



UNIVERSITATEA NAȚIONALĂ DE ȘTIINȚĂ ȘI TEHNOLOGIE POLITEHNICA DIN BUCUREȘTI

Facultatea de Electronică, Telecomunicații și Tehnologia
Informației

Verificare în proiectarea circuitelor

Modul: AXI GPIO

Coordonator: Vasile Costin-Emanuel

Nume student: Plămădeală Catarina-Alexandra

Grupă: 432E

CUPRINS

I.	Introducere	3
1.	Prezentare modul AXI GPIO	3
2.	Scopul lucrării	3
3.	Metodologia de testare	4
II.	Interfața AXI4-Lite: Structură, semnalizare și funcționalitate	5
1.	Descrierea interfeței	5
2.	Funcționarea semnalelor	6
3.	Procesul de handshake	8
III.	Metodologia UVM: arhitectura și fluxul de verificare	8
1.	Fluxul semnalelor în arhitectura UVM	8
2.1.	Environment	10
2.2	Agent	11
2.3	Driver	13
2.4	Monitorul	16
2.5	Scoreboard	18
IV.	Testare modul	19
1.	Testarea pinilor ca IEȘIRE	19
2.	Testarea pinilor ca INTRARE	20
3.	Testare randomizată a tuturor pinilor	22
V.	Concluzie	25

I. Introducere

1. Prezentare modul AXI GPIO

Modulul AXI GPIO (General Purpose Input/Output) este un IP (Intellectual Property) care permite comunicarea cu dispozitivele externe prin intermediul pinilor GPIO. Acesta este integrat într-un sistem bazat pe arhitectura AXI (Advanced Extensible Interface) și permite controlul și citirea semnalelor de intrare/ieșire pe pinii GPIO prin intermediul unui protocol de comunicare standardizat.

Caracteristici principale ale AXI GPIO:

1. **Configurare pinilor GPIO:** Pinii pot fi setați ca intrare sau ieșire, în funcție de nevoile aplicației. Configurarea se face prin scrierea unor valori într-un registru dedicat.
2. **Citire și scriere:** Permite citirea valorilor de pe pinii de intrare și scrierea valorilor pe pinii de ieșire.
3. **Tranzacții de tip AXI:** Utilizează semnalele de control AXI (cum ar fi ARVALID, ARREADY, RVALID, etc.) pentru a gestiona tranzacțiile de citire și scriere.
4. **Acces la registre:** Configurarea și manipularea pinilor GPIO se face prin accesul la registrele interne ale modului.

2. Scopul lucrării

Scopul acestei lucrări este de a configura și verifica funcționarea modului AXI GPIO într-un sistem bazat pe arhitectura AXI. Se urmărește testarea corectitudinii interfeței și a tranzacțiilor de

citire și scriere pe pinii GPIO, precum și asigurarea că acesta funcționează conform specificațiilor, prin testarea setării pinilor în mod intrare sau ieșire. Verificarea funcționării modului se va face prin simulare și monitorizarea comportamentului semnalelor corespunzătoare în scoreboard.

3. Metodologia de testare

În cadrul procesului de verificare a funcționalității modului, am implementat o metodologie de testare treptată (incrementală), pentru a acoperi diferite scenarii și comportamente ale semnalelor și canalelor de comunicare ale modului. Testarea a fost organizată astfel încât să verifice progresiv comportamentul semnalelor și interacțiunea acestora cu diferite condiții. Metodologia a fost structurată pe următoarele etape:

1. Testarea ieșirii:

- Primul pas: Un singur pin a fost configurat ca pin de ieșire, iar datele au fost transmise către canalul de scriere, pentru a verifica corectitudinea transferului de informații.
- Al doilea pas: Toți pinii au fost randomizați pentru a funcționa ca pini de ieșire, iar datele au fost transmise simultan prin canalul de scriere, pentru a verifica comportamentul sistemului atunci când toate ieșirile sunt active.

2. Testarea intrării:

- Primul pas: Un singur pin a fost configurat ca pin de intrare, iar datele au fost citite din canalul de citire, pentru a valida corectitudinea recepției informațiilor.
- Al doilea pas: Toți pinii au fost randomizați pentru a funcționa ca pini de intrare, iar datele au fost verificate simultan pe canalul de citire, pentru a evalua comportamentul modulelor de citire în condiții de randomizare a intrărilor.

3. Testarea intrării și ieșirii prin tristate:

- Toți pinii au fost randomizați pentru a funcționa simultan atât ca pini de intrare, cât și ca pini de ieșire prin mecanismul de tristate. În acest scenariu, datele au fost citite simultan

din ambele canale (intrare și ieșire), pentru a verifica interacțiunea corectă și prevenirea conflictelor între semnalele de intrare și ieșire.

Pentru testarea separată a pinilor, s-a folosit în prima instanță simularea (vizualizarea semnalelor) pentru a observa comportamentul acestora în diferite condiții de operare. După verificarea și validarea comportamentului în simulare, implementarea a fost testată și monitorizată folosind un monitor și un scoreboard, pentru a asigura corectitudinea și integritatea datelor în timpul testelor, precum și pentru a detecta orice abateri față de comportamentul așteptat.

II. Interfața AXI4-Lite: Structură, semnalizare și funcționalitate

1. Descrierea interfeței

AXI4-Lite este o versiune simplificată a protocolului AXI4, creată pentru scenarii în care este necesară o interfață eficientă și ușor de implementat. Câteva caracteristici și aspecte foarte specifice AXI4-Lite sunt:

1. **Nu suportă transferuri de tip burst:** Fiecare tranzacție de citire sau scriere este single-beat, simplificând implementarea.
2. **Memorie mapată:** Toate registrele perifericului sunt accesibile prin adrese unice din spațiul de memorie.
3. **Eficiență pentru registre de control:** AXI4-Lite este folosit în principal pentru configurarea și accesarea registrelor de stare/control din periferice.

2. Funcționarea semnalelor

Interfața AXI4-Lite este formată din cinci canale: Read Address, Read Data, Write Address, Write Data, Write Response. În continuare, vor fi descrise semnalele cele mai importante fiecărui canal. Primele două semnale sunt **semnale globale**.

