Proiect de Verificare în Proiectarea Circuitelor

Angelescu Denisa Andreea

Cuprins

Introducere

Tipul de Memorie Utilizat

Acest proiect are ca obiectiv implementarea, simularea și verificarea unei memorii RAM cu un singur port, generată folosind Distributed Memory Generator de la AMD/Xilinx. Memoria este utilizată pentru stocarea și accesarea datelor, fiind testată cu Universal Verification Methodology (UVM) pentru a valida corectitudinea operațiilor de scriere, citire și reset.

Memorie RAM distribuită

- Implementată în FPGA folosind LUT-uri în loc de BRAM (Block RAM).
- Utilizată pentru aplicații cu latență mică și dimensiuni mici-medii de memorie.

Single Port RAM

- Are un singur port de acces, ceea ce înseamnă că citirea și scrierea se fac pe aceleași semnale de adresă și control.
- Nu permite acces simultan la două locații de memorie diferite (spre deosebire de Dual Port RAM).

Scopul Proiectului

Prin intermediul acestui proiect, se urmăresc următoarele obiective principale:

1. Implementarea memoriei Single Port RAM
    - Memoria este configurată cu semnalele:
        - d - Datele de scriere
        - a - Adresa de acces
        - we - Semnal de scriere
        - qspo_ce - Enable pentru citire sincronă
        - qspo_rst - Reset asincron pentru ieșirea sincronă
        - qspo_srst - Reset sincron pentru ieșirea sincronă

- clk - Semnal de ceas pentru operațiile sincrone
- spo - Ieșire asincronă
- qspo - Ieșire sincronă

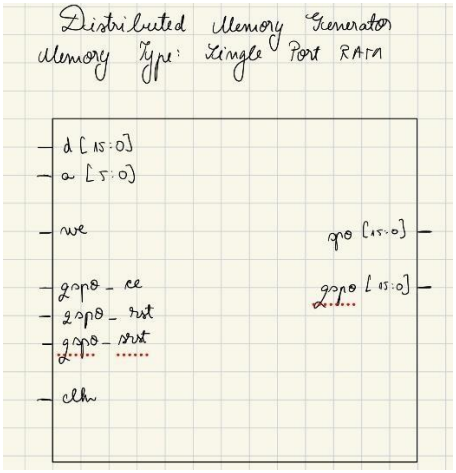

Fig. 1 PINOUT

2. Generarea corectă a memoriei cu Distributed Memory Generator
    - Alegerea configurației potrivite pentru porturi

Fig. 2 Opțiuni pentru Input

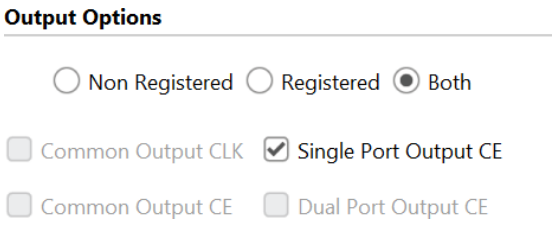

Fig. 3 Opțiuni pentru Output

    - spo (Simple Port Output - asincronă)

➢ Ieșirea se actualizează imediat când adresa de intrare (a) se schimbă.

➢ Nu necesită semnal de ceas (clk).

➢ Utilă pentru acces rapid, dar poate introduce instabilitate dacă adresa se modifică frecvent.

    - qspo (Registered Synchronous Port Output - sincronă)

➢ Ieșirea se actualizează doar la posedge clk, atunci când qspo_ce este activ.

- Dacă qspo_ce nu este activ, qspo păstrează ultima valoare citită.
- Utilizată pentru acces sincronizat, evitând erorile de competiție a datelor.
    - Definirea opțiunilor de reset și control

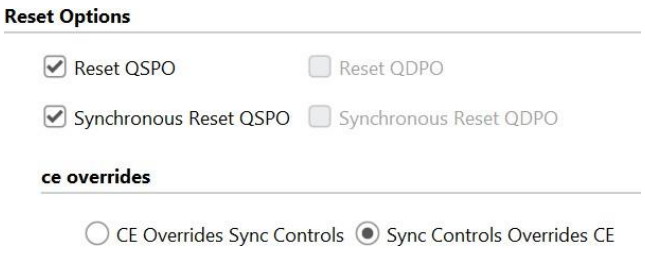

Fig. 4 Opțiuni pentru Reset

3. Verificarea corectitudinii funcționale folosind UVM
    - Testarea scrierii și citirii:
        - Asigurarea că datele scrise la o adresă pot fi citite corect.
    - Verificarea resetului:
        - Resetul asincron (qspo_rst) trebuie să șteargă ieșirea fără a depinde de ceas.
        - Resetul sincron (qspo_srst) trebuie să reseteze ieșirea la primul posedge de clock.
    - Analiza formelor de undă pentru a confirma comportamentul așteptat.

4. Depanarea și îmbunătățirea testbench-ului
    - Identificarea și remedierea problemelor întâlnite în testare.
    - Compararea rezultatelor obținute cu comportamentul așteptat.

Testarea Memoriei cu UVM

Pentru a valida funcționalitatea memoriei, se utilizează Universal Verification Methodology (UVM), o metodologie standardizată pentru testarea modulelor hardware. Testarea include:

1. Testarea scrierii și citirii
    - Se scriu date într-o anumită locație de memorie.
    - Se verifică dacă datele pot fi citite corect prin spo și qspo.
2. Testarea resetului
    - Reset asincron (qspo_rst) → trebuie să reseteze qspo imediat, indiferent de clk.
    - Reset sincron (qspo_srst) → trebuie să reseteze qspo la primul posedge clk după activare.
3. Analiza formelor de undă
    - Se compară comportamentul observat în simulare cu cel așteptat.
    - Se verifică dacă spo se modifică instantaneu și qspo doar la posedge clk.
4. Depanarea problemelor
    - Se folosesc log-uri UVM (uvm_info, uvm_error) pentru a urmări execuția testului.
    - Se analizează cazurile în care qspo sau spo nu se comportă conform specificațiilor.

Prin această testare, se asigură că memoria funcționează corect în orice scenariu, fie că este vorba de scriere, citire sincronă sau resetare.

Integrarea Memoriei în Testbench

Wrapper SystemVerilog pentru modulul dist_mem_gen_0 (VHDL)

Înainte de a începe verificarea memoriei, trebuie să ne asigurăm că aceasta poate fi utilizată corect într-un mediu de testare SystemVerilog-UVM. Deoarece Distributed Memory Generator generează automat un modul VHDL, dar testbench-ul nostru este scris în SystemVerilog, avem nevoie de un wrapper care să permită comunicarea dintre cele două limbaje.

Pentru a rezolva această problemă, am creat un wrapper SystemVerilog care înglobează modulul generat în VHDL (dist_mem_gen_0) și îl expune în mod compatibil cu restul testbench-ului nostru.

Acest wrapper definește aceleași porturi de intrare și ieșire ca și memoria și doar instanțiază modulul VHDL, transmițând semnalele mai departe. Aceasta este o metodă utilizată frecvent pentru a permite interoperabilitatea între limbaje diferite în proiectele FPGA.

Mai exact, wrapper-ul primește următoarele semnale:

- Semnale de intrare: a (adresă), d (date), clk (ceas), we (scriere), qspo_ce (enable pentru citire sincronă), qspo_rst (reset asincron), qspo_srst (reset sincron).
- Semnale de ieșire: spo (citire asincronă), qspo (citire sincronă).

După crearea acestui wrapper, putem folosi memoria exact ca un modul SystemVerilog nativ în testbench-ul nostru.

```
// Wrapper SystemVerilog pentru modulul dist_mem_gen_0 (VHDL)
module dist_mem_gen_wrapper (
    input  logic [5:0]   a,
    input  logic [15:0]  d,
    input  logic         clk,
    input  logic         we,
    input  logic         qspo_ce,
    input  logic         qspo_rst,
    input  logic         qspo_srst,
    output logic [15:0]  spo,
    output logic [15:0]  qspo
);

    // Instanțierea modulului VHDL
    dist_mem_gen_0 inst_dist_mem_gen_0 (
        .a         (a),
        .d         (d),
        .clk       (clk),
        .we        (we),
        .qspo_ce   (qspo_ce),
        .qspo_rst  (qspo_rst),
        .qspo_srst (qspo_srst),
        .spo       (spo),
        .qspo      (qspo)
    );

endmodule
```

Fig. 5

Definirea Interfeței

Pentru a simplifica conexiunile dintre testbench și DUT am definit o interfață (dist_mem_gen_intf1). Aceasta include toate semnalele necesare pentru operarea memoriei, atât de intrare, cât și de ieșire. Ea interfață va fi instanțiată în testbench și utilizată pentru comunicarea dintre componente.

Utilizarea unei interfețe are mai multe avantaje, precum: claritate în cod (în loc să conectăm fiecare semnal manual, folosim interfața ca un pachet compact), ușurință în accesare (driver-ul, monitorul și alte componente UVM pot accesa toate semnalele prin această interfață), menținere și extindere facilă (dacă memoria se modifică, trebuie doar să schimbăm interfața, fără să modificăm întregul testbench).

```
interface dist_mem_gen_intf1();
    // Input
    logic [5:0]   a;
    logic [15:0]  d;
    logic         clk;
    logic         we;
    logic         qspo_ce;
    logic         qspo_rst;
    logic         qspo_srst;

    // Output
    logic [15:0]  spo;
    logic [15:0]  qspo;
endinterface : dist_mem_gen_intf1
```

Fig. 6

Testbench SystemVerilog

Testbench-ul verifică funcționarea corectă a memoriei distribuite prin generarea de stimuli și analiza răspunsurilor acesteia. Acesta instanțiază memoria, creează semnalul de ceas și mapează semnalele printr-o interfață. Testele UVM sunt lansate prin run_test(), trimițând tranzacții de citire și scriere, iar rezultatele sunt comparate cu valorile așteptate. Pentru a evita rularea infinită, testbench-ul impune o limită de cicluri de ceas și oprește simularea dacă testul nu se finalizează în timp util.

Testbench-ul începe prin definirea unității de timp și includerea bibliotecilor UVM necesare pentru simulare.

```
`timescale 1ns / 1ps

`include "uvm_macros.svh"
`include "includes.sv"
import uvm_pkg::*;
```

Fig. 7

Definim semnalele de intrare și ieșire ale testbench-ului. Acestea vor fi utilizate pentru a comunica cu memoria instanțiată.

```
module testbench();

logic [5:0]   a;
logic [15:0]  d;
logic         clk;
logic         we;
logic         qspo_ce;
logic         qspo_rst;
logic         qspo_srst;

logic [15:0]  spo;
logic [15:0]  qspo;
```

Fig. 8

Instanțiem Distributive Memory Generator-ul printr-un wrapper, care permite utilizarea unui modul VHDL în SystemVerilog.

```
dist_mem_gen_wrapper dut_inst(.*);
```

Fig. 9

Instanțiem interfața de test, care va fi utilizată pentru a transmite semnale între componentele UVM și DUT.

```
dist_mem_gen_intf1 mem_intf();
```
Fig. 10

Generăm semnalul de ceas, care alternează la fiecare 10 ns, oferind un semnal de 50 MHz.

```
initial begin
    clk = 0;
    forever #10 clk = ~clk;
end
```
Fig. 11

Conectăm semnalele testbench-ului la interfață, astfel încât acestea să fie utilizate corect de driver, monitor și DUT. Atunci când un semnal este atribuit direct dintr-o interfață, înseamnă că valoarea lui vine din acea interfață și este preluată de modulul în care se află această instrucțiune. Cu alte cuvinte, este un semnal de **citire din interfață**, nu de scriere. Dacă un semnal din mem_intf (ex: mem_intf.spo) este atribuit cu spo, înseamnă că spo provine din modulul curent și este transmis către mem_intf. Astfel, spo și celelalte sunt semnale de ieșire din acest modul către interfața de memorie. clk este un semnal de sincronizare și, în mod uzual, trebuie distribuit către toate modulele care depind de el.

```
assign mem_intf.spo = spo;
assign mem_intf.qspo = qspo;

assign a = mem_intf.a;
assign d = mem_intf.d;
assign we = mem_intf.we;
assign qspo_ce = mem_intf.qspo_ce;
assign qspo_rst = mem_intf.qspo_rst;
assign qspo_srst = mem_intf.qspo_srst;

assign mem_intf.clk = clk;
```
Fig. 12

Salvăm interfața în baza de date UVM, astfel încât toate componentele testbench-ului să poată accesa semnalele.

```
initial begin
    uvm_config_db#(virtual dist_mem_gen_intf1)::set(null, "", "dist_mem_gen_intf1", mem_intf);
```
Fig. 13

Pornim testul UVM folosind run_test(), care va executa secvențele de scriere, citire și resetare.

```
fork
    begin
        run_test("rw_test");
    end
```

Fig. 14

Setăm o limită a ciclurilor de ceas, pentru a evita ca simularea să ruleze la infinit. Dacă testul nu s-a terminat după 100 de cicluri, simularea este oprită forțat.

```
        begin
            automatic int clkLimit = 100;

            repeat(clkLimit) @(posedge mem_intf.clk);

            `uvm_fatal(
                "SIM_END",
                $sformatf(
                    "Reached the simulation limit of %0d clk cycles",
                    clkLimit
                )
            );
        end
    join_any;
end
```

Fig. 15

Componente Principale

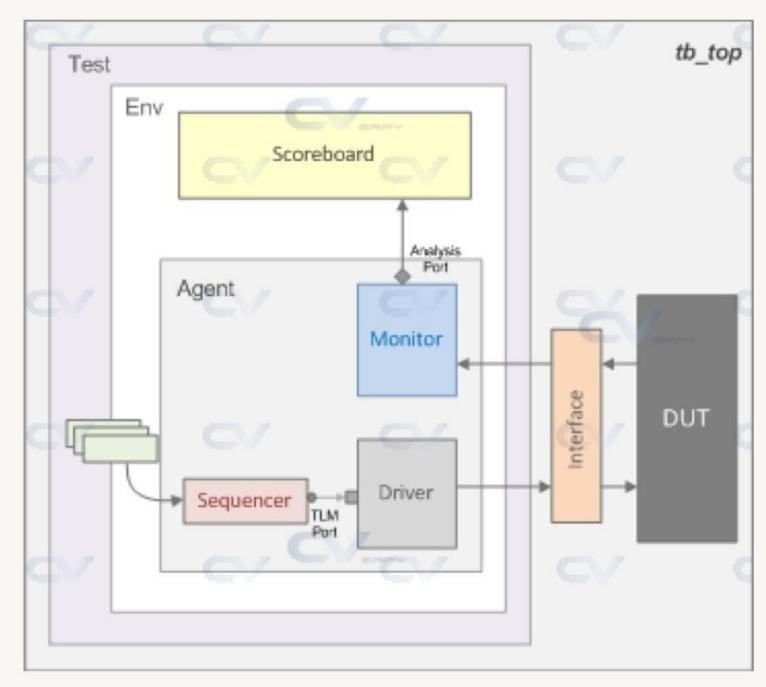

Fig. 16

Testul

Testul rw_test este responsabil de verificarea funcționalității memoriei distribuite într-un mediu de simulare UVM. Acesta validează corectitudinea operațiunilor fundamentale ale memoriei, precum scrierea, citirea sincronă și asincronă, și mecanismele de resetare. Prin generarea și gestionarea tranzacțiilor de test, rw_test se asigură că memoria reacționează corect la diferite scenarii și că datele sunt accesate corespunzător, conform specificațiilor.

Testul este construit folosind metodologia UVM și include o serie de sarcini (tasks) care trimit comenzi către memorie și colectează răspunsurile acesteia. Metodele de scriere (writeMemory), citire (readMemorySync, readMemoryAsync) și resetare (resetSync, resetAsync) sunt implementate pentru a acoperi toate modurile de operare ale memoriei. Prin run_phase, testul coordonează execuția acestor operațiuni într-un mod secvențial, incluzând verificarea rezultatelor pentru a confirma integritatea datelor.

Obiectivul principal al testului rw_test este de a detecta eventuale probleme în comportamentul memoriei și de a verifica respectarea specificațiilor de funcționare, oferind astfel o soluție robustă pentru validarea memoriei într-un sistem digital.

Testul rw_test este definit ca o clasă care extinde uvm_test. Acesta trebuie să fie înregistrat în UVM pentru a putea fi utilizat în testbench.

```
class rw_test extends uvm_test;

    `uvm_component_utils(rw_test)

    environment env;
```

Fig. 17

Liniile de mai sus declară clasa și înregistrează testul în UVM prin macro-ul uvm_component_utils(rw_test). Variabila env reprezintă mediul de testare, care conține agentul UVM și componentele sale.

În build_phase, testul creează instanța mediului de testare.

```
function new(input string name = "rw_test", uvm_component parent = null);
    super.new(name, parent);
endfunction : new

virtual function void build_phase(uvm_phase phase);
    env = environment::type_id::create("env", this);
endfunction : build_phase
```

Fig. 18

Constructorul new apelează constructorul de bază pentru uvm_test. În build_phase, se creează o instanță a clasei environment, care conține agentul și componentele sale.

Metoda writeMemory creează o tranzacție de scriere, setează câmpurile necesare și o trimite către sequencer.

```
task writeMemory(int addr, int readdata);
    dist_mem_gen_rw_sequence writeSeq;
    writeSeq = dist_mem_gen_rw_sequence::type_id::create("writeSeq");

    writeSeq.we = 1;
    writeSeq.qspo_ce = 0;
    writeSeq.qspo_rst = 0;
    writeSeq.qspo_srst = 0;
    writeSeq.a = addr;
    writeSeq.d = readdata;

    `uvm_info("WriteMemory Debug",
              $sformatf("Sending transaction: we=%0d, addr=%0h, data=%0h",
              writeSeq.we, writeSeq.a, writeSeq.d), UVM_LOW);

    writeSeq.start(env.agent.sequencer);

    `uvm_info("rw_test",
              $sformatf("Write issued: addr=%0h, data=%0h",
              addr, readdata), UVM_NONE);
endtask : writeMemory
```

Fig. 19

Aceasta creează un obiect writeSeq de tip dist_mem_gen_rw_sequence, setează semnalul we la 1 pentru a indica o scriere, atribuie adresa și datele, apoi trimite tranzacția către sequencer. Mesajele uvm_info sunt utilizate pentru a urmări tranzacțiile.

În readMemorySync, datele sunt citite folosind semnalul qspo, care este valid doar la posedge clock.

```
task readMemorySync(input int addr, output int readdata);
    dist_mem_gen_rw_sequence readSeq;
    readSeq = dist_mem_gen_rw_sequence::type_id::create("readSeq");

    readSeq.we = 0;
    readSeq.qspo_ce = 1;
    readSeq.qspo_rst = 0;
    readSeq.qspo_srst = 0;
    readSeq.a = addr;
    readSeq.d = env.agent.monitor.mem_intf.d;

    readSeq.start(env.agent.sequencer);

    readdata = env.agent.monitor.mem_intf.qspo;
endtask : readMemorySync
```

Fig. 20

Se setează we = 0 pentru a indica o citire și qspo_ce = 1 pentru a activa ieșirea sincronă qspo. După pornirea tranzacției, valoarea citită este preluată din qspo. Data este cea din trecut, își menține valoarea.

În readMemoryAsync, ieșirea spo este utilizată pentru a returna imediat valoarea corespunzătoare adresei.

```
task readMemoryAsync(input int addr, output int readdata);
    dist_mem_gen_rw_sequence readSeq;
    readSeq = dist_mem_gen_rw_sequence::type_id::create("readSeq");

    readSeq.we = 0;
    readSeq.qspo_ce = 0;
    readSeq.qspo_rst = 0;
    readSeq.qspo_srst = 0;
    readSeq.a = addr;
    readSeq.d = env.agent.monitor.mem_intf.d;

    readSeq.start(env.agent.sequencer);

    `uvm_info("rw_test", $sformatf("Async Read Initiated: addr=%0h, SPO=%0h",
        addr, env.agent.monitor.mem_intf.spo), UVM_NONE);

    readdata = env.agent.monitor.mem_intf.spo;
endtask : readMemoryAsync
```

Fig. 21

În acest caz, qspo_ce rămâne dezactivat (0), iar valoarea este returnată imediat prin spo, fără a fi necesar un posedge clock. Data este cea din trecut, își menține valoarea.

Această metodă activează qspo_srst, resetând ieșirea qspo la 0.

```
task resetSync();
    dist_mem_gen_rw_sequence resetSeq;
    resetSeq = dist_mem_gen_rw_sequence::type_id::create("resetSyncSeq");

    resetSeq.we = 0;
    resetSeq.qspo_ce = 0;
    resetSeq.qspo_rst = 0;
    resetSeq.qspo_srst = 1;
    resetSeq.a = env.agent.monitor.mem_intf.a;

    resetSeq.start(env.agent.sequencer);
endtask : resetSync
```

Fig. 22

Aceasta trimite un semnal qspo_srst = 1, iar resetarea se va realiza la următorul posedge clock. Adresa este cea din trecut, își menține valoarea.

Această metodă activează qspo_rst, ceea ce duce la resetarea imediată a qspo.

```
task resetAsync();
    dist_mem_gen_rw_sequence resetSeq;
    resetSeq = dist_mem_gen_rw_sequence::type_id::create("resetAsyncSeq");

    resetSeq.we = 0;
    resetSeq.qspo_ce = 0;
    resetSeq.qspo_rst = 1;
    resetSeq.qspo_srst = 0;
    resetSeq.a = env.agent.monitor.mem_intf.a;

    resetSeq.start(env.agent.sequencer);
endtask : resetAsync
```

Fig. 23

Resetarea are loc imediat ce qspo_rst este activat. Adresa este cea din trecut, își menține valoarea.

În metodele writeMemory și readMemory, variabila readdata are un rol diferit în funcție de operația efectuată.

- În writeMemory, readdata conține valoarea care trebuie scrisă în memorie. Aceasta este atribuită câmpului d al tranzacției writeSeq, reprezentând datele care vor fi stocate la adresa addr.
- În readMemorySync și readMemoryAsync, readdata este folosit pentru a colecta valoarea citită din memorie. În cazul citirii sincrone (qspo), aceasta este actualizată la posedge clock, iar în cazul citirii asincrone (spo), aceasta este disponibilă imediat.

În run_phase, sunt executate secvențele de scriere, citire și resetare într-un mod organizat. Testul verifică scrierea, citirea și efectul resetării asupra memoriei:

```
writeMemory(6'h10, 16'h1234);
#25
writeMemory(6'h20, 16'h5678);
```
Fig. 24

Aceste două instrucțiuni scriu valorile 0x1234 și 0x5678 la adresele 0x10 și 0x20. După fiecare scriere, testul așteaptă un anumit timp înainte de următoarea operație.

```
readMemoryAsync(6'h10, readdata);
$display("Async Read Data after Write 1: %0h", readdata);
```
Fig. 25

Această secvență citește valoarea din adresa 0x10 în mod asincron și o afișează. Deoarece citirea asincronă returnează datele imediat, readdata este actualizat instant cu valoarea citită.

```
resetSync();
#40
readMemorySync(6'h10, readdata);
$display("Sync Read Data: %0h", readdata);
```
Fig. 26

Această secvență activează resetul sincron, așteaptă 40ns, apoi încearcă să citească din adresa 0x10. După resetare, valoarea citită qspo ar trebui să fie 0 sincron cu ceasul, indicând că resetul a fost aplicat corect.

```
resetAsync();
#13
readMemorySync(6'h30, readdata);
$display("Sync Read Data: %0h", readdata);
```
Fig. 27

Aici, resetul asincron este aplicat, ceea ce ar trebui să reseteze imediat valoarea memoriei. După 13ns, testul efectuează o citire sincronă pentru a verifica dacă resetul a avut efect.

Variabila readdata este utilizată pentru a colecta și verifica datele din memorie în timpul citirii. În cadrul scrierii, aceasta conține datele care trebuie stocate. Prin combinația dintre scriere, citire și resetare, testul rw_test validează corectitudinea memoriei și comportamentul acesteia în diferite scenarii.

Sequence Items

Clasa dist_mem_gen_seq_items1 reprezintă elementul de bază al tranzacțiilor utilizate în testarea memoriei distribuite. Aceasta definește structura unei tranzacții, incluzând adresa, datele și semnalele de control necesare pentru scriere, citire și resetare.

Fiind o extensie a uvm_sequence_item, această clasă permite generarea de stimuli într-un mod abstract, facilitând interacțiunea dintre secvențele de test (sequence) și driver. Astfel, sequencer-ul poate trimite aceste obiecte de tranzacție către driver, care le va converti în semnale hardware reale aplicate memoriei.

În plus, clasa include funcționalități de inițializare și debugging, permițând verificarea valorilor tranzacțiilor printr-o metodă de conversie în string. Acest aspect este esențial pentru monitorizarea și analiza comportamentului memoriei în timpul simulării.

Scopul dist_mem_gen_seq_items1 este de a asigura un model clar și organizat pentru schimbul de date între componentele testbench-ului, contribuind astfel la verificarea funcționării corecte a memoriei distribuite.

Această clasă extinde uvm_sequence_item, ceea ce înseamnă că este utilizată pentru a transporta date între sequencer și driver. Tranzacțiile definite în această clasă sunt esențiale pentru a modela operațiile memoriei.

```
class dist_mem_gen_seq_items1 extends uvm_sequence_item;
```
Fig. 28

Această macro înregistrează clasa în cadrul UVM, permițând utilizarea sa în secvențe (uvm_sequence).

```
`uvm_object_utils(dist_mem_gen_seq_items1)
```
Fig. 29

Aceasta este zona în care sunt declarate variabilele pentru tranzacții:

- **spo și qspo** – Acestea reprezintă ieșirile memoriei, spo fiind citirea asincronă, iar qspo citirea sincronă.
- **d —** Datele care trebuie scrise în memorie. Este declarată ca rand, ceea ce înseamnă că poate fi randomizată pentru testare.
- **we** – Semnalul de scriere (Write Enable).
- **qspo_ce** – Semnal pentru activarea citirii sincronizate.

- **qspo_rst și qspo_srst** – Semnale de reset, asincron și sincron.
- **a** – Adresa memoriei unde se citește sau scrie.

```
logic [15:0] spo, qspo;
rand logic [15:0] d;
rand logic we, qspo_ce, qspo_rst, qspo_srst;
rand logic [5:0] a;
```

Fig. 30

Constructorul inițializează variabilele tranzacției cu valori implicite (0). Astfel, atunci când se creează un nou obiect de tranzacție, acesta nu conține valori nedeterminate.

```
function new(input string name="dist_mem_gen_seq_items1");
    super.new(name);

    spo = 0;
    qspo = 0;
    d = 0;
    we = 0;
    a = 0;
    qspo_ce = 0;
    qspo_rst = 0;
    qspo_srst = 0;
endfunction : new
```

Fig. 31

Această funcție returnează o reprezentare sub formă de text a tranzacției, utilă pentru debug în timpul simulării.

```
function string convert2string();
    string outputString = "";

    outputString = $sformatf("%s\n\t * spo=%0h", outputString, spo);
    outputString = $sformatf("%s\n\t * qspo=%0h", outputString, qspo);
    outputString = $sformatf("%s\n\t * a=%0h", outputString, a);
    outputString = $sformatf("%s\n\t * we=%0h", outputString, we);
    outputString = $sformatf("%s\n\t * d=%0h", outputString, d);
    outputString = $sformatf("%s\n\t * qspo_ce=%0h", outputString, qspo_ce);
    outputString = $sformatf("%s\n\t * qspo_rst=%0h", outputString, qspo_rst);
    outputString = $sformatf("%s\n\t * qspo_srst=%0h", outputString, qspo_srst);

    return outputString;
endfunction
```

Fig. 32

Sequence

Clasa dist_mem_gen_rw_sequence este o secvență UVM care generează tranzacții pentru interacțiunea cu memoria distribuită. Aceasta este responsabilă de inițializarea și trimiterea operațiunilor către sequencer, care le va direcționa către driver pentru a fi aplicate efectiv asupra memoriei. Secvența permite testarea scrierii, citirii sincrone și resetării memoriei, asigurând o verificare completă a funcționalității acesteia.

Această secvență este utilizată în cadrul testului UVM pentru a crea tranzacțiile necesare și a le livra agentului de testare. Prin definirea variabilelor care controlează operațiile memoriei, dist_mem_gen_rw_sequence facilitează execuția testelor și permite verificarea răspunsului memoriei în diferite condiții de operare. Obiectivul principal este de a modela stimuli reali care să permită validarea corectă a funcționalității memoriei distribuite.

Macro-ul uvm_object_utils(dist_mem_gen_rw_sequence) înregistrează clasa în UVM, permițând utilizarea sa în testbench și facilitând crearea de instanțe.

```
class dist_mem_gen_rw_sequence extends uvm_sequence #(dist_mem_gen_seq_items1);
```

Fig. 33

Macro-ul uvm_object_utils(dist_mem_gen_rw_sequence) înregistrează clasa în UVM, permițând utilizarea sa în testbench și facilitând crearea de instanțe.

```
`uvm_object_utils(dist_mem_gen_rw_sequence)
```

Fig. 34

Variabilele we, qspo_ce, qspo_rst, qspo_srst, a și d sunt utilizate pentru configurarea tranzacțiilor generate. we controlează scrierea în memorie, qspo_ce activează citirea sincronizată, qspo_rst și qspo_srst sunt semnale de resetare asincronă și sincronă, iar a și d reprezintă adresa și datele utilizate în operațiile asupra memoriei. Tipul bit este folosit pentru semnalele de control deoarece poate avea doar valorile 0 și 1, fiind ideal pentru logica digitală.

```
bit we;
bit qspo_ce;
bit qspo_rst;
bit qspo_srst;
int a;
int d;
```

Fig. 35

Constructorul new apelează constructorul clasei de bază și permite crearea de instanțe ale secvenței.

```
function new(input string name = "dist_mem_gen_rw_sequence");
    super.new(name);
endfunction : new
```
Fig. 36

Metoda body este responsabilă pentru generarea și trimiterea tranzacțiilor. În această metodă, se creează un obiect rw_item de tip dist_mem_gen_seq_items1, care este inițializat cu valorile setate în testul rw_test.

```
virtual task body();
    dist_mem_gen_seq_items1 rw_item;
    rw_item = dist_mem_gen_seq_items1::type_id::create("rw_item");
```
Fig. 37

Tranzacția este transmisă către sequencer prin start_item, după care sunt atribuite valorile corespunzătoare pentru adresă, date și semnalele de control.

```
start_item(rw_item);

rw_item.a = a;
rw_item.d = d;
rw_item.we = we;
rw_item.qspo_ce = qspo_ce;
rw_item.qspo_rst = qspo_rst;
rw_item.qspo_srst = qspo_srst;
```
Fig. 38

Un mesaj uvm_info afișează detalii despre tranzacție, inclusiv adresa, datele și semnalul we, permițând verificarea corectitudinii execuției.

```
`uvm_info("dist_mem_gen_rw_sequence",
          $sformatf("Before finish_item: addr=%0h, data=%0h, we=%0h",
          a, d, we), UVM_NONE);
```
Fig. 39

Tranzacția este finalizată prin finish_item, ceea ce semnalează sequencer-ului că aceasta poate fi preluată și trimisă către driver.

```
    finish_item(rw_item);
endtask : body
```
Fig. 40

Driver

Clasa dist_mem_gen_driver este responsabilă pentru preluarea tranzacțiilor din sequencer și aplicarea acestora asupra interfeței memoriei distribuite. Aceasta transformă tranzacțiile abstracte generate de secvență în stimuli reali care sunt trimiși către DUT (Device Under Test).

Driverul funcționează într-un mod continuu, așteptând tranzacții din secvență și actualizând semnalele interfeței corespunzător pentru fiecare operație. În cadrul metodei run_phase, acesta identifică tipul de tranzacție primită și acționează în consecință:

- Dacă tranzacția este o **scriere**, driverul setează adresa și datele în memorie, activează semnalul we, apoi îl dezactivează după un ciclu de ceas.
- Dacă tranzacția este o **citire sincronizată**, driverul setează adresa și activează qspo_ce, așteptând un ciclu de ceas pentru a obține valoarea citită.
- Dacă tranzacția este o **citire asincronă**, driverul setează adresa și citește imediat valoarea spo.
- Dacă tranzacția este un **reset sincron**, driverul activează qspo_srst, așteaptă un ciclu de ceas, apoi îl dezactivează.
- Dacă tranzacția este un **reset asincron**, driverul activează qspo_rst și îl dezactivează imediat după un delay minim.

La fiecare tranzacție procesată, driverul semnalează secvenței că poate trimite următoarea tranzacție, asigurând un flux continuu al execuției. În plus, acesta folosește mesaje uvm_info pentru a afișa informații despre fiecare tranzacție, facilitând procesul de debug și verificare.

Această clasă extinde uvm_driver și specifică tipul de tranzacție procesată, care este dist_mem_gen_seq_items1. Driverul este componenta UVM care preia tranzacțiile din secvență și le convertește în stimuli reali pentru DUT.

```
class dist_mem_gen_driver extends uvm_driver#(dist_mem_gen_seq_items1);
```
Fig. 41

Această macro înregistrează driverul în UVM, permițând utilizarea sa în testbench și facilitând crearea de instanțe.

```
`uvm_component_utils(dist_mem_gen_driver)
```
Fig. 42

Constructorul clasei este necesar pentru inițializarea driverului și apelarea constructorului clasei părinte.

```
function new(input string name = "", uvm_component parent = null);
    super.new(name,parent);
endfunction : new
```
Fig. 43

Metoda run_phase este responsabilă pentru rularea driverului în timpul fazei run din UVM. Aici, driverul așteaptă tranzacții, le procesează și le aplică pe interfață. Variabila dist_mem_gen_item reprezintă o tranzacție primită de la sequencer.

```
virtual task run_phase(uvm_phase phase);
    dist_mem_gen_seq_items1 dist_mem_gen_item;
```
Fig. 44

Se creează o referință la interfața de test pentru a permite driverului să controleze semnalele memoriei. Interfața este extrasă din uvm_config_db, ceea ce permite accesul la ea din alte componente.

```
virtual dist_mem_gen_intf1 mem_intf;
uvm_config_db#(virtual dist_mem_gen_intf1)::get(null, "", "dist_mem_gen_intf1", mem_intf);
```
Fig. 45

Driverul rulează într-un loop continuu, așteptând și preluând tranzacții din sequencer. Odată ce o tranzacție este primită, aceasta este stocată în dist_mem_gen_item.

```
forever begin
    seq_item_port.get_next_item(dist_mem_gen_item);
```
Fig. 46

Un mesaj de debug este afișat pentru a indica detaliile tranzacției primite.

```
`uvm_info("dist_mem_gen_driver",
        $sformatf("Received new item: %s",
        dist_mem_gen_item.convert2string()),
        UVM_NONE)
```
Fig. 47

Semnalele interfeței sunt actualizate conform valorilor din tranzacția primită, în special până la primul posedge clk care oricum le inițializează. Acest pas pregătește stimulii pentru a fi aplicați asupra DUT (înainte de orice posedge clk).

```
mem_intf.we = dist_mem_gen_item.we;
mem_intf.qspo_ce = dist_mem_gen_item.qspo_ce;
mem_intf.qspo_rst = dist_mem_gen_item.qspo_rst;
mem_intf.qspo_srst = dist_mem_gen_item.qspo_srst;
mem_intf.a = dist_mem_gen_item.a;
mem_intf.d = dist_mem_gen_item.d;
```

Fig. 48

Dacă tranzacția primită este o scriere (we == 1), driverul așteaptă un ciclu de clock, setează adresa și datele pe interfață, activează we și apoi îl dezactivează după un ciclu de clock.

```
if (dist_mem_gen_item.we == 1) begin
    @(posedge mem_intf.clk);
    `uvm_info("dist_mem_gen_driver", "CLK edge detected in driver.", UVM_NONE);

    mem_intf.d = dist_mem_gen_item.d;
    mem_intf.a = dist_mem_gen_item.a;
    mem_intf.we = 1;

    mem_intf.qspo_ce = 0;
    mem_intf.qspo_rst = 0;
    mem_intf.qspo_srst = 0;

    @(posedge mem_intf.clk);
    mem_intf.we = 0;

    `uvm_info("dist_mem_gen_driver", "Write operation completed successfully.", UVM_NONE);
end
```

Fig. 49

Dacă tranzacția primită este un reset sincron (qspo_srst == 1), driverul activează semnalul qspo_srst, așteaptă un ciclu de clock și apoi îl dezactivează.

```
else if (dist_mem_gen_item.qspo_srst == 1) begin
    @(posedge mem_intf.clk);
    mem_intf.qspo_srst = 1;
    @(posedge mem_intf.clk);
    mem_intf.qspo_srst = 0;

    `uvm_info("dist_mem_gen_driver", "Synchronous reset applied to QSPO.", UVM_NONE);
end
```

Fig. 50

Dacă tranzacția este un reset asincron (qspo_rst == 1), driverul activează qspo_rst, așteaptă un delay minim și apoi îl dezactivează imediat.

```
else if (dist_mem_gen_item.qspo_rst == 1) begin
    mem_intf.qspo_rst = 1;
    #1;
    mem_intf.qspo_rst = 0;

    `uvm_info("dist_mem_gen_driver", "Asynchronous reset applied to QSPO.", UVM_NONE);
end
```

Fig. 51

Dacă tranzacția primită este o citire sincronizată (qspo_ce == 1), driverul setează adresa, activează qspo_ce, așteaptă un ciclu de clock și stochează valoarea citită în qspo și dezactivează qspo_ce.

```
else if (dist_mem_gen_item.qspo_ce == 1) begin
    @(posedge mem_intf.clk);
    mem_intf.a = dist_mem_gen_item.a;
    mem_intf.qspo_ce = 1;

    @(posedge mem_intf.clk);
    dist_mem_gen_item.qspo = mem_intf.qspo;

    mem_intf.qspo_ce = 0;

    `uvm_info("dist_mem_gen_driver",
              $sformatf("Synchronous read data: %0h",
              dist_mem_gen_item.qspo),
              UVM_NONE);
end
```

Fig. 52

Dacă tranzacția este o citire asincronă, driverul setează adresa și citește imediat valoarea spo de la ieșirea memoriei.

```
else begin
    mem_intf.a = dist_mem_gen_item.a;
    dist_mem_gen_item.spo = mem_intf.spo;

    `uvm_info("dist_mem_gen_driver",
              $sformatf("Asynchronous read data: %0h",
              dist_mem_gen_item.spo),
              UVM_NONE);
end
```

Fig. 53

După procesarea tranzacției, driverul semnalează sequencer-ului că a finalizat tranzacția, astfel încât următoarea să poată fi trimisă.

```
seq_item_port.item_done();
```
Fig. 54

Agent

Clasa dist_mem_gen_agent reprezintă un agent UVM care grupează componentele esențiale pentru interacțiunea cu DUT: driver, sequencer și monitor. Agentul facilitează comunicarea între aceste componente și gestionează fluxul de tranzacții dintre secvență și dispozitivul testat.

În cadrul metodei build_phase, sunt create instanțele pentru sequencer, driver și monitor, asigurând astfel structura completă a agentului. sequencer este responsabil pentru generarea tranzacțiilor și trimiterea acestora către driver, care le aplică asupra DUT. monitor observă semnalele DUT și colectează datele pentru analiză.

Metoda connect_phase stabilește conexiunile dintre driver și sequencer, permițând trimiterea tranzacțiilor către DUT. Aceasta se asigură că seq_item_port al driver este conectat la seq_item_export al sequencer, permițând preluarea și execuția tranzacțiilor generate. Agentul joacă un rol crucial în organizarea testbench-ului UVM, oferind o interfață clară între generarea de stimuli și verificarea comportamentului DUT.

Clasa moștenește uvm_agent, ceea ce înseamnă că acționează ca un container pentru componentele de bază ale unui testbench UVM.

```
class dist_mem_gen_agent extends uvm_agent;
```

Fig. 55

Această macro înregistrează clasa în UVM, permițând crearea de instanțe și utilizarea sa în testbench.

```
`uvm_component_utils(dist_mem_gen_agent)
```

Fig. 56

Aici sunt declarate componentele agentului. monitor este responsabil pentru captarea și analizarea semnalelor DUT, driver aplică tranzacțiile asupra DUT, iar sequencer generează și trimite tranzacții către driver.

```
dist_mem_gen_monitor monitor;
dist_mem_gen_driver driver;
uvm_sequencer #(dist_mem_gen_seq_items1) sequencer;
```

Fig. 57

Constructorul new apelează constructorul clasei părinte și permite crearea de instanțe ale agentului.

```
function new(input string name = "", uvm_component parent = null);
    super.new(name, parent);
endfunction : new
```
Fig. 58

Metoda build_phase inițializează componentele agentului. Se creează instanțe pentru sequencer, driver și monitor, asigurându-se că acestea sunt disponibile pentru test.

```
virtual function void build_phase(uvm_phase phase);
    sequencer = uvm_sequencer#(dist_mem_gen_seq_items1)::type_id::create("sequencer", this);
    driver    = dist_mem_gen_driver::type_id::create("driver", this);
    monitor   = dist_mem_gen_monitor::type_id::create("monitor", this);
endfunction
```
Fig. 59

Metoda connect_phase este utilizată pentru a conecta componentele agentului. Se afișează un mesaj de debug care indică faptul că conexiunile au fost stabilite.

```
virtual function void connect_phase(uvm_phase phase);
    `uvm_info("AGENT_DEBUG",
              "Connections between driver, monitor and sequencer established.",
              UVM_LOW);
```
Fig. 60

Această linie de cod conectează seq_item_port al driver la seq_item_export al sequencer. Astfel, tranzacțiile generate de sequencer sunt transmise către driver, care le aplică pe DUT.

```
driver.seq_item_port.connect(sequencer.seq_item_export);
```
Fig. 61

Environment

Clasa environment reprezintă componenta UVM care grupează toate elementele esențiale pentru testare, inclusiv agent și scoreboard. Aceasta asigură organizarea ierarhică a testbench-ului și gestionează fluxul de date între componentele sale.

Agentul (dist_mem_gen_agent) este responsabil pentru generarea, aplicarea și monitorizarea tranzacțiilor asupra DUT, iar scoreboard se ocupă cu verificarea corectitudinii rezultatelor.

În metoda build_phase, sunt create instanțele agentului și scoreboard-ului, asigurând structura testbench-ului. Metoda connect_phase realizează conexiunea între monitor-ul agentului și scoreboard, permițând verificarea datelor colectate în timpul testării.

Clasa environment servește drept nivel de integrare pentru componentele de testare, facilitând verificarea completă a DUT prin gestionarea fluxului de date între generatorul de stimuli, monitor și verificator.

Clasa environment extinde uvm_env, ceea ce înseamnă că acționează ca un container pentru componentele de testare, cum ar fi agentul și scoreboard.

```
class environment extends uvm_env;
```
Fig. 62

Această macro permite înregistrarea clasei în UVM, facilitând crearea de instanțe și utilizarea sa în testbench.

```
`uvm_component_utils(environment)
```
Fig. 63

Aici sunt declarate componentele mediului de testare. agent este responsabil pentru generarea, aplicarea și monitorizarea tranzacțiilor asupra DUT, iar sb (scoreboard) verifică dacă datele obținute corespund rezultatelor așteptate.

```
dist_mem_gen_agent agent;
scoreboard sb;
```
Fig. 64

Aceasta este funcția constructor, care apelează constructorul clasei de bază (uvm_env) și permite inițializarea instanțelor de environment.

```
function new(input string name = "env", uvm_component parent = null);
    super.new(name, parent);
endfunction
```
Fig. 65

În build_phase, sunt create instanțele agentului și scoreboard-ului. build_phase este utilizată pentru inițializarea componentelor UVM înainte de rularea testului. Se afișează un mesaj de debug pentru a confirma apelarea acestei metode.

```
virtual function void build_phase(uvm_phase phase);
    `uvm_info("DEBUG", "The build_phase of the environment was called", UVM_NONE)

    agent = dist_mem_gen_agent::type_id::create("agent", this);
    sb = scoreboard::type_id::create("sb", this);
endfunction
```
Fig. 66

Metoda connect_phase realizează conexiunea dintre monitor-ul agentului și scoreboard, permițând transferul datelor capturate în monitor către scoreboard pentru verificare. Aceasta asigură că informațiile colectate sunt analizate corect și comparate cu rezultatele așteptate.

```
virtual function void connect_phase(uvm_phase phase);
    agent.monitor.monitor_analysis_port.connect(sb.monitor_analysis_port);
endfunction : connect_phase
```
Fig. 67

Monitor

Clasa dist_mem_gen_monitor capturează și analizează tranzacțiile din timpul testării memoriei, observând semnalele DUT și trimițându-le către scoreboard pentru verificare. Include un covergroup pentru măsurarea acoperirii testării și folosește monitor_analysis_port pentru transmiterea tranzacțiilor.

În build_phase, inițializează instanțele necesare, iar în connect_phase, asociază interfața virtuală pentru acces la semnale. run_phase pornește monitorizarea operațiunilor de scriere, citire și resetare. Metodele dedicate detectează aceste operațiuni și trimit datele către scoreboard.

La final, report_phase afișează rezultatele testării și nivelul de acoperire obținut.

Acest covergroup colectează informații despre tranzacțiile testate, cum ar fi adresele accesate, datele scrise, activarea semnalelor de scriere și citire. Într-o etapă ulterioară, aceste date vor fi utilizate pentru a îmbunătăți testarea și pentru randomizarea stimulilor astfel încât toate scenariile relevante să fie acoperite.

```
covergroup cg_transactions(ref logic [5:0] a, ref logic [15:0] d, ref logic we,
                           ref logic qspo_ce, ref logic qspo_rst, ref logic qspo_srst);
    coverpoint a { bins all_addresses = {[0:63]}; }
    coverpoint d { bins all_data = {[16'h0:16'hFFFF]}; }
    coverpoint we { bins write_enable = {0, 1}; }
    coverpoint qspo_ce { bins sync_read_enable = {0, 1}; }
    coverpoint qspo_rst { bins async_reset = {0, 1}; }
    coverpoint qspo_srst { bins sync_reset = {0, 1}; }
    cross we, a;
endgroup
```

Fig. 68

Aceasta definește monitorul și include un port de analiză (monitor_analysis_port) pentru a trimite tranzacțiile observate către scoreboard. De asemenea, instanțiază covergroup pentru a colecta date despre testare.

```
class dist_mem_gen_monitor extends uvm_monitor;

    `uvm_component_utils(dist_mem_gen_monitor)

    uvm_analysis_port #(dist_mem_gen_seq_items1) monitor_analysis_port;

    cg_transactions my_cg;

    virtual dist_mem_gen_intf1 mem_intf;
```

Fig. 69

Constructorul clasei inițializează monitorul și permite integrarea sa în ierarhia UVM.

```
function new(input string name = "", uvm_component parent = null);
    super.new(name, parent);
endfunction : new
```
Fig. 70

În această etapă, este creat portul de analiză și se inițializează instanța covergroup, care va colecta datele în timpul rulării testului.

```
virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    monitor_analysis_port = new("monitor_analysis_port", this);
    my_cg = new(mem_intf.a, mem_intf.d, mem_intf.we,
                mem_intf.qspo_ce, mem_intf.qspo_rst,
                mem_intf.qspo_srst);
endfunction
```
Fig. 71

Această funcție asociază monitorul cu interfața virtuală a DUT-ului, permițând astfel accesul la semnalele acestuia.

```
virtual function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    uvm_config_db#(virtual dist_mem_gen_intf1)::get(this, "",
        "dist_mem_gen_intf1", mem_intf);
endfunction
```
Fig. 72

Monitorul rulează în paralel mai multe procese pentru capturarea diferitelor tipuri de tranzacții: scriere, resetare, citire sincronă și citire asincronă.

```
virtual task run_phase(uvm_phase phase);
    fork
        monitor_write();
        monitor_resets();
        monitor_sync_read();
        monitor_async_read();
    join
endtask : run_phase
```
Fig. 73

Această metodă detectează scrierile în memorie (sincronizate cu posedge clk) și le transmite către scoreboard. În același timp, înregistrează datele în covergroup, care va fi folosit ulterior pentru îmbunătățirea testării.

```
task monitor_write();
    forever begin
        dist_mem_gen_seq_items1 observed_item;
        repeat(1)@(posedge mem_intf.clk);
        if (mem_intf.we == 1) begin
            observed_item = dist_mem_gen_seq_items1::type_id::create(
                "observed_item_write");
            observed_item.we = mem_intf.we;
            observed_item.d = mem_intf.d;
            observed_item.a = mem_intf.a;

            `uvm_info("dist_mem_gen_monitor",
                $sformatf("Write operation observed: Address=%0h, Data=%0h",
                observed_item.a, observed_item.d), UVM_NONE);

            my_cg.sample();
            monitor_analysis_port.write(observed_item);
        end
    end
endtask : monitor_write
```

Fig. 74

Aceasta capturează operațiile de citire sincronă (când este activă citirea sincronă -> qspo_ce = 1), trimite tranzacțiile observate la scoreboard și le înregistrează în covergroup.

```
task monitor_sync_read();
    forever begin
        dist_mem_gen_seq_items1 observed_item;
        @(posedge mem_intf.clk);
        if (mem_intf.qspo_ce) begin
            #5
            observed_item = dist_mem_gen_seq_items1::type_id::create(
                "observed_item_sync_read");
            observed_item.qspo_ce = mem_intf.qspo_ce;
            observed_item.a = mem_intf.a;
            observed_item.qspo = mem_intf.qspo;

            `uvm_info("dist_mem_gen_monitor",
                $sformatf("Synchronous read operation observed: Address=%0h, Data=%0h",
                observed_item.a, observed_item.qspo), UVM_NONE);

            my_cg.sample();
            monitor_analysis_port.write(observed_item);
        end
    end
endtask : monitor_sync_read
```

Fig. 75

Monitorizează citirile asincrone (nu există nicio constrângere aici, decât că se modifică la schimbarea adresei sau a ieșirii din DUT) și înregistrează tranzacțiile observate pentru a verifica dacă testele acoperă toate scenariile necesare.

```
task monitor_async_read();
    dist_mem_gen_seq_items1 observed_item;
    forever begin
        @(mem_intf.a or mem_intf.spo);
        observed_item = dist_mem_gen_seq_items1::type_id::create(
            "observed_item_async_read");
        observed_item.a = mem_intf.a;
        observed_item.spo = mem_intf.spo;

        `uvm_info("dist_mem_gen_monitor",
            $sformatf("Asynchronous read operation observed: Address=%0h, Data=%0h",
            observed_item.a, observed_item.spo), UVM_NONE);

        my_cg.sample();
        monitor_analysis_port.write(observed_item);
    end
endtask : monitor_async_read
```

Fig. 76

Monitorizează semnalele de reset (se așteaptă un posedge pentru qspo_rst si qspo_srst; dacă qspo_rst = 1, atunci resetul este activat imediat; dacă qspo_srst = 1, atunci se așteaptă un posedge clk pentru ca resetul sa fie activat) și asigură că acestea sunt detectate corect în cadrul testului.

```
task monitor_resets();
    dist_mem_gen_seq_items1 observed_item;
    forever begin
        @(posedge mem_intf.qspo_rst or posedge mem_intf.qspo_srst);
        observed_item = dist_mem_gen_seq_items1::type_id::create(
            "observed_item_reset");
        if (mem_intf.qspo_rst == 1) begin
            `uvm_info("dist_mem_gen_monitor", "Asynchronous reset observed.", UVM_NONE);
        end
        if (mem_intf.qspo_srst == 1) begin
            @(posedge mem_intf.clk);
            `uvm_info("dist_mem_gen_monitor", "Synchronous reset observed.", UVM_NONE);
        end

        my_cg.sample();
        monitor_analysis_port.write(observed_item);
    end
endtask : monitor_resets
```

Fig. 77

La finalul testului, se raportează procentajul de acoperire colectat de covergroup, ceea ce ajută la analiza calității testării și identificarea cazurilor neacoperite.

```
virtual function void report_phase(uvm_phase phase);
    super.report_phase(phase);
    $display("Coverage Results in Monitor:");
    `uvm_info("COVERAGE_INFO",
        $sformatf("Coverage on my_cg=%d", my_cg.get_inst_coverage()),
        UVM_NONE);
endfunction : report_phase
```

Fig. 78

Scoreboard

Clasa scoreboard verifică corectitudinea tranzacțiilor din memorie. Aceasta primește tranzacțiile de la monitor prin monitor_analysis_port. În build_phase, inițializează interfața virtuală și registrele de memorie. În run_phase, rulează procese paralele pentru verificarea operațiunilor.

Metoda write_monitor actualizează memoria la fiecare scriere detectată. Task-ul verifySyncRead validează citirile sincronizate prin compararea ieșirii qspo cu valoarea așteptată. Task-ul verifyAsyncRead verifică citirile asincronizate prin compararea ieșirii spo cu datele stocate.

Task-ul verifyResets monitorizează reseturile asincrone și sincronizate. Erorile sunt raportate prin uvm_error, iar operațiunile corecte sunt confirmate prin uvm_info. Scoreboard-ul asigură validarea corectă a tranzacțiilor și detectarea erorilor în memorie.

include "uvm_macros.svh" importă macrocomenzi UVM necesare pentru mesaje de debug și erori. import uvm_pkg::*; importă pachetul UVM pentru utilizarea claselor și funcțiilor specifice verificării. `uvm_analysis_imp_decl(_monitor) creează un analysis port pentru primirea tranzacțiilor de la monitor.

```
`include "uvm_macros.svh"
import uvm_pkg::*;
`uvm_analysis_imp_decl(_monitor)
```
Fig. 79

Clasa scoreboard extinde uvm_scoreboard, ceea ce o face compatibilă cu infrastructura UVM. uvm_component_utils(scoreboard) înregistrează clasa în baza de date UVM pentru a permite crearea dinamică.

```
class scoreboard extends uvm_scoreboard;

    `uvm_component_utils(scoreboard)
```
Fig. 80

Variabila mem_intf este o interfață virtuală utilizată pentru accesul la semnalele DUT. registerBank este o bancă de 64 de registre, fiecare având 16 biți, folosită pentru stocarea valorilor scrise în memorie. monitor_analysis_port primește tranzacțiile din monitor și le trimite spre verificare.

```
virtual dist_mem_gen_intf1 mem_intf;
logic [15:0] registerBank [0:63];
uvm_analysis_imp_monitor#(dist_mem_gen_seq_items1, scoreboard) monitor_analysis_port;
```
Fig. 81

Funcția new inițializează obiectul scoreboard și apelează constructorul de bază al uvm_scoreboard. Aceasta permite crearea dinamică a instanței scoreboard în timpul testării.

```
function new(input string name = "", uvm_component parent = null);
    super.new(name, parent);
endfunction : new
```

Fig. 82

build_phase este responsabilă pentru inițializarea variabilelor și componentelor necesare. monitor_analysis_port este creat pentru a primi tranzacții din monitor. uvm_config_db::get(...) asigură legătura între scoreboard și interfața virtuală mem_intf. Banca de registre este inițializată cu valori de 0 pentru a asigura un început curat al testării.

```
virtual function void build_phase(uvm_phase phase);
    monitor_analysis_port = new("monitor_analysis_port", this);
    uvm_config_db#(virtual dist_mem_gen_intf1)::get(null, "", "dist_mem_gen_intf1", mem_intf);
    for (int i = 0; i < 64; i++) registerBank[i] = 0;
endfunction
```

Fig. 83

Funcția write_monitor este apelată automat atunci când monitorul detectează o tranzacție de scriere. Aceasta verifică dacă semnalul we este activ (we == 1), ceea ce indică o scriere validă.

Se folosesc asertii pentru validarea datelor:

- Prima aserție verifică dacă adresa a este în intervalul valid [0, 63].
- A doua aserție verifică dacă valoarea d se încadrează în intervalul 0 - 0xFFFF.

Dacă o adresă sau o valoare este invalidă, se generează un mesaj de eroare cu uvm_error. Dacă tranzacția este validă, valoarea este stocată în registerBank la adresa specificată (registerBank[monitor_item.a] = monitor_item.d). Un mesaj uvm_info confirmă scrierea în registerBank.

```
virtual function void write_monitor(dist_mem_gen_seq_items1 monitor_item);
    if (monitor_item.we == 1) begin
        assert(monitor_item.a >= 0 && monitor_item.a < 64)
            else `uvm_error("scoreboard",
                $sformatf("Invalid write address: %0h", monitor_item.a));

        assert(monitor_item.d >= 0 && monitor_item.d <= 16'hFFFF)
            else `uvm_error("scoreboard",
                $sformatf("Invalid write data: %0h", monitor_item.d));

        registerBank[monitor_item.a] = monitor_item.d;
        `uvm_info("scoreboard",
            $sformatf("Write detected: addr=%0h, data=%0h",
                monitor_item.a, monitor_item.d),
            UVM_NONE);
    end
endfunction : write_monitor
```

Fig. 84

Task-ul verifySyncRead verifică dacă citirile sincronizate funcționează corect. La fiecare front pozitiv al ceasului (clk), se verifică dacă qspo_ce este activ (qspo_ce == 1).

Dacă în același ciclu de ceas se face și o scriere (we == 1), citirea este ignorată pentru a evita erori. Dacă scrierea nu este activă, se așteaptă încă un ciclu de ceas pentru propagarea valorilor (@(posedge clk)).

Se compară apoi valoarea stocată în registerBank la adresa a cu valoarea returnată pe qspo. Dacă valorile nu coincid, se generează un mesaj uvm_error. Dacă valorile sunt identice, se afișează un mesaj uvm_info care confirmă citirea corectă.

```
task verifySyncRead();
    int expected_data;

    forever begin
        @(posedge mem_intf.clk);

        if (mem_intf.qspo_ce == 1) begin
            if (mem_intf.we == 1) begin
                `uvm_info("scoreboard",
                    $sformatf("Sync read ignored: write in progress at addr=%0h",
                    mem_intf.a), UVM_LOW);
                continue;
            end

            @(posedge mem_intf.clk);
            expected_data = registerBank[mem_intf.a];

            if (mem_intf.qspo !== expected_data) begin
                `uvm_error("scoreboard",
                    $sformatf("Sync Read Mismatch: addr=%0h, expected=%0h, got=%0h",
                    mem_intf.a, expected_data, mem_intf.qspo));
            end else begin
                `uvm_info("scoreboard",
                    $sformatf("Sync Read Match: addr=%0h, value=%0h",
                    mem_intf.a, expected_data),
                    UVM_NONE);
            end
        end
    end
endtask : verifySyncRead
```

Fig. 85

Task-ul verifyResets monitorizează semnalele de reset și verifică dacă memoria revine la starea inițială (0). Dacă qspo_rst (reset asincron) devine activ (1), se verifică dacă qspo a fost resetat la 0. Dacă valoarea nu este 0, se generează un uvm_error.

Dacă qspo_srst (reset sincron) devine activ (1), se așteaptă un ciclu de ceas (@(posedge clk)) înainte de verificare. Se verifică dacă qspo == 0