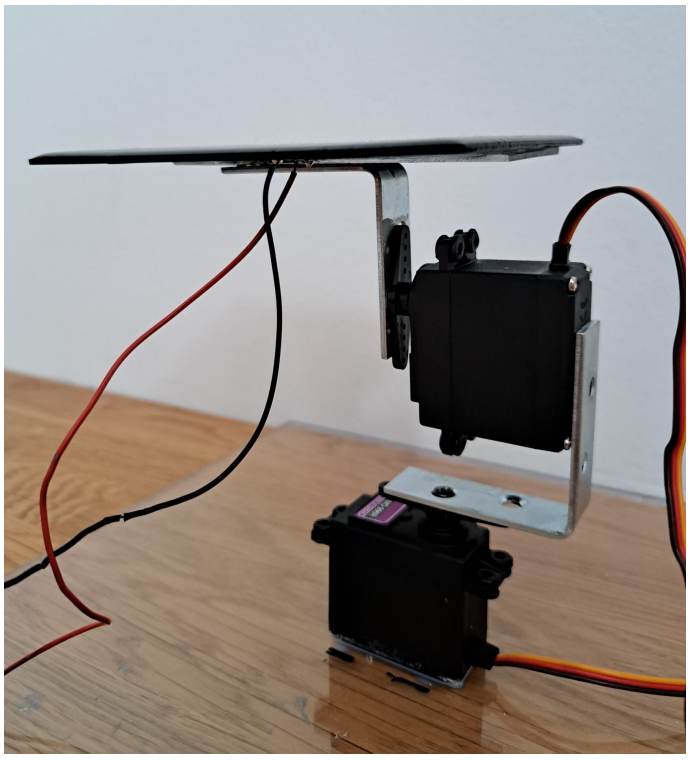


Figura 3.15: Configurația fizică compusă din panoul solar și servomotoarele fixate cu colțare metalice.

3.4.2 Divizor de tensiune 12V → [0,1]V

Divizorul de tensiune este alcătuit dintr-un potențiometru de 10 kΩ și un rezistor cu o valoare de 100 Ω. În imaginea de mai jos, se observă că pentru setarea la jumătate din valoarea sa și tensiunea maximă posibilă de alimentare de 12V provenită de la panoul solar, obținem pe ieșire 235.3 mV. Acest divizor a fost realizat deoarece XADC-ul permite maxim 1V pe intrările analogice ale acestuia.

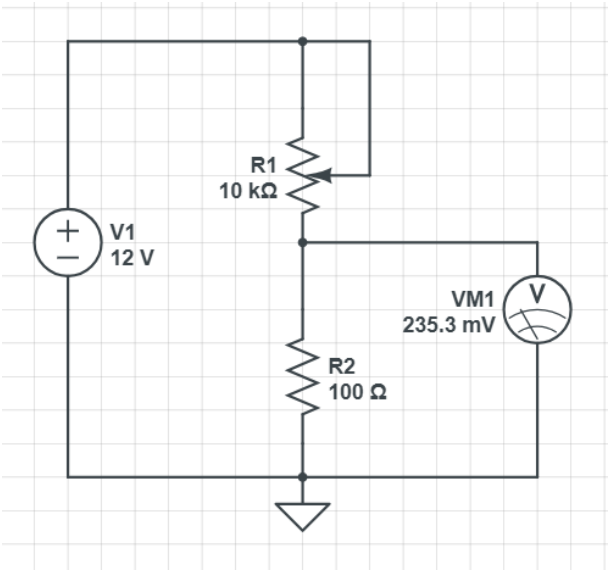


Figura 3.16: Divizor de tensiune 12V → [0,1]V

Formula pentru divizorul de tensiune este:

$$V_{out} = V_{in} \cdot \frac{R_2}{R_1 + R_2} \tag{3.1}$$

Pentru valorile:

$R_2 = 100\,\Omega$
$R_1 = 5\,k\Omega = 5000\,\Omega$
$V_{in} = 12\,V$

Calculăm tensiunea de ieșire (V_{out}):

$$V_{out} = 12\,V \cdot \frac{100\,\Omega}{5000\,\Omega + 100\,\Omega} = 12\,V \cdot \frac{100}{5100} = 12\,V \cdot \frac{1}{51} \approx 0.235\,V$$

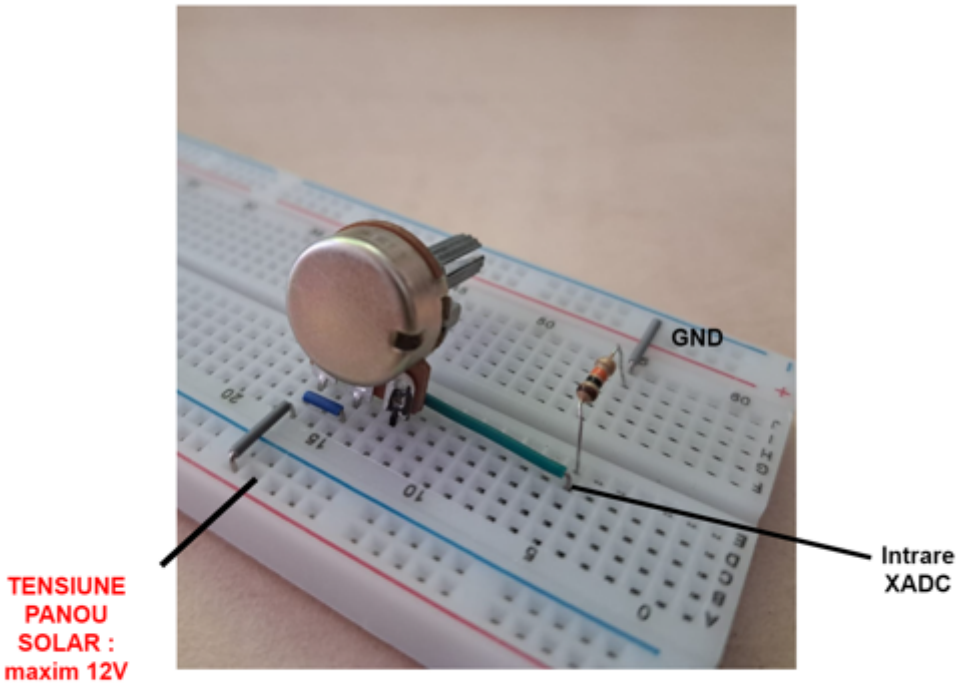


Figura 3.17: Configurarea pe breadboard a divizorului de tensiune

3.4.3 Convertor nivel logic 3.3V → 5V

Servomotoarele MG996R au o tensiune de operare în intervalul [4.8, 7.2] V, iar pinii de ieșire a plăcii PYNQ Z2 suportă maxim 3.3 V. Astfel, pentru ca montajul să fie funcțional, a fost utilizat un convertor de nicel logic conectat între pinii de ieșire a FPGA-ului și intrările servomotoarelor care primesc semnalul PWM. Acest modul trebuie să fie conectat în permanență la tensiunile de 5V și 3.3V, pentru a putea seta nivelel de HV și LV.

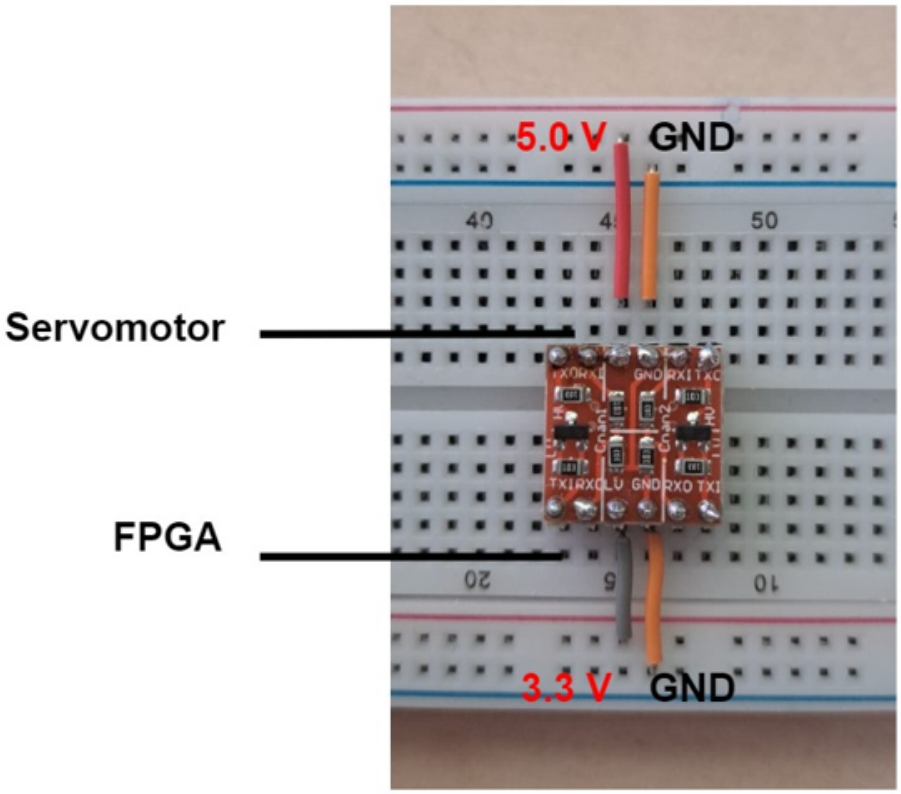


Figura 3.18: Configurarea pe breadboard a convertorului logic 3.3V → 5V

3.4.4 Montajul fizic final

După cum se poate observa, componentele au fost plasate și conectate în ordinea din schmea principala..De asemenea, masa și tensiunile de alimentare pentru componentele de pe breadboard sunt cele furnizate de FPGA. Aceasta placa are atât pin de alimentare de 5V, cât și de 3.3V.

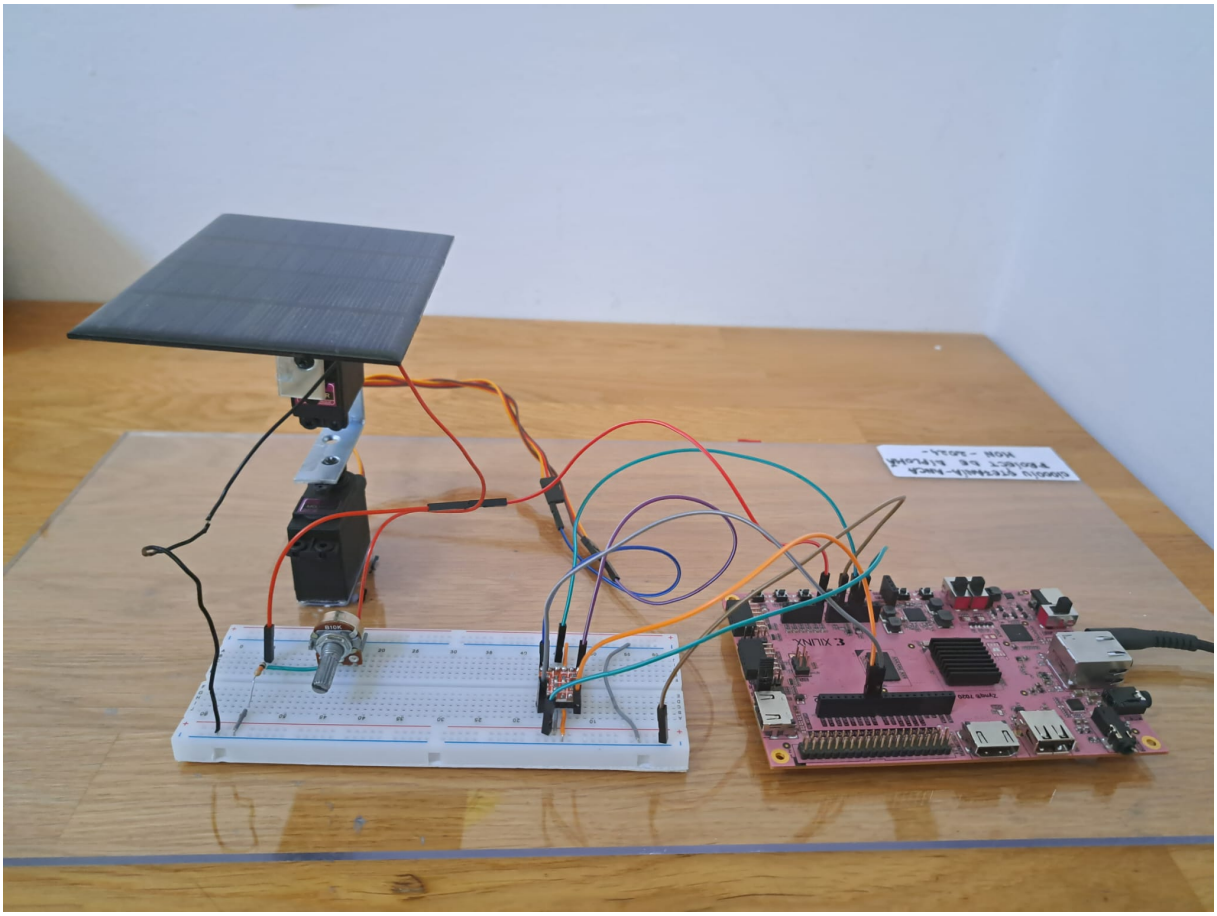


Figura 3.19: Montajul fizic realizat pe suport

Capitolul 4

Rezultate

4.1 Servo controller

În simulare, se observa că semnalul de ieșire, un semnal de PWM, respectă întocmai documentația servomotoarelor MG996R.

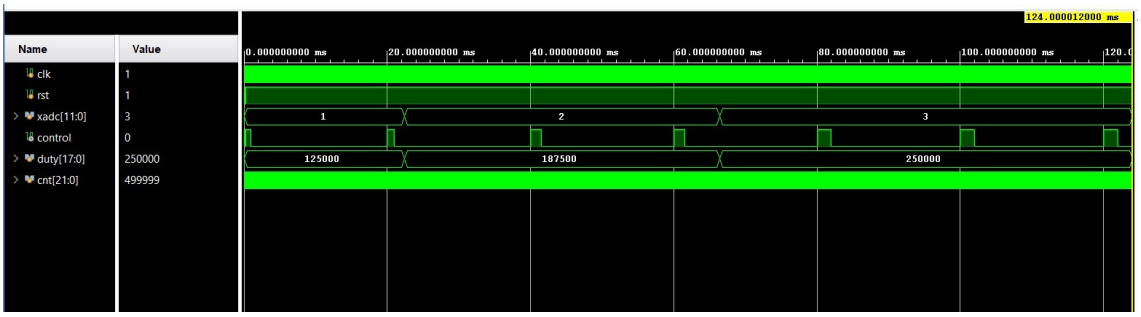


Figura 4.1: Simulare servo controller

Schema de mai jos reprezintă schema în urma procesului de sinteza. Buffer-ele sunt utilizate pentru a îmbunătăți capacitatea de încărcare a semnalelor. Într-o schemă FPGA, semnalele pot avea nevoie să conducă mai multe resurse (cum ar fi alte elemente logice sau linii de ieșire). Prin intermediul buffer-elor, semnalele pot fi amplificate pentru a asigura că pot conduce eficient toate elementele conectate.Utilizarea buffer-elor poate ajuta la reducerea întârzierilor semnalelor, în special în lanțurile critice ale circuitului. În funcție de topologia design-ului, amplificarea semnalelor prin intermediul buffer-elor poate optimiza traseele semnalelor și poate îmbunătăți performanța generală a sistemului. Vivado utilizează buffer-e pentru a gestiona resursele FPGA într-un mod mai eficient. Sinteza logică poate decide automat unde să plaseze buffer-ele pentru a minimiza întârzierile și a optimiza utilizarea resurselor fizice ale FPGA-ului (cum ar fi LUT-urile și flip-flop-urile).

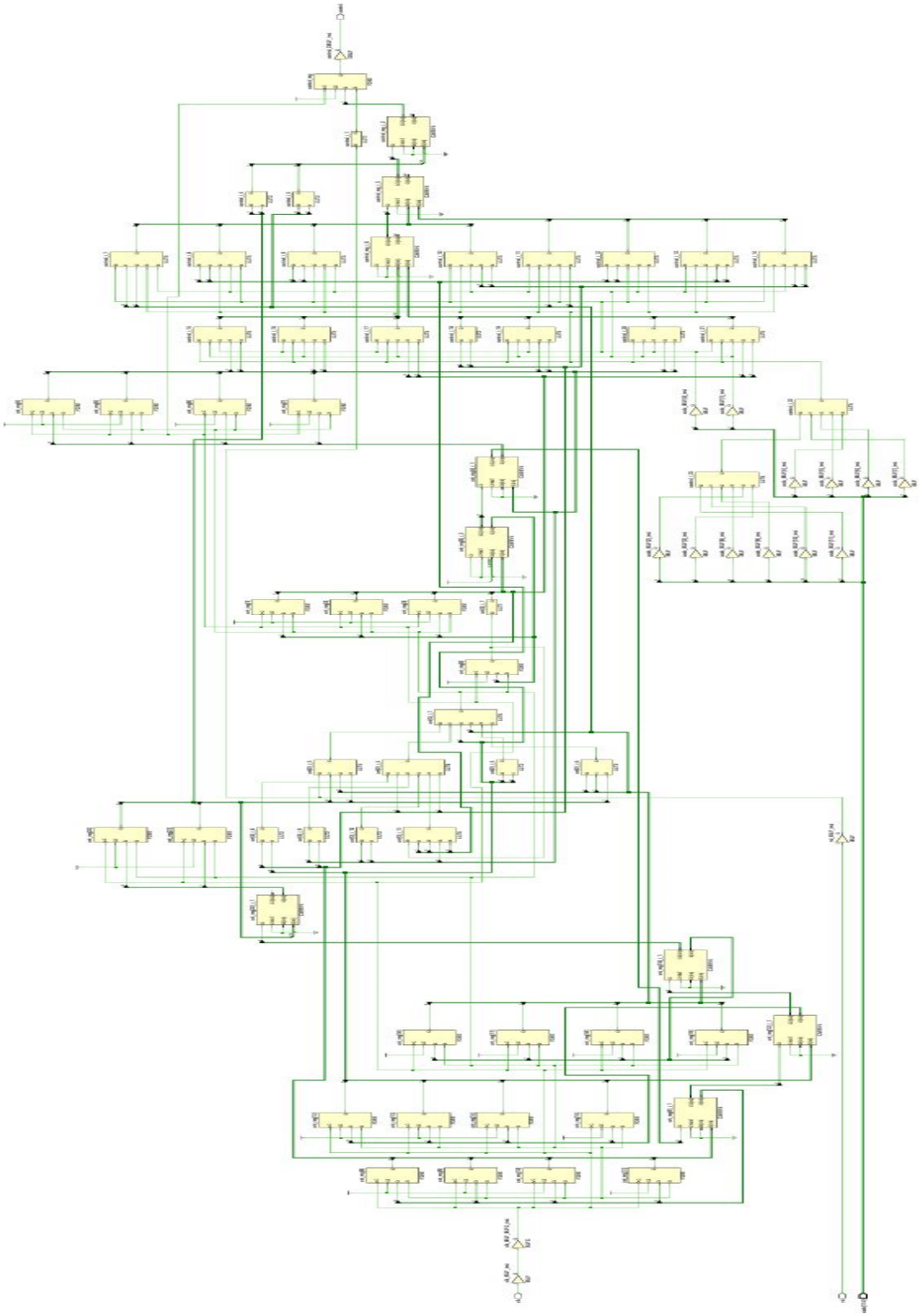


Figura 4.2: Schemă servo controller-ului în urma procesului de sinteză

Raportul de putere consumată în Vivado exprimă estimări și măsurători legate de consumul de putere al design-ului implementat pe FPGA. Acest raport este extrem de util în etapa de proiectare pentru a evalua impactul asupra consumului de putere al design-ului și pentru a face optimizări în funcție de acesta. Puterea statica ste puterea consumată de FPGA când design-ul este în stare statică, adică când nu se fac schimbări semnificative în starea logică a elementelor. Aceasta este influențată de elementele logice statice din FPGA și de configurația acestora. Puterea dinamica reprezinta puterea consumată de FPGA în timpul funcționării active, când are loc activitate logică și semnalele se schimbă. Acest consum poate varia în funcție de frecvența de lucru a FPGA-ului și de numărul de tranzacții logice efectuate într-o unitate de timp.

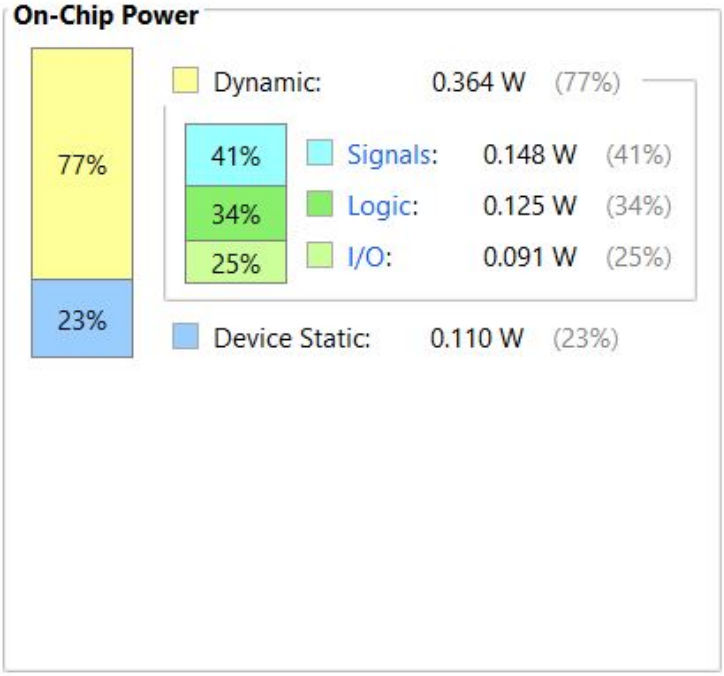


Figura 4.3: Raport putere consumată

In raportul de utilizare, se observa ca pentru modulul de servo controller sunt utilizate 22 de lUT-uri, 23 de module flip-flop (FF) si 15 porturi IO.

Resource	Utilization	Available	Utilization %
LUT	23	53200	0.04
FF	23	106400	0.02
IO	15	125	12.00

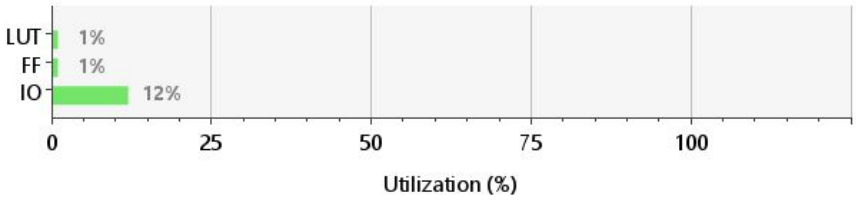


Figura 4.4: Raport de utilizare

4.2 XADC

Conexiunile Vp_Vn_0_v_n și Vp_Vn_0_v_p reprezintă intrările analogice diferențiale (p și n) ale modulului XADC. Aceste semnale trec prin IBUF (Input Buffer), ceea ce sugerează că semnalele sunt pregătite pentru a fi utilizate de către modulele interne ale FPGA.

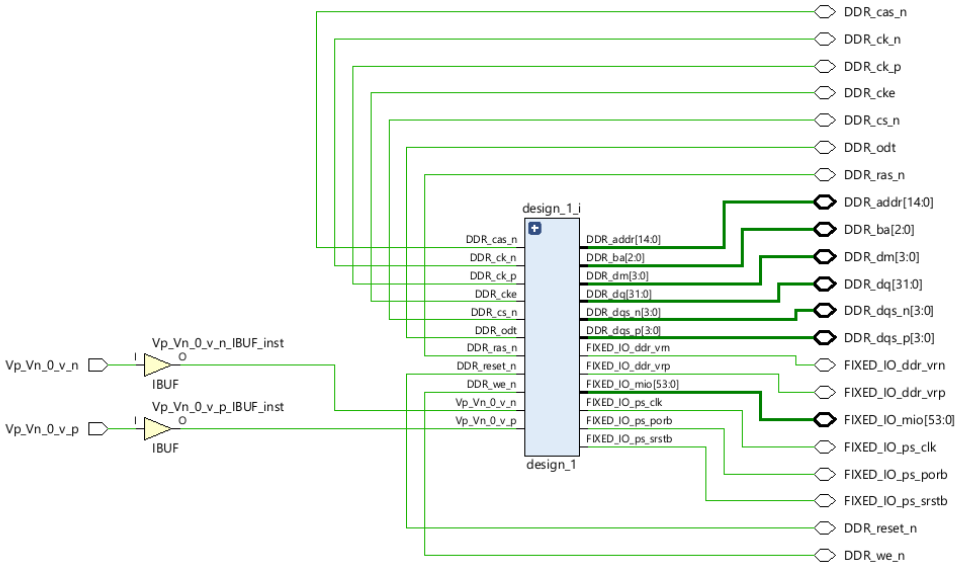


Figura 4.5: Schemă XADC în urma procesului de sinteză

În raportul de utilizare, se observă că pentru modulul de servo controller sutn utilizate 523 de lUT-uri, 712 de module flip-flop (FF) și 48 de module LUTRAM.

Resource	Utilization	Available	Utilization %
LUT	523	53200	0.98
LUTRAM	48	17400	0.28
FF	712	106400	0.67

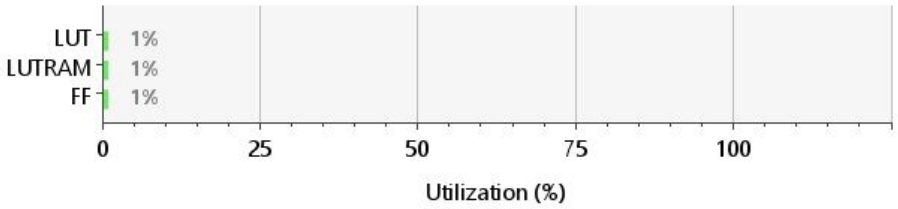


Figura 4.6: Raport de utilizare

Se observa că 96% din puterea consuamta totala se datorează modulului PS7, adică Zynq Processing System. Puterea dinamica consumată este de 1.268 W, iar cea statică este de 0.134 W.

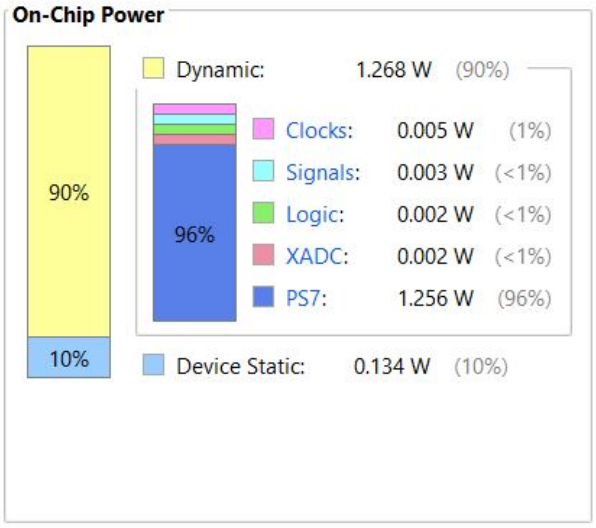


Figura 4.7: Raport putere consumată

4.3 Modulul de top

În urma configurării modulului de top, s-a reușit generarea bitstream-ului, dar nu și citirea tensiunii analogice de la breadboard. În urma încercărilor, am reușit să obțin în consola din terminalul PuTTY, valorea data = 0 mV sau data = 999 mV, afișată în mod continuu.

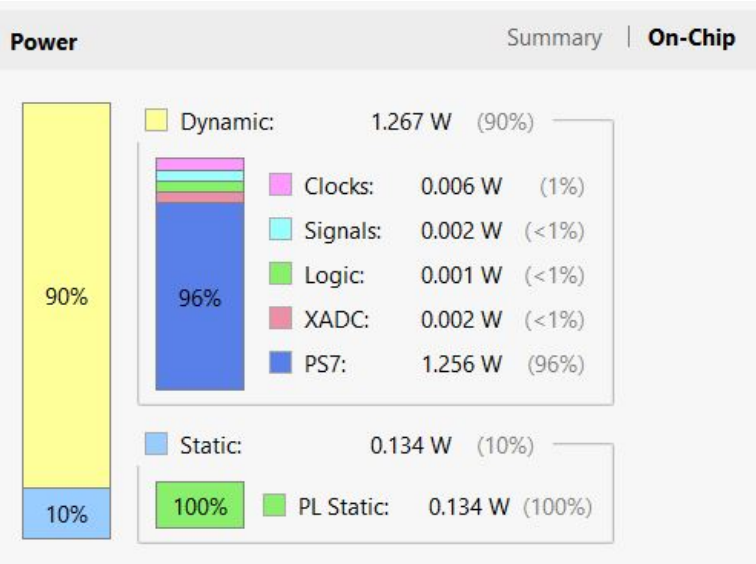


Figura 4.8: Raport putere consumata

Timpul de setup, timpul de hold și lățimea impulsului sunt parametri critici în proiectarea și analiza circuitelor digitale. Acești termeni sunt folosiți pentru a descrie comportamentul temporar al semnalelor digitale în relație cu ceasul (clock) și au un impact semnificativ asupra fiabilității și performanței circuitului. Observăm că acești timpi sunt de ordinul nanosecundelor, sunt timpi foarte optimi.

Resource	Utilization	Available	Utilization...
LUT	734	53200	1.38
LUTRAM	62	17400	0.36
FF	1084	106400	1.02
BUFG	1	32	3.13

Figura 4.9: Raport de utilizare

Timing	**Setup** | Hold | Pulse Width
Worst Negative Slack (WNS):	2.721 ns
Total Negative Slack (TNS):	0 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	2466

Figura 4.10: Timpul de setup

Timing	Setup | **Hold** | Pulse Width
Worst Hold Slack (WHS):	0.037 ns
Total Hold Slack (THS):	0 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	2466
Implemented Timing Report	

Figura 4.11: Timpul de hold

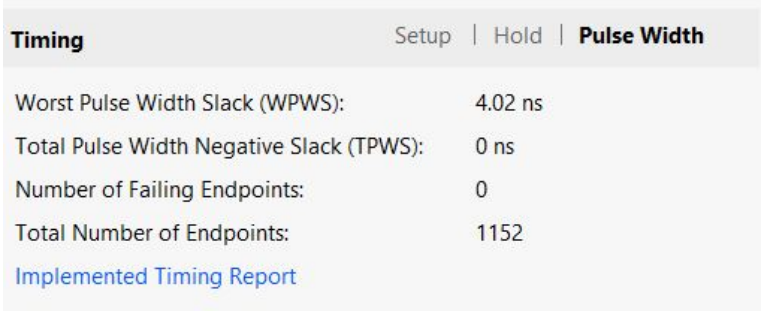


Figura 4.12: Lățimea pulsului

Capitolul 5

Concluzii

5.1 Contribuții personale

Printre contribuțiile personale în cadrul acestui proiect, se enumeră:

- Realizarea analizei State of Art.
- Decizia asupra componentelor electronice utilizate.
- Realizarea divizorului de tensiune.
- Implementarea și testarea XADC-ului.
- Implementarea și testarea modulului Servo_Controller.
- Implementarea unui design hardware în Vivado care să conțină toate modulele necesare sisemului.
- Realizarea IP-ului Servo_Controller.
- Realizarea de cod C în Vitis HLS pentru afișarea datelor măsurate de XADC.
- Realizarea montajului fizic ce constituie toate componentele prezentare în acest proiect.

5.2 Probleme întâmpinate pe parcursul proiectului

- O primă problemă întâmpinată a fost legată de circuitul de rezistențe și fotorezistențe. Deși am proiectat schema circuitului, măsurătorile efectuate în laborator au indicat o tensiune de 0V, conform multimetrelor utilizate. Am achiziționat fotorezistențe cu valori mai mari, presupunând că problema

ar rezida în caracteristicile acestora, însă rezultatul a rămas neschimbat. Ipoteza inițială a fost că fotorezistențele necesită un timp mai lung pentru a deveni active. În consecință, am abandonat acest circuit și am decis să utilizez tensiunea generată direct de panoul solar, de până la 12V. Astfel, a fost necesară adăugarea unui divizor de tensiune, deoarece XADC-ul acceptă valori analogice de maxim 1V.

- O mare parte din timp a fost dedicată studierii documentației XADC-ului. Inițial, nu am reușit să înțeleg modul optim de configurare pentru aplicația mea, respectiv dacă ar trebui setat în modul Single Channel sau Channel Sequencer, și dacă trebuie configurat prin AXI4Lite sau DRP. Ulterior, a trebuit să determin care module din IP Catalog trebuie conectate pentru a asigura funcționarea corectă. După configurarea modulelor, am constatat că instrumentul software Vitis HLS nu putea fi lansat din Vivado. Problema a fost rezolvată prin efectuarea unui upgrade la program și alegerea suitei Vitis HLS, care include Vivado (Vivado nu include Vitis HLS). După înțelegerea pașilor de tranziție din Vivado în Vitis HLS, am construit codul în limbajul C pentru inițializarea registrelor cu adresele corespunzătoare și afișarea datelor măsurate. Am optat pentru afișarea datelor prin terminalul PuTTY, deoarece oferă o interfață mai dinamică și mai ușor de urmărit decât simpla afișare din Vitis HLS.

- Inițial, intenționam să reproduc un proiect interesant găsit pe internet, care controla panoul solar printr-un FSM, cu numărătoare verticale și orizontale. Cu toate acestea, nu am reușit să înțeleg logica din spatele proiectului și nici să transform codul VHDL în Verilog, motiv pentru care am decis să construiesc un modul simplu care primește valoarea citită de XADC și determină modul în care sunt poziționate servomotoarele și, implicit, panoul solar.

- Nu am găsit servomotoare care să funcționeze la o tensiune maximă de 3.3V, așa că a trebuit să adaptez sistemul și să adaug un convertor logic de nivel pentru a realiza conexiunea între pinii de ieșire ai FPGA-ului și servomotoare.

- Crearea IP-ului Servo_Controller și conectarea acestuia la restul diagramei IP, a reprezentat de asemenea o provocare, mai ales la partea conectării modulului de control cu procesorul, practic crearea unui flux de date intre XADC, servo controller și procesor.

5.3 Dezvoltări ulterioare

În cadrul dezvoltărilor ulterioare, se preconizează implementarea proiectului utilizând un modul FSM (Finite State Machine) și numărătoare verticale și orizontale , ceea ce ar conduce la o optimizare mai eficientă a sistemului. Datorită naturii sale practice, acest proiect poate servi drept temă de studiu pentru studenți, oferindu-le oportunitatea de a-și dezvolta competențele în utilizarea FPGA-ului, interpretarea și aplicarea documentației tehnice, precum și în lucrul cu servomotoare, implementând module PWM în Verilog.

Bibliography

[1] *Pynq-z2 reference manual v1.0.*

[2] Mouser Electronics. Amd / xilinx zynq®-7000 soc first generation architecture.

[3] FPGAKey. Configurable logic module.

[4] NAND LAND. What is a block ram (bram) in an fpga?

[5] *7 Series FPGAs and Zynq-7000 SoC XADC Dual 12-Bit 1 MSPS Analog-to-Digital Converter User Guide UG480.* AMD Technical Information Portal, 2016.

[6] Pantech Solutions. Uart interface with spartan3an fpga starter kit.

[7] FYS4220. State machines.

[8] eMAG. Panou solar.

[9] Farnell. Parallax 900-00005.

[10] Daly Electronic. Rezistenta 100 1w ±1

[11] *XADC Wizard v3.3 Product Guide (PG091).* AMD Technical Information Portal, 2016.

[12] ALDATASHEET.COM. Mg996r datasheet.

[13] AUTODESK Instructables. Fpga solar panel optimizer.

[14] Jui-Ho Chen, Her-Terng Yau, and Tzu-Hsiang Hung. Design and implementation of fpga-based taguchi-chaos-pso sun tracking systems. *Mechatronics*, 25:55–64, 2015.

[15] Hardik Modi, K. Frank, Amtulla Khambhati, and Himanshu K. Patel. Dual axis solar tracking system using verilog. 2019.

[16] Peter Minns and Ian Elliott. *FSM-based Digital Design using Verilog HDL.* John Wiley Sons Ltd, 2008.

[17] *7 Series FPGAs Configuration, UG470*. AMD Technical Information Portal, v1.17 edition, 2023.

[18] *Vitis High-Level Synthesis User Guide UG1399*. AMD Technical Information Portal, 2024.

[19] National Instruments. Programming with an fpga target.

[20] Education 4u. Fpga architecture | switch matrix.

[21] Medium. Programmable logic devices.

[22] *7 Series FPGAs GTP Transcievers UG482*. AMD Technical Information Portal, 2016.

[23] Analog Devices. Uart: A hardware communication protocol understanding universal asynchronous receiver/transmitter.

[24] Columbia University. Putty — configuration and tutorial.

[25] WEVOLVER. Understanding baud rate: Why is it important?

[26] NPRE 201 Solar Panel Project. Npre 201 solar panel project.

[27] Makerguides. How to control a 360 degree servo motor with arduino.

[28] Syinthims. Introduction to servo motors.

Annex A

Anexa 2: Codul sursa

```
Servo_Controller.v

module Servo_Controller(
    input         clk,
    input         rst,
    input [11:0] xadc,
    output reg    control
);

    reg [17:0] duty;
    reg [21:0] cnt;

    always @(*) begin
        case (xadc)
            12'd1:   duty = 18'd125000;     // 1   ms. -90 degrees.
            12'd2:   duty = 18'd187500;     // 1.5 ms. 0   degrees.
            12'd3:   duty = 18'd250000;     // 2   ms. 90  degrees.
            default: duty = 18'd0;
        endcase
    end

    // OBS: Valorile 1, 2 si 3 sunt orientative. Trebuie puse valorile pe care le
    returneaza ADC-ul pentru tensiunile potrivite.

    always @(posedge clk) begin
        if (!rst) begin
            cnt     <= 22'd0;
            control <= 1'b0;
        end
        else begin
            if (cnt < 22'd2499999)
                cnt <= cnt + 1;
            else
                cnt <= 22'd0;
            if (cnt < duty)
                control <= 1'b1;
            else
                control <= 1'b0;
        end
    end

endmodule

Servo_Controller.TB

`timescale 1ns / 1ps

```

```
47 module Servo_Controller_TB();
48
49     reg        clk;
50     reg        rst;
51     reg [11:0] xadc;
52     wire       control;
53
54     Servo_Controller DUT(
55         .clk(clk),
56         .rst(rst),
57         .xadc(xadc),
58         .control(control)
59     );
60
61     initial begin
62         clk = 1'b0;
63         forever #4 clk = !clk;  // T = 8 ns <=> f = 125 MHz
64     end
65
66     initial begin
67         #1.5;
68         rst = 1'b0;
69         xadc = 12'd1;
70         #13;
71         rst = 1'b1;
72         repeat (2800000) @(posedge clk);
73         xadc = 12'd2;
74         repeat (5500000) @(posedge clk);
75         xadc = 12'd3;
76         repeat (7200000) @(posedge clk);
77         $finish;
78     end
79
80
81 endmodule
82 `timescale 1 ns / 1 ps
83
84 module Servo_Controller(
85     input        clk,
86     input        rst,
87     input [11:0] xadc,
88     output reg   control
89 );
90
91     reg [17:0] duty;
92     reg [21:0] cnt;
93
94     always @(*) begin
95         case (xadc)
96             12'd1:   duty = 18'd125000;     // 1   ms. -90 degrees.
97             12'd2:   duty = 18'd187500;     // 1.5 ms. 0   degrees.
98             12'd3:   duty = 18'd250000;     // 2   ms. 90  degrees.
99             default: duty = 18'd0;
100        endcase
101    end
102
103    // OBS: Valorile 1, 2 si 3 sunt orientative. Trebuie puse valorile pe care le
       returneaza ADC-ul pentru tensiunile potrivite.
104
105    always @(posedge clk) begin
106        if (!rst) begin
107            cnt     <= 22'd0;
```

```
108             control <= 1'b0;
109         end
110         else begin
111             if (cnt < 22'd2499999)
112                 cnt <= cnt + 1;
113             else
114                 cnt <= 22'd0;
115             if (cnt < duty)
116                 control <= 1'b1;
117             else
118                 control <= 1'b0;
119         end
120     end
121 
122 endmodule
123 
124 
125 
126   module Servo_Controller_v1_0_S01_AXI #
127   (
128     // Users to add parameters here
129 
130     // User parameters ends
131     // Do not modify the parameters beyond this line
132 
133     // Width of S_AXI data bus
134     parameter integer C_S_AXI_DATA_WIDTH  = 32,
135     // Width of S_AXI address bus
136     parameter integer C_S_AXI_ADDR_WIDTH  = 4
137   )
138   (
139     // Users to add ports here
140 
141     // User ports ends
142     // Do not modify the ports beyond this line
143 
144     // Global Clock Signal
145     input wire  S_AXI_ACLK,
146     // Global Reset Signal. This Signal is Active LOW
147     input wire  S_AXI_ARESETN,
148     // Write address (issued by master, acceped by Slave)
149     input wire [C_S_AXI_ADDR_WIDTH-1 : 0] S_AXI_AWADDR,
150     // Write channel Protection type. This signal indicates the
151         // privilege and security level of the transaction, and whether
152         // the transaction is a data access or an instruction access.
153     input wire [2 : 0] S_AXI_AWPROT,
154     // Write address valid. This signal indicates that the master signaling
155         // valid write address and control information.
156     input wire  S_AXI_AWVALID,
157     // Write address ready. This signal indicates that the slave is ready
158         // to accept an address and associated control signals.
159     output wire  S_AXI_AWREADY,
160     // Write data (issued by master, acceped by Slave)
161     input wire [C_S_AXI_DATA_WIDTH-1 : 0] S_AXI_WDATA,
162     // Write strobes. This signal indicates which byte lanes hold
163         // valid data. There is one write strobe bit for each eight
164         // bits of the write data bus.
165     input wire [(C_S_AXI_DATA_WIDTH/8)-1 : 0] S_AXI_WSTRB,
166     // Write valid. This signal indicates that valid write
167         // data and strobes are available.
168     input wire  S_AXI_WVALID,
169     // Write ready. This signal indicates that the slave
```

```
170         // can accept the write data.
171     output wire  S_AXI_WREADY,
172     // Write response. This signal indicates the status
173         // of the write transaction.
174     output wire [1 : 0] S_AXI_BRESP,
175     // Write response valid. This signal indicates that the channel
176         // is signaling a valid write response.
177     output wire  S_AXI_BVALID,
178     // Response ready. This signal indicates that the master
179         // can accept a write response.
180     input wire  S_AXI_BREADY,
181     // Read address (issued by master, acceped by Slave)
182     input wire [C_S_AXI_ADDR_WIDTH-1 : 0] S_AXI_ARADDR,
183     // Protection type. This signal indicates the privilege
184         // and security level of the transaction, and whether the
185         // transaction is a data access or an instruction access.
186     input wire [2 : 0] S_AXI_ARPROT,
187     // Read address valid. This signal indicates that the channel
188         // is signaling valid read address and control information.
189     input wire  S_AXI_ARVALID,
190     // Read address ready. This signal indicates that the slave is
191         // ready to accept an address and associated control signals.
192     output wire  S_AXI_ARREADY,
193     // Read data (issued by slave)
194     output wire [C_S_AXI_DATA_WIDTH-1 : 0] S_AXI_RDATA,
195     // Read response. This signal indicates the status of the
196         // read transfer.
197     output wire [1 : 0] S_AXI_RRESP,
198     // Read valid. This signal indicates that the channel is
199         // signaling the required read data.
200     output wire  S_AXI_RVALID,
201     // Read ready. This signal indicates that the master can
202         // accept the read data and response information.
203     input wire  S_AXI_RREADY
204 );
205 
206 // AXI4LITE signals
207 reg [C_S_AXI_ADDR_WIDTH-1 : 0]  axi_awaddr;
208 reg   axi_awready;
209 reg   axi_wready;
210 reg [1 : 0]   axi_bresp;
211 reg   axi_bvalid;
212 reg [C_S_AXI_ADDR_WIDTH-1 : 0]  axi_araddr;
213 reg   axi_arready;
214 reg [C_S_AXI_DATA_WIDTH-1 : 0]  axi_rdata;
215 reg [1 : 0]   axi_rresp;
216 reg   axi_rvalid;
217 
218 // Example-specific design signals
219 // local parameter for addressing 32 bit / 64 bit C_S_AXI_DATA_WIDTH
220 // ADDR_LSB is used for addressing 32/64 bit registers/memories
221 // ADDR_LSB = 2 for 32 bits (n downto 2)
222 // ADDR_LSB = 3 for 64 bits (n downto 3)
223 localparam integer ADDR_LSB = (C_S_AXI_DATA_WIDTH/32) + 1;
224 localparam integer OPT_MEM_ADDR_BITS = 1;
225 //------------------------------------------------
226 //-- Signals for user logic register space example
227 //------------------------------------------------
228 //-- Number of Slave Registers 4
229 reg [C_S_AXI_DATA_WIDTH-1:0]  slv_reg0;
230 reg [C_S_AXI_DATA_WIDTH-1:0]  slv_reg1;
231 reg [C_S_AXI_DATA_WIDTH-1:0]  slv_reg2;
```

```
232   reg [C_S_AXI_DATA_WIDTH-1:0]  slv_reg3;
233   wire   slv_reg_rden;
234   wire   slv_reg_wren;
235   reg [C_S_AXI_DATA_WIDTH-1:0]   reg_data_out;
236   integer  byte_index;
237   reg  aw_en;
238
239   // I/O Connections assignments
240
241   assign S_AXI_AWREADY  = axi_awready;
242   assign S_AXI_WREADY = axi_wready;
243   assign S_AXI_BRESP  = axi_bresp;
244   assign S_AXI_BVALID = axi_bvalid;
245   assign S_AXI_ARREADY  = axi_arready;
246   assign S_AXI_RDATA  = axi_rdata;
247   assign S_AXI_RRESP  = axi_rresp;
248   assign S_AXI_RVALID = axi_rvalid;
249   // Implement axi_awready generation
250   // axi_awready is asserted for one S_AXI_ACLK clock cycle when both
251   // S_AXI_AWVALID and S_AXI_WVALID are asserted. axi_awready is
252   // de-asserted when reset is low.
253
254   always @( posedge S_AXI_ACLK )
255   begin
256     if ( S_AXI_ARESETN == 1'b0 )
257       begin
258         axi_awready <= 1'b0;
259         aw_en <= 1'b1;
260       end
261     else
262       begin
263         if (~axi_awready && S_AXI_AWVALID && S_AXI_WVALID && aw_en)
264           begin
265             // slave is ready to accept write address when
266             // there is a valid write address and write data
267             // on the write address and data bus. This design
268             // expects no outstanding transactions.
269             axi_awready <= 1'b1;
270             aw_en <= 1'b0;
271           end
272           else if (S_AXI_BREADY && axi_bvalid)
273               begin
274                 aw_en <= 1'b1;
275                 axi_awready <= 1'b0;
276               end
277         else
278           begin
279             axi_awready <= 1'b0;
280           end
281       end
282   end
283
284   // Implement axi_awaddr latching
285   // This process is used to latch the address when both
286   // S_AXI_AWVALID and S_AXI_WVALID are valid.
287
288   always @( posedge S_AXI_ACLK )
289   begin
290     if ( S_AXI_ARESETN == 1'b0 )
291       begin
292         axi_awaddr <= 0;
293       end
```

```
294       else
295         begin
296           if (~axi_awready && S_AXI_AWVALID && S_AXI_WVALID && aw_en)
297             begin
298               // Write Address latching
299               axi_awaddr <= S_AXI_AWADDR;
300             end
301         end
302     end
303
304     // Implement axi_wready generation
305     // axi_wready is asserted for one S_AXI_ACLK clock cycle when both
306     // S_AXI_AWVALID and S_AXI_WVALID are asserted. axi_wready is
307     // de-asserted when reset is low.
308
309     always @( posedge S_AXI_ACLK )
310     begin
311       if ( S_AXI_ARESETN == 1'b0 )
312         begin
313           axi_wready <= 1'b0;
314         end
315       else
316         begin
317           if (~axi_wready && S_AXI_WVALID && S_AXI_AWVALID && aw_en )
318             begin
319               // slave is ready to accept write data when
320               // there is a valid write address and write data
321               // on the write address and data bus. This design
322               // expects no outstanding transactions.
323               axi_wready <= 1'b1;
324             end
325           else
326             begin
327               axi_wready <= 1'b0;
328             end
329         end
330     end
331
332     // Implement memory mapped register select and write logic generation
333     // The write data is accepted and written to memory mapped registers when
334     // axi_awready, S_AXI_WVALID, axi_wready and S_AXI_WVALID are asserted. Write
        strobes are used to
335     // select byte enables of slave registers while writing.
336     // These registers are cleared when reset (active low) is applied.
337     // Slave register write enable is asserted when valid address and data are
        available
338     // and the slave is ready to accept the write address and write data.
339     assign slv_reg_wren = axi_wready && S_AXI_WVALID && axi_awready && S_AXI_AWVALID
        ;
340
341     always @( posedge S_AXI_ACLK )
342     begin
343       if ( S_AXI_ARESETN == 1'b0 )
344         begin
345           slv_reg0 <= 0;
346           slv_reg1 <= 0;
347           slv_reg2 <= 0;
348           slv_reg3 <= 0;
349         end
350       else begin
351         if (slv_reg_wren)
352           begin
```

```
353           case ( axi_awaddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] )
354             2'h0:
355               for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1;
    byte_index = byte_index+1 )
356                 if ( S_AXI_WSTRB[byte_index] == 1 ) begin
357                   // Respective byte enables are asserted as per write strobes
358                   // Slave register 0
359                   slv_reg0[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +:
    8];
360                 end
361             2'h1:
362               for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1;
    byte_index = byte_index+1 )
363                 if ( S_AXI_WSTRB[byte_index] == 1 ) begin
364                   // Respective byte enables are asserted as per write strobes
365                   // Slave register 1
366                   slv_reg1[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +:
    8];
367                 end
368             2'h2:
369               for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1;
    byte_index = byte_index+1 )
370                 if ( S_AXI_WSTRB[byte_index] == 1 ) begin
371                   // Respective byte enables are asserted as per write strobes
372                   // Slave register 2
373                   slv_reg2[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +:
    8];
374                 end
375             2'h3:
376               for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1;
    byte_index = byte_index+1 )
377                 if ( S_AXI_WSTRB[byte_index] == 1 ) begin
378                   // Respective byte enables are asserted as per write strobes
379                   // Slave register 3
380                   slv_reg3[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +:
    8];
381                 end
382             default : begin
383                         slv_reg0 <= slv_reg0;
384                         slv_reg1 <= slv_reg1;
385                         slv_reg2 <= slv_reg2;
386                         slv_reg3 <= slv_reg3;
387                       end
388           endcase
389         end
390     end
391   end
392
393   // Implement write response logic generation
394   // The write response and response valid signals are asserted by the slave
395   // when axi_wready, S_AXI_WVALID, axi_wready and S_AXI_WVALID are asserted.
396   // This marks the acceptance of address and indicates the status of
397   // write transaction.
398
399   always @( posedge S_AXI_ACLK )
400   begin
401     if ( S_AXI_ARESETN == 1'b0 )
402       begin
403         axi_bvalid  <= 0;
404         axi_bresp   <= 2'b0;
405       end
406     else
```

```
407       begin
408         if (axi_awready && S_AXI_AWVALID && ~axi_bvalid && axi_wready &&
      S_AXI_WVALID)
409           begin
410             // indicates a valid write response is available
411             axi_bvalid <= 1'b1;
412             axi_bresp  <= 2'b0; // 'OKAY' response
413           end                   // work error responses in future
414         else
415           begin
416             if (S_AXI_BREADY && axi_bvalid)
417               //check if bready is asserted while bvalid is high)
418               //(there is a possibility that bready is always asserted high)
419               begin
420                 axi_bvalid <= 1'b0;
421               end
422           end
423       end
424   end
425
426   // Implement axi_arready generation
427   // axi_arready is asserted for one S_AXI_ACLK clock cycle when
428   // S_AXI_ARVALID is asserted. axi_awready is
429   // de-asserted when reset (active low) is asserted.
430   // The read address is also latched when S_AXI_ARVALID is
431   // asserted. axi_araddr is reset to zero on reset assertion.
432
433   always @( posedge S_AXI_ACLK )
434   begin
435     if ( S_AXI_ARESETN == 1'b0 )
436       begin
437         axi_arready <= 1'b0;
438         axi_araddr  <= 32'b0;
439       end
440     else
441       begin
442         if (~axi_arready && S_AXI_ARVALID)
443           begin
444             // indicates that the slave has acceped the valid read address
445             axi_arready <= 1'b1;
446             // Read address latching
447             axi_araddr  <= S_AXI_ARADDR;
448           end
449         else
450           begin
451             axi_arready <= 1'b0;
452           end
453       end
454   end
455
456   // Implement axi_arvalid generation
457   // axi_rvalid is asserted for one S_AXI_ACLK clock cycle when both
458   // S_AXI_ARVALID and axi_arready are asserted. The slave registers
459   // data are available on the axi_rdata bus at this instance. The
460   // assertion of axi_rvalid marks the validity of read data on the
461   // bus and axi_rresp indicates the status of read transaction.axi_rvalid
462   // is deasserted on reset (active low). axi_rresp and axi_rdata are
463   // cleared to zero on reset (active low).
464   always @( posedge S_AXI_ACLK )
465   begin
466     if ( S_AXI_ARESETN == 1'b0 )
467       begin
```

```
468         axi_rvalid <= 0;
469         axi_rresp  <= 0;
470       end
471     else
472       begin
473         if (axi_arready && S_AXI_ARVALID && ~axi_rvalid)
474           begin
475             // Valid read data is available at the read data bus
476             axi_rvalid <= 1'b1;
477             axi_rresp  <= 2'b0; // 'OKAY' response
478           end
479         else if (axi_rvalid && S_AXI_RREADY)
480           begin
481             // Read data is accepted by the master
482             axi_rvalid <= 1'b0;
483           end
484       end
485   end
486
487     wire [31:0] control;
488
489   // Implement memory mapped register select and read logic generation
490   // Slave register read enable is asserted when valid address is available
491   // and the slave is ready to accept the read address.
492   assign slv_reg_rden = axi_arready & S_AXI_ARVALID & ~axi_rvalid;
493   always @(*)
494   begin
495         // Address decoding for reading registers
496         case ( axi_araddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] )
497           2'h0   : reg_data_out <= slv_reg0;     // xadc
498           2'h1   : reg_data_out <= control;      //slv_reg1
499           2'h2   : reg_data_out <= slv_reg2;
500           2'h3   : reg_data_out <= slv_reg3;
501           default : reg_data_out <= 0;
502         endcase
503   end
504
505   // Output register or memory read data
506   always @( posedge S_AXI_ACLK )
507   begin
508     if ( S_AXI_ARESETN == 1'b0 )
509       begin
510         axi_rdata  <= 0;
511       end
512     else
513       begin
514         // When there is a valid read address (S_AXI_ARVALID) with
515         // acceptance of read address by the slave (axi_arready),
516         // output the read dada
517         if (slv_reg_rden)
518           begin
519             axi_rdata <= reg_data_out;     // register read data
520           end
521       end
522   end
523
524   // Add user logic here
525     Servo_Controller servo_controller(
526         .clk(S_AXI_ACLK),
527         .rst(S_AXI_ARESETN),
528         .xadc(slv_reg0[11:0]),
529         .control(control[0])
```

```
    );
  // User logic ends

  endmodule

XADC Vitis: helloworld.c

#include "xsysmon.h"
#include "xparameters.h"
#include "xstatus.h"
#include "xil_exception.h"
#include "xil_printf.h"
#include "sleep.h"

#define XPAR_AXI_XADC_0_DEVICE_ID 0

#define C_BASEADDR 0x43C00000

int main()
{
  u16 data;

  Xil_Out32(C_BASEADDR + 0x300, 0x9103);//40
  Xil_Out32(C_BASEADDR + 0x304, 0x3F0F);//41
  Xil_Out32(C_BASEADDR + 0x308, 0x0400);//42
  Xil_Out32(C_BASEADDR + 0x320, 0x800);//48

  while(1)
  {
    data=Xil_In32(C_BASEADDR + 0x20C);
    data=data>>4;
    data=data*0.244;
    xil_printf("data=%03dmv\n\r", data);

    sleep(2);
  }
  return 0;
}
`timescale 1 ps / 1 ps

module top_module(
    inout [14:0] DDR_addr,
    inout [2:0] DDR_ba,
    inout DDR_cas_n,
    inout DDR_ck_n,
    inout DDR_ck_p,
    inout DDR_cke,
    inout DDR_cs_n,
    inout [3:0] DDR_dm,
    inout [31:0] DDR_dq,
    inout [3:0] DDR_dqs_n,
    inout [3:0] DDR_dqs_p,
    inout DDR_odt,
    inout DDR_ras_n,
    inout DDR_reset_n,
    inout DDR_we_n,
    inout FIXED_IO_ddr_vrn,
    inout FIXED_IO_ddr_vrp,
    inout [53:0] FIXED_IO_mio,
    inout FIXED_IO_ps_clk,
    inout FIXED_IO_ps_porb,
    inout FIXED_IO_ps_srstb,
```

```
592     input Vp_Vn_0_v_n,
593     input Vp_Vn_0_v_p,
594     output servo_control
595 );
596 
597   wire clk;
598   wire rst;
599   wire [11:0] xadc_data;
600 
601   adc design_1_i(
602     .DDR_addr(DDR_addr),
603     .DDR_ba(DDR_ba),
604     .DDR_cas_n(DDR_cas_n),
605     .DDR_ck_n(DDR_ck_n),
606     .DDR_ck_p(DDR_ck_p),
607     .DDR_cke(DDR_cke),
608     .DDR_cs_n(DDR_cs_n),
609     .DDR_dm(DDR_dm),
610     .DDR_dq(DDR_dq),
611     .DDR_dqs_n(DDR_dqs_n),
612     .DDR_dqs_p(DDR_dqs_p),
613     .DDR_odt(DDR_odt),
614     .DDR_ras_n(DDR_ras_n),
615     .DDR_reset_n(DDR_reset_n),
616     .DDR_we_n(DDR_we_n),
617     .FIXED_IO_ddr_vrn(FIXED_IO_ddr_vrn),
618     .FIXED_IO_ddr_vrp(FIXED_IO_ddr_vrp),
619     .FIXED_IO_mio(FIXED_IO_mio),
620     .FIXED_IO_ps_clk(FIXED_IO_ps_clk),
621     .FIXED_IO_ps_porb(FIXED_IO_ps_porb),
622     .FIXED_IO_ps_srstb(FIXED_IO_ps_srstb),
623     .Vp_Vn_0_v_n(Vp_Vn_0_v_n),
624     .Vp_Vn_0_v_p(Vp_Vn_0_v_p),
625     .xadc_data(xadc_data),
626     .clk(clk),
627     .rst(rst)
628   );
629 
630   Servo_Controller servo_controller_i(
631     .clk(clk),
632     .rst(rst),
633     .xadc(xadc_data),
634     .control(servo_control)
635   );
636 
637 endmodule
```

Listing A.1: Codul in limbajul Verilog