ACLK (Clock Source)

Tip: Semnal de intrare pentru dispozitivele slave și master


Funcție: Este sursa de ceas folosită pentru sincronizarea tuturor semnalelor de pe magistrala AXI4-Lite. Interfața folosește un protocol sincron, ceea ce înseamnă că toate semnalele sunt coordonate pe baza acestui semnal (frontul crescător sau descrescător al ceasului, depinde de implementare)


ARESETN

Tip: Semnal de intrare pentru dispozitivele slave și master

Funcție: Este semnalul folosit pentru resetarea sau inițializarea magistralei AXI4-Lite. N-ul din coada semnalului semnifică faptul că semnalul este considerat activ când este în starea logică de 0. Când ARESETN este activ, toate componentele master/slave ale AXI4-Lite trebuie să își revină în starea inițială, transferurile curente sunt întrerupte, iar liniile de date și adresare revin la valorile implicite. Acest semnal este utilizat pentru a inițializa magistrala la pornire sau pentru a recupera sistemul în caz de eroare

Canalul de citire a adreselor (Read Address Channel)

 ARADDR: Semnal esențial prin care se specifică adresa la care masterul dorește să citească date de la slave

 ARREADY: Semnal prin care slave-ul indică dacă este pregătit să accepte adresa de citire și semnalele de control trimise de master prin ARADDR

Canalul de citire (Read Data Channel)

- ✚ RDATA: Semnalul prin care slave-ul trimite datele citite către master. Aceste date sunt trimise ca răspuns la o cerere de citire făcută de master prin semnalul ARADDR. În cazul AXI4-Lite, semnalul RDATA este de obicei de 32 de biți
- ✚ RRESP: Semnal folosit în protocolul AXI4-Lite pentru a semnaliza starea transferului de citire. Acest semnal indică statusul tranzacției de citire efectuate între master și slave și ajută master-ul să știe dacă transferul de date s-a realizat cu succes sau dacă a avut loc o eroare
- ✚ RVALID: Semnalul indică faptul că datele citite, transportate prin RDATA, sunt valide și gata pentru a fi citite de master
- ✚ RREADY: Semnal generat de master ce semnifică faptul că master-ul este pregătit să accepte datele citite și răspunsul la cererea de citire

Canalul de scriere a adreselor (Write Address Channel)

- ✚ AWADDR: Semnal folosit în canalul de scriere al protocolului AXI4-Lite pentru a specifica adresa la care masterul dorește să scrie date
- ✚ AWVALID: Semnal generat de la master către slave, indicând că adresa de scriere (transmisă prin AWADDR) și semnalele de control asociate sunt valide și gata pentru a fi acceptate de slave
- ✚ AWREADY: Semnal generat de slave ce semnifică faptul că poate accepta adresa de scriere și semnalele de control

Canalul de scriere a datelor (Write Data Channel)

- ✚ WDATA: Semnalul prin care masterul transmite datele de scriere către slave. În cazul AXI4-Lite, WDATA este de obicei de 32 de biți

Canalul de scriere a răspunsului (Write Response Channel)

- ✚ BRESP: Semnal ce indică starea tranzacției de scriere efectuate de master și procesată de slave

- ✚ BVALID: Semnal generat de slave pentru a semnala că răspunsul la cererea de scriere (BRESP) este valid și poate fi preluat de master
- ✚ BREADY: Semnal generat de master când este pregătit să accepte răspunsul la tranzacția de scriere generată de SLAVE

3. Procesul de handshake

În cazul protocolului AXI4-Lite, procesul de handshake se referă la schimbul de semnale între Master și Slave pentru a confirma și sincroniza operațiile de citire și scriere.

Master-ul sau Slave-ul trimite un semnal VALID pentru a indica faptul că adresa, datele sau semnalele de control sunt valide și gata de utilizare. Celălalt dispozitiv (destinația informației) răspunde cu semnalul READY, indicând faptul că poate accepta informația respectivă.

Procesul de handshake este complet atunci când ambele semnale (VALID și READY) sunt active la același moment, de obicei, pe frontul crescător al semnalului de ceas (CLK), ceea ce garantează că transferul de date a fost realizat corect.

III. Metodologia UVM: arhitectura și fluxul de verificare

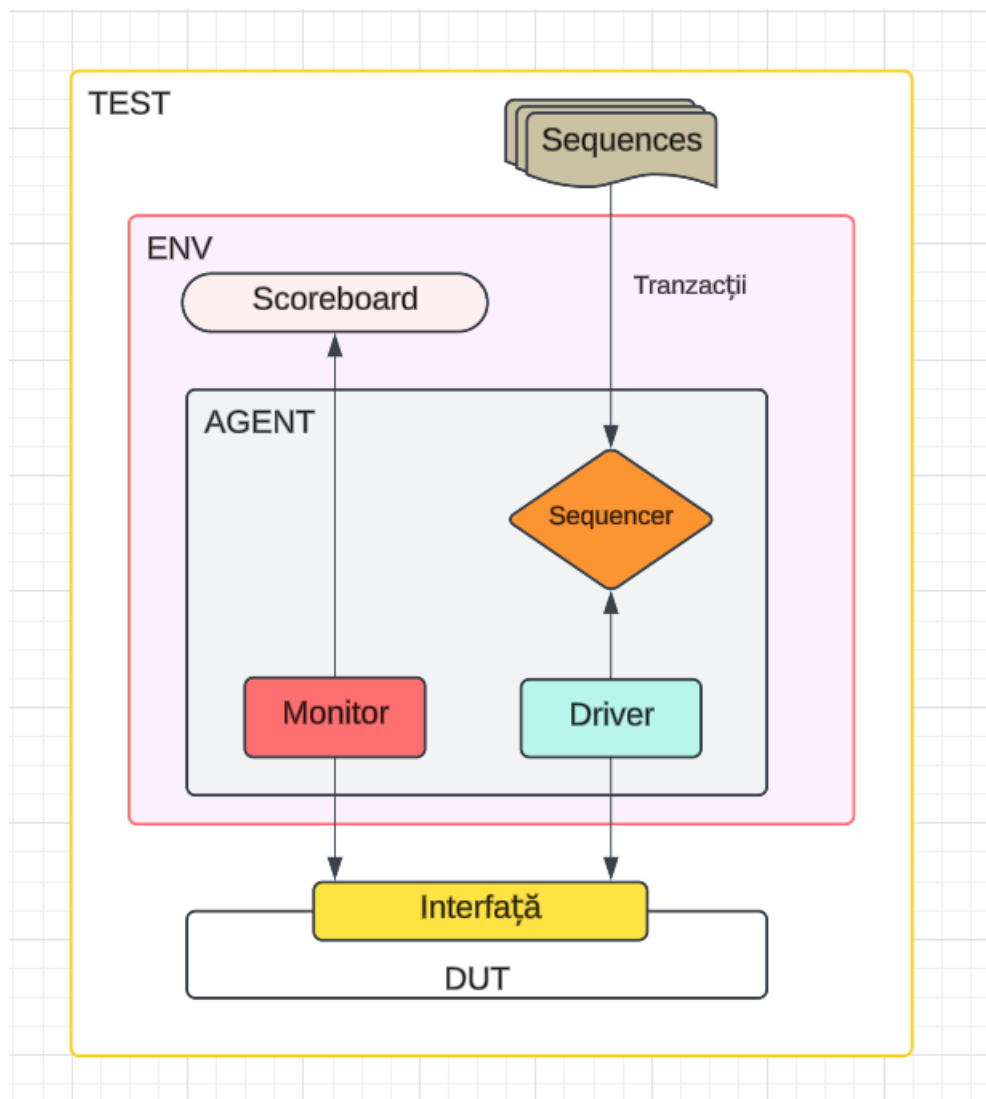
1. Fluxul semnalelor în arhitectura UVM

Universal Verification Methodology (UVM) este o abordare puternică și flexibilă pentru verificarea designurilor hardware complexe. Arhitectura sa este o orchestrare elaborată a componentelor specializate, fiecare având un rol bine definit. Totul începe cu testbench-ul UVM, care este punctul central al întregului proces. Acesta conectează Designul de Testat (DUT) la un mediu de verificare sofisticat și asigură fluxul ordonat al informațiilor între componente.

Testbench-ul include clasa UVM Test, responsabilă pentru gestionarea stimulilor și configurarea mediului de verificare. Deși testbench-ul este compilat o singură dată, el permite rularea mai

multor teste, fiecare cu o configurație și stimuli diferiți. Acest lucru oferă flexibilitate și eficiență în procesul de verificare.

În interiorul testbench-ului, mediul UVM organizează toate componentele de verificare necesare. Mediul este structurat ierarhic, astfel încât să poată grupa mai multe subcomponente, precum agenți UVM și plăci de verificare. Fiecare dintre aceste elemente are propriile responsabilități. Agenții UVM, de exemplu, sunt specializați pentru a interacționa cu o anumită interfață a DUT-ului. Ei includ trei subcomponente esențiale: secvențierul, driverul și monitorul.



Procesul începe cu generarea stimulilor în cadrul secvențelor UVM. Acestea sunt obiecte care definesc comportamentul de testare, creând tranzacții – pachete de date care reprezintă operații precum citiri sau scrieri. Secvențele trimit aceste tranzacții către secvențer, care le preia și le organizează. Secvențerul, asemănător unui dirijor, decide ordinea în care tranzacțiile sunt livrate către driver.

Driverul are o sarcină crucială: transformarea tranzacțiilor de nivel înalt în stimuli de nivel pin, care pot fi aplicați direct pe interfața DUT. Astfel, driverul face legătura între lumea abstractă a tranzacțiilor și realitatea semnalelor hardware. Odată aplicate pe DUT, răspunsurile acestuia sunt capturate de monitor.

Monitorul acționează ca un observator tăcut, preluând activitatea de pe interfața DUT și convertind semnalele brute în tranzacții analizabile. Aceste tranzacții sunt trimise mai departe către scoreboard, o componentă esențială în procesul de validare.

Scoreboardul compară tranzacțiile capturate cu rezultatele așteptate, generate de un model de referință. Acest pas este esențial pentru a determina dacă DUT-ul respectă specificațiile dorite sau dacă există discrepanțe care necesită corecții.

Astfel, fluxul semnalelor în UVM este o călătorie bine organizată. Totul pornește de la generarea stimulilor, urmează aplicarea acestora pe DUT și se încheie cu analiza rezultatelor. Această metodologie modulară și bine definită transformă un proces complex într-un mecanism eficient, capabil să asigure funcționarea corectă a celor mai sofisticate designuri hardware.

2.1. Environment

Environment-ul în UVM servește drept element central care integrează și coordonează toate componentele necesare pentru verificare, precum agenți, monitoare, scoreboards și alte module esențiale. Acesta asigură o interconectare logică și coerentă între componente, facilitând un flux de verificare structurat și eficient.

În secțiunile următoare, vor fi analizate fragmente de cod relevante pentru înțelegerea funcționării.

```

virtual function void build_phase(uvm_phase phase);
    agent = axigpio_agent::type_id::create("agent", this);
    sb = scoreboard::type_id::create("sb", this);
endfunction

virtual function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);

    agent.monitor.analysisPort.connect(sb.axigpio_imp_monitor);
endfunction

```

build_phase: Instanțiază agentul și scoreboard-ul. Acesta este locul în care componentele sunt create și pregătite pentru utilizare.

connect_phase: Realizează conexiunile necesare între componentele create, în acest caz, conectând portul de analiză al monitorului din agent la portul corespunzător din scoreboard. Acest lucru permite ca tranzacțiile monitorizate de agent să fie trimise către scoreboard pentru verificare

2.2 Agent

Agentul este o unitate de verificare care poate genera stimuli, monitoriza ieșirile și analiza rezultatele.

```

virtual function void build_phase(uvm_phase phase);
    sequencer = uvm_sequencer#(axigpio_transaction)::type_id::create("sequencer", this);
    driver = axigpio_driver::type_id::create("driver", this);
    monitor = axigpio_monitor::type_id::create("monitor", this);
endfunction

```

Blocul de cod de mai sus reprezintă implementarea funcției `build_phase` din UVM, care e folosită pentru crearea componentelor interne ale agentului. În această fază, agentul creează componente precum sequencer, driver și monitor.

`uvm_sequencer#(axigpio_transaction)` creează un sequencer specializat pentru tranzacțiile de tip `axigpio_transaction`.

`Type_id::create` este metoda standard din UVM folosită pentru instanțiere dinamică. Instanțierea dinamică înseamnă crearea unui obiect (instanță a unei clase) în timpul execuției (runtime), spre deosebire de instanțierea statică care se face direct în cod la compilare. Argumentele metodei sunt "sequencer" (numele instanței sequencerului (folosit pentru identificare în rapoarte sau debugging) și `this` (referință la componenta părinte, adică agentul care deține acest sequencer).

Analog pentru driver și monitor. După ce `build_phase` se termină, agentul are toate componentele pregătite.

```
virtual function void connect_phase(uvm_phase phase);  
    driver.seq_item_port.connect(sequencer.seq_item_export);  
endfunction
```

Porturile sunt mecanisme utilizate pentru a conecta și transmite date între diferite componente ce aparțin aceluiași testbench.

`Seq_item_port` este un port de import în driver, care permite driverului să primească tranzacții generate de sequencer.

`Seq_item_export` este un port de export al sequencerului și îi permite acestuia să trimită tranzacții către driver.

Aceste porturi sunt parte a infrastructurii UVM și sunt utilizate pentru protocolul de handshake între sequencer și driver.

Cum funcționează? Sequencer-ul generează tranzacții și le pune la dispoziție prin `seq_item_port`, iar driverul le trage prin `seq_item_export`. Acest mecanism asigură sincronizarea între generarea de tranzacții și consumarea lor.

De ce acest handshake are loc în agent? Deoarece agentul reprezintă un container care grupează toate componentele necesare pentru verificarea unui modul specific.

2.3 Driver

```
virtual axigpio_intf axigpio_interface;  
uvm_config_db#(virtual axigpio_intf)::get(null, "", "axigpio_interface", axigpio_interface);
```

Axigpio_interface reprezintă interfața virtuală care permite legarea abstractă la semnalele hardware din testbench, iar axigpio_intf este numele interfeței reale asociate DUT-ului. Următoarea linie este esențială pentru a lega componenta UVM la semnalele hardware.

uvm_config_db este o bază de date globală UVM folosită pentru configurare și transfer de informații între componente.

uvm_config_db#(virtual axigpio_intf) specifică tipul de obiect care trebuie extras din baza de date. În acest caz, este o interfață virtuală de tip axigpio_intf.

Parametrul null semnifică faptul că se caută global, iar "" înseamnă că se caută direct cu cheia.

"axigpio_interface" reprezintă numele obiectului configurat anterior (cheia), iar axigpio_interface variabila locală unde este stocată referința la interfața virtuală.

Această linie recuperează o interfață virtuală (virtual axigpio_intf) din baza de date UVM (uvm_config_db) și o atribuie variabilei locale axigpio_interface.

Configurare interfața AXI

Această secțiune de cod configurează corect funcționarea interfeței AXI și a semnalelor descrise anterior. În primul rând, se verifică dacă tranzacția este de tip scriere (writeEnable = 1). Ulterior, întregul proces se desfășoară conform următoarei structuri:

1. Masterul plasează adresa pe canalul de scriere a adresei și datele pe canalul de scriere a datelor. În același timp, setează semnalele AWVALID și WVALID la 1, indicând că adresa și datele de pe canale sunt valide.

2. Slave-ul setează semnalele AWREADY și WREADY la 1 pe canalele de scriere a adresei și a datelor.
3. Odată ce ambele semnale valid și ready sunt active pe canale, procesul de handshake se realizează, iar semnalele valid sunt resetate la 0.
4. Semnalul BVALID devine 1, indicând că slave-ul poate transmite răspunsul și că acesta este valid, iar semnalul BRESP reflectă starea tranzacției, semnalizând dacă aceasta a fost efectuată cu succes.
5. Pentru a confirma că tranzacția a avut loc, se dă un scurt impuls cu semnalul bready.

```

if(axigpio_item.writeEnable == 1) begin // write
    // Drive the address and data
    axigpio_interface.s_axi_wdata = axigpio_item.writeData;
    axigpio_interface.s_axi_awaddr = axigpio_item.addr;
    // Signal that the data and the address are valid on the bus
    axigpio_interface.s_axi_awvalid = 1;
    axigpio_interface.s_axi_wvalid = 1;

    // Wait until the consumer acknowledges the data and the address, then clear the valid signals
    wait(axigpio_interface.s_axi_awready == 1 && axigpio_interface.s_axi_wready == 1);
    @(posedge axigpio_interface.s_axi_aclk);
    axigpio_interface.s_axi_awvalid = 0;
    axigpio_interface.s_axi_wvalid = 0;

    // Wait until the consumer acknowledges the write and check the response
    wait(axigpio_interface.s_axi_bvalid == 1);
    @(posedge axigpio_interface.s_axi_aclk);
    if(axigpio_interface.s_axi_bresp == 0)
        `uvm_info("axigpio_driver", "Write access successfull", UVM_NONE)
    else
        `uvm_warning("axigpio_driver", $sprintf("The previous write access generated %0b response", axigpio_interface.s_axi_bresp))

    // Acknowledge the response by setting BREADY for 1 clock cycle
    axigpio_interface.s_axi_bready = 1;
    @(posedge axigpio_interface.s_axi_aclk);
    axigpio_interface.s_axi_bready = 0;
end

```

Activato Valid

În sens similar se întâmplă și procesul de citire.

Dacă tranzacția este de tip citire (read), atunci întregul flux se desfășoară conform următoarei structuri:

1. Masterul plasează adresa pe canalul de citire a adresei (s_axi_araddr) și semnalizează că aceasta este validă prin setarea semnalului ARVALID la 1.
2. Slave-ul răspunde setând semnalul ARREADY la 1, indicând că este pregătit să primească adresa.

3. Odată ce ARREADY este activ, semnalul ARVALID este resetat la 0, semnalizând că adresa nu mai este valabilă și că tranzacția de citire poate avansa.
4. Masterul așteaptă ca Slave-ul să semnaleze că datele de citire sunt disponibile pe magistrală, prin setarea semnalului RVALID la 1.
5. După ce RVALID devine activ, Masterul verifică răspunsul de citire (RRESP):
 - Dacă răspunsul este 0 (indică succes), se trimite un mesaj de tip info în jurnalul UVM, afișând datele citite (pentru scopuri de depanare).
 - Dacă răspunsul nu este 0, se generează un avertisment cu detalii despre eroarea apărută.
6. După procesarea răspunsului, Masterul recunoaște răspunsul citirii prin setarea semnalului RREADY la 1 pentru un ciclu de ceas.
7. După un ciclu de ceas, RREADY este resetat la 0, semnalizând că procesul de citire a fost complet și că Masterul este pregătit pentru o nouă operațiune.

```

else begin // read
    // Drive the address and signal that it is valid
    axigpio_interface.s_axi_araddr = axigpio_item.addr;
    axigpio_interface.s_axi_arvalid = 1;

    // Wait until the consumer acknowledges the address, then clear the valid signal
    wait (axigpio_interface.s_axi_arready == 1);
    @(posedge axigpio_interface.s_axi_aclk );
    axigpio_interface.s_axi_arvalid = 0;

    // Wait until the consumer signals that the read data is available on the bus
    wait (axigpio_interface.s_axi_rvalid == 1);

    // Check the response
    // If successful, print the data in the log. This is only for debugging purposes, since the monitor will capture it
    @(posedge axigpio_interface.s_axi_aclk);
    if(axigpio_interface.s_axi_rresp == 0)
        `uvm_info("axigpio_driver", $sprintf("Successfully read data %0h", axigpio_interface.s_axi_rdata), UVM_NONE)
    else
        `uvm_warning("axigpio_driver", $sprintf("The previous read access generated %0b response", axigpio_interface.s_axi_bresp))

    // Acknowledge the response by setting RREADY for 1 clock cycle
    axigpio_interface.s_axi_rready = 1;
    @(posedge axigpio_interface.s_axi_aclk);
    axigpio_interface.s_axi_rready = 0;
end

```

2.4 Monitorul

Monitorul este responsabil pentru observarea activității pe interfața AXI și capturarea tranzacțiilor care au loc. Monitorul nu controlează interfața, ci doar o urmărește. El primește răspunsuri de la interfața AXI și înregistrează aceste informații.

```
uvm_analysis_port #(axigpio_transaction) analysisPort;
```

Acest port va fi folosit pentru a trimite tranzacțiile axigpio_transaction către o componentă UVM care analizează tranzacțiile respective.

```
virtual function void build_phase(uvm_phase phase);  
    super.build_phase(phase);  
    analysisPort = new("axigpioAnalysisPort", this);  
endfunction
```

```
analysisPort = new("axigpioAnalysisPort", this);
```

- Se creează un port de analiză numit analysisPort. Acest port va fi folosit pentru a trimite informații (tranzacții) între componentele testbench-ului
- this înseamnă că portul de analiză este creat pentru componenta curentă în care se află acest cod

De ce este necesar acest cod?

Crearea portului de analiză: Codul creează un port pentru a transmite date (tranzacții). Porturile sunt importante în UVM pentru a comunica între diferite părți ale testbench-ului.

Inițializare corectă: Asigură că acest port este pregătit și gata de utilizare înainte de a începe testul, pentru a putea transmite tranzacțiile pentru verificare


```

forever begin // WRITE
    axigpio_transaction axigpio_item;
    axigpio_item = axigpio_transaction::type_id::create("axigpio_item_write");

    axigpio_item.writeEnable = 1;

    wait(axigpio_interface.s_axi_awready == 1 && axigpio_interface.s_axi_wready == 1);
    @(posedge axigpio_interface.s_axi_aclk);

    axigpio_item.addr      = axigpio_interface.s_axi_awaddr;
    axigpio_item.writeData = axigpio_interface.s_axi_wdata;

    wait(axigpio_interface.s_axi_bvalid == 1);
    @(posedge axigpio_interface.s_axi_aclk);

    `uvm_info("axigpio_monitor", $sprintf("Detected a new write response: %s", axigpio_item.convert2string())
    analysisPort.write(axigpio_item);
end

```

Acest cod monitorizează o tranzacție de scriere pe interfața AXI. El așteaptă semnalele de ready, trimite adresa și datele, apoi așteaptă confirmarea de scriere și loghează rezultatele. În final, tranzacția este transmisă prin portul de analiză pentru a fi procesată mai departe (monitorul transformă această informație în formă abstractă).

```

forever begin // READ
    axigpio_transaction axigpio_item;
    axigpio_item = axigpio_transaction::type_id::create("axigpio_item_read");

    wait(axigpio_interface.s_axi_arvalid == 1);
    @(posedge axigpio_interface.s_axi_aclk);

    axigpio_item.addr = axigpio_interface.s_axi_araddr;

    wait(axigpio_interface.s_axi_rvalid == 1);
    @(posedge axigpio_interface.s_axi_aclk);

    axigpio_item.readData = axigpio_interface.s_axi_rdata;

    `uvm_info("axigpio_monitor", $sprintf("Detected a new read response: %s", axigpio_item.convert2string())
    analysisPort.write(axigpio_item);
end

```

În mod similar se procedează și pentru tranzacțiile de scriere.

De ce fac asta și în monitor, dacă a fost deja realizată scrierea în driver?

Chiar dacă tranzacția de scriere a fost inițiată de driver, monitor-ul trebuie să capteze răspunsul de la slave-ul AXI (de exemplu, BVALID și BRESP) pentru a verifica succesul tranzacției de scriere. Monitor-ul este locul unde se verifică dacă tranzacția s-a efectuat cu succes sau nu, iar această verificare poate fi folosită pentru logare, analize ulterioare sau validări.

2.5 Scoreboard

În scoreboard se află **golden model**, care reprezintă comportamentul ideal al sistemului. În acest model, se implementează logica de referință pentru comportamentul semnalelor, iar semnalele extrase din monitor sunt comparate cu ceea ce ar trebui să facă conform modelului de aur. Astfel, scoreboard-ul validează comportamentul real al sistemului, verificând dacă semnalele simulate corespund cu așteptările definite în golden model.

Problema a fost abordată în două cazuri distincte: când se realizează o tranzacție de scriere și când se efectuează una de citire. Fiecare caz a fost verificat pentru a asigura corectitudinea transferului de date între canalele de scriere/citire și semnalele de intrare/ieșire ale sistemului.

```
if (monitorItem.writeEnable == 1) begin
    // Scriere în registrul simulat
    registerBank[monitorItem.addr / 4] = monitorItem.writeData;

    // Verifică dacă datele sunt propagate corect pe axi_gpio_o
    if (monitorItem.addr == 0) begin // Registrul de date asociat GPIO
        if (axigpio_gpio_io_o != monitorItem.writeData) begin
            `uvm_error("GPIO_ERROR", $sprintf("Mismatch on GPIO output. Expected %0h, but got %0h",
            monitorItem.writeData, axigpio_gpio_io_o));
        end else begin
            `uvm_info("GPIO_MONITOR", $sprintf("Correctly propagated data %0h to axi_gpio_o", monitorItem.writeData), UVM_LOW);
        end
    end
end
```

În cadrul testării, se urmărește ca valoarea scrisă pe canalul de scriere să fie corect reflectată și de pinii de ieșire. `monitorItem.writeData` reprezintă datele care sunt scrise și care ajung în monitor, în timp ce `axigpio_gpio_io_o` reprezintă semnalul de ieșire de pe interfață, care ar trebui să corespundă acestor date.

```
else begin
    for (int i = 0; i <= 31; i++) begin
        // Dacă pinul este configurat ca intrare în gpio_io_t
        if (axigpio_gpio_io_t[i] == 1) begin
            // Validare pentru pini configurați ca intrare
            if (axigpio_gpio_io_i[i] != monitorItem.readData[i]) begin
                `uvm_error("GPIO_INPUT_ERROR", $formatf("Mismatch on input GPIO pin %0d. Expected %0b, but got %0b", i, monitorItem.readData[i], axigpio_gpio_io_i[i]));
            end else begin
                `uvm_info("GPIO_INPUT", $formatf("Input pin %0d matched correctly with value %0b.", i, axigpio_gpio_io_i[i]), UVM_LOW);
            end
        end else begin
            // Validare pentru pini configurați ca ieșire
            if (axigpio_gpio_io_o[i] != monitorItem.readData[i]) begin
                `uvm_error("GPIO_OUTPUT_ERROR", $formatf("Mismatch on output GPIO pin %0d. Expected %0b, but got %0b", i, monitorItem.readData[i], axigpio_gpio_io_o[i]));
            end else begin
                `uvm_info("GPIO_OUTPUT", $formatf("Output pin %0d matched correctly with value %0b.", i, axigpio_gpio_io_o[i]), UVM_LOW);
            end
        end
    end
end
end
```

Pentru verificarea tranzacției de citire, se parcurg pozițiile și valorile pinilor semnalului tristate (gpio_io_t). Pentru pinii configurați ca intrare, se verifică dacă valoarea de pe pozițiile respective este aceeași între semnalul de intrare (gpio_io_i) și RDATA. Astfel, se asigură că valorile citite de pe pinii de intrare corespund corect cu datele primite de la canalul de citire. Se procedează în mod similar și pentru pinii configurați ca ieșire.

IV. Testare modul

Testarea **AXI GPIO** presupune validarea funcționalității configurării pinurilor ca intrare sau ieșire utilizând bufferul **3-state**. Pinurile sunt configurate prin intermediul registrului **GPIOx_TRI** (tri-state control), unde fiecare bit controlează starea unui pin:

- 0 = Pin configurat ca **ieșire**
- 1 = Pin configurat ca **intrare**

Prin alternarea configurării pinurilor între **output** și **input**, se validează corectitudinea comportamentului GPIO, precum și accesul la registrele de control.

Planul de Testare AXI GPIO

1. Testarea pinilor ca IEȘIRE

- **Obiectiv:** Configurarea tuturor pinilor GPIO ca **ieșire** și scrierea unei valori.
- **Pași:** Accesarea adresei 0x04 pentru configurarea GPIO în modul **output**, conform documentației și scrierea în registrul de date utilizând adresa 0x00. Verificarea prin simulare a valorii setate și a adresei corespunzătoare.

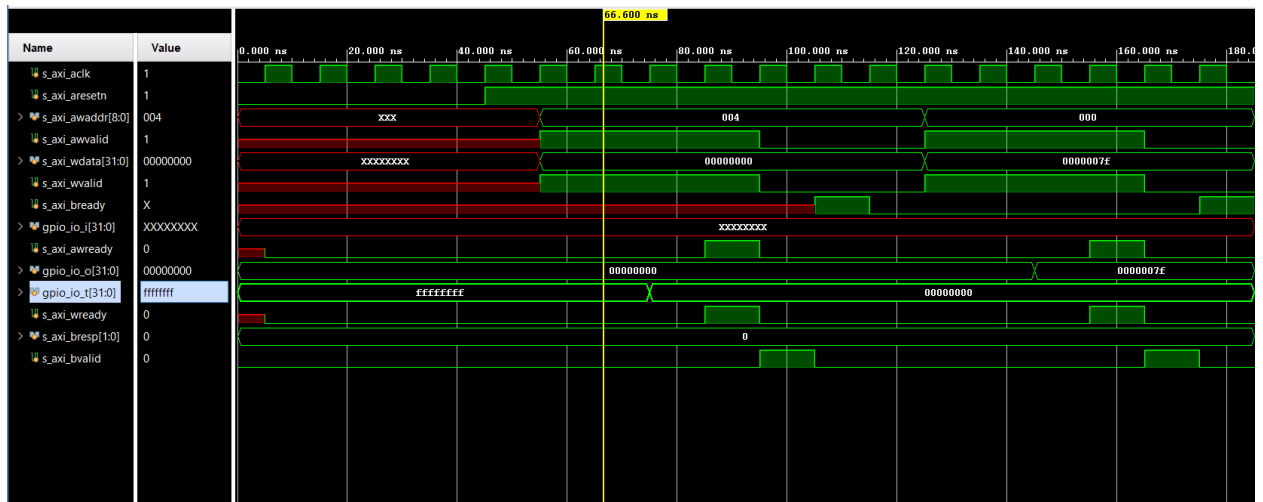
Base Address + Offset (hex)	Register Name	Access Type	Default Value (hex)	Description
C_BASEADDR + 0x00	GPIO_DATA	Read/Write	0x0	Channel 1 AXI GPIO Data Register
C_BASEADDR + 0x04	GPIO_TRI	Read/Write	0x0	Channel 1 AXI GPIO 3-state Register

În cadrul acestui test, valoarea pinului a fost setată la 127.

```

logic [31:0] gpio_io_t=0;
writeRegister(4, gpio_io_t);
writeRegister(0, 127);

```



În imaginea de mai sus se analizează rezultatul simulării. Se observă semnalul de clock și semnalul de reset, care apare la al 5-lea ciclu de clock, conform configurației. Inițial, se așteaptă validarea semnalelor de scriere a adresei și datelor (**WVALID** și **AWADDR**), urmată de validarea semnalelor de disponibilitate (**AWREADY** și **WREADY**). Ulterior, se așteaptă ca semnalul **BVALID** să devină 1, iar tranzacția este considerată încheiată după apariția unui impuls scurt al semnalului **BREADY**.

În prima fază, se observă pe canalul de scriere a adreselor valoarea **0x04**, iar pe canalul de scriere a datelor apare **'00000000'**, ceea ce indică faptul că toți biții au fost setați ca **output**. Ulterior, pe canalul de scriere a adreselor apare **0x00**, adresa utilizată pentru accesarea registrului de date, iar pe canalul de scriere a datelor apare valoarea **'0000007F'**, care, transformată în zecimal, este $7 \times 16 + 15 = 112 + 15 = 127$.

2. Testarea pinilor ca INTRARE

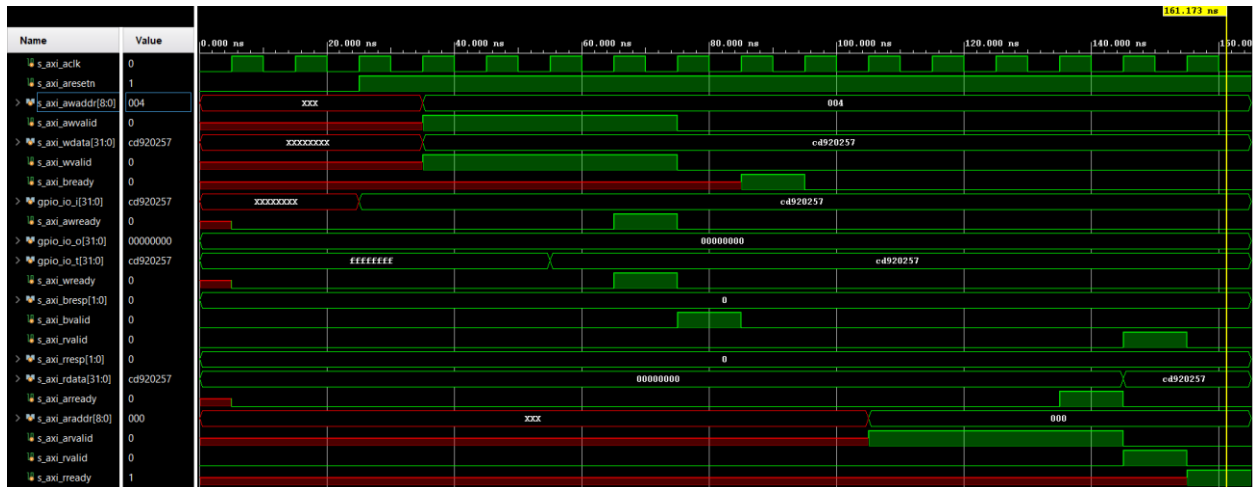
- **Obiectiv:** Configurarea pinilor GPIO ca **intrare** și citirea valorilor de la adresa respectivă prin randomizare
- **Pași:** Accesarea adresei 0x04 pentru configurarea GPIO în modul **input**, conform documentației, iar mai apoi citirea valorii de la adresa 0x00 pentru a determina

starea pinului (1 sau 0, în funcție de configurație). Verificarea, prin simulare, a valorilor citite și a adresei corespunzătoare, pentru a valida corectitudinea configurației și funcționarea pinilor ca intrare.

```
virtual task runGenerateMode();

virtual axigpio_intf axigpio;
int readValue;
uvm_config_db#(virtual axigpio_intf)::get(null, "", "axigpio_interface", axigpio);
// Randomizează toți biții din gpio_io_i
randomize(axigpio.gpio_io_i);
// Scrie valoarea pentru a seta pinul ca intrare
writeRegister(4, axigpio.gpio_io_i);
// Citește valoarea de la această adresă
readRegister(0, readValue);

endtask : runGenerateMode
```



În această imagine se poate urmări tranzacția de citire. Biții semnalului **gpio_io_i** au fost randomizați pentru a simula o configurație variabilă a pinilor. La adresa 0 (folosită doar pentru citire sau scriere), valoarea semnalului poate fi aleatorie, iar semnalul **ARVALID** este activ (1), indicând că citirea este validă. Se așteaptă ca semnalul **ARREADY** să devină 1, semn că master-ul este pregătit să preia datele. După un ciclu de ceas, semnalul **ARVALID** devine 0. În continuare, se așteaptă ca semnalul **RVALID** să devină 1, iar valoarea citită apare pe canalul de răspuns, variind în funcție de randomizarea realizată, așa cum se observă și în simulare.

De ce nu este necesar să setezi semnalul three-state (întă impedanță) atunci când pinii sunt configurați ca intrare?

Când semnalul este configurat ca intrare, semnalul va fi în stare de întă impedanță (tri-state) dacă nu este activat pentru a transmite un semnal. Acesta va copia semnalul de intrare, adică va prelua valoarea semnalului extern pe care îl primește (fie 0, fie 1). Așadar, pinul respectiv va lăsa semnalul de intrare să treacă fără a-l influența, deoarece se află în stare de întă impedanță și nu va genera o valoare proprie.

Când semnalul este configurat ca ieșire, semnalul va transmite o valoare proprie pe pin și nu va mai fi în stare de tri-state. În schimb, va furniza semnalul dorit pe linie.

3. Testare randomizată a tuturor pinilor

În acest pas al testării, se efectuează randomizarea mai multor semnale cheie pentru a valida comportamentul sistemului în condiții variate. Semnalul responsabil pentru controlul direcției pinilor (`gpio_io_t`) este generat aleatoriu pentru a acoperi diferite configurații de intrare și ieșire. De asemenea, se generează în mod aleatoriu valoarea care urmează să fie scrisă pe canalul de ieșire, simulând diverse scenarii de utilizare.

În plus, se randomizează și valorile semnalului de intrare (`gpio_io_i`), care vor fi aplicate pe interfață, pentru a verifica răspunsul corect al sistemului la diferite date de intrare. Acest proces de randomizare asigură o acoperire extinsă a scenariilor de testare, crescând șansele de a identifica eventuale probleme.

În urma testării, se verifică dacă datele scrise pe canalul de scriere se regăsesc corect pe interfața modulului, reprezentată de semnalul `gpio_io_o`. Pentru operațiunile de citire, se urmărește ca biții de pe canalul de citire să reflecte corect valorile pinilor `gpio_io_i` configurați ca intrare, conform setărilor realizate prin semnalul `gpio_io_t`.

```

}    virtual task runGenerateMode();

        virtual axigpio_intf axigpio;

        logic [31:0] gpio_io_t;    // Direcția GPIO
        int readValue;             // Valoare citită
        int writeValue;            // Valoare scrisă

        // Randomizarea datelor
        writeValue = $random();    // Randomizezi valoarea pentru scriere

        // Configurarea axigpio_intf (în cazul în care ai un astfel de obiect)
        uvm_config_db#(virtual axigpio_intf)::get(null, "", "axigpio_interface", axigpio);

        // Randomizează semnalul gpio_io_i
        axigpio.gpio_io_i = $random();

        // Scrierea în registrul TRI (pentru a configura direcția)
        gpio_io_t = $random();    // Randomizezi direcția pinilor (intrare/ieșire)
        writeRegister(4, gpio_io_t); // Scriere în registrul TRI pentru adresa 4 (pentru direcții)

        // Scrierea în registrul DATA
        writeRegister(0, writeValue); // Scrierea valorii în DATA register (pentru adresa 0)

        // Citirea valorii din DATA register
        readRegister(0, readValue); // Citirea valorii din registrul DATA pentru adresa 0

    }    endtask : runGenerateMode

```

Această verificare este mai complexă și dificil de urmărit direct prin simulare. Din acest motiv, a fost implementată în **scoreboard**, conform practicilor recomandate. Totuși, pentru scopuri didactice, am analizat și rezultatele simulării pentru a înțelege mai bine comportamentul sistemului.



În simulare, canalul de scriere al datelor poate fi urmărit destul de ușor. Inițial, acesta conține valoarea semnalului `gpio_io_t`, care configurează direcția pinilor (scriere la adresa 0x04). Ulterior, când se scrie la adresa de date 0x00, semnalul WDATA reflectă corect valoarea semnalului de ieșire, `gpio_io_o`, conform așteptărilor.

Canalul de citire al datelor este mai dificil de interpretat grafic, deoarece reprezintă o combinație între semnalele de intrare și ieșire. Conform așteptărilor, acesta ar trebui să reflecte, pe pozițiile biților configurați ca intrare prin semnalul de tristate (`gpio_io_t`), valorile corespunzătoare semnalului de intrare `gpio_io_i`.

Valoarea semnalului de tristate din simulare este 8484d609 (hexazecimal), care, transformat în binar, devine: **1000 0100 1000 0100 1101 0110 0000 1001**. Astfel, pozițiile pinilor configurați ca intrare prin semnalul de tristate sunt: 31, 26, 23, 18, 15, 14, 12, 10, 9, 3 și 0.

Semnalul RDATA, cu valoarea 92917725 (hexazecimal), transformat în binar devine: **1001 0010 1001 0001 0111 0111 0010 0101**. De interes sunt pozițiile pinilor pentru care `gpio_io_t` este configurat ca intrare.

Semnalul de intrare `gpio_io_i`, cu valoarea C0895e81 (hexazecimal), transformat în binar devine: **1100 0000 1000 1001 0101 1110 1000 0001**.

În continuare, vom compara valorile dintre RDATA și WDATA pe pozițiile configurate de `gpio_io_t` ca intrare.

<code>gpio_io_t</code> (input)	31	26	23	18	15	14	12	10	9	3	0
RDATA	1	0	1	0	0	1	1	1	1	0	1
<code>gpio_io_i</code>	1	0	1	0	0	1	1	1	1	0	1

După analizarea valorilor și compararea semnalelor, se poate concluziona că toți bitii corespunzători sunt corect configurați și valorile citite sunt conforme cu așteptările. Astfel, verificarea este validă, iar funcționalitatea semnalelor este conformă cu specificațiile.

Acest lucru este, bineînțeles, confirmat și de testarea implementat

V. Concluzie

În cadrul acestei lucrări, am realizat o testare detaliată a modului AXI GPIO utilizând metodologia UVM. Modulul s-a comportat în mod normal, conform așteptărilor, iar rezultatele obținute în urma testării au fost corecte. Totuși, procesul a fost marcat de diverse provocări. La început, am întâmpinat dificultăți în înțelegerea interfeței și în implementarea acesteia în driver, ceea ce a necesitat un timp considerabil de documentare și experimentare. Apoi, am avut de înțeles cum funcționează efectiv modulul, ceea ce a presupus o familiarizare aprofundată cu arhitectura acestuia.

Un alt obstacol semnificativ a fost debugging-ul în Vivado, care s-a dovedit destul de dificil. La început, semnalele nu apăreau pe ecran din cauza unui semnal neconectat la interfață, ceea ce împiedica implementarea corectă a codului. În plus, analiza semnalelor din simulări a fost o provocare, având în vedere complexitatea acestora și necesitatea de a corela valorile obținute cu comportamentul așteptat.

Partea teoretică a lucrării a implicat o înțelegere solidă a funcționalității UVM și a modului în care această metodologie susține dezvoltarea și verificarea modulară a sistemelor complexe. Deși unele concepte au fost dificile de înțeles la început, în cele din urmă am dobândit o înțelegere completă a acestora, iar procesul a fost extrem de educativ.

În final, am învățat multe despre testarea și verificarea sistemelor bazate pe UVM, despre utilizarea corectă a interfețelor AXI și despre importanța debugging-ului detaliat pentru identificarea problemelor. Aceste experiențe m-au ajutat să dezvolt abilități esențiale în domeniul verificării și al proiectării hardware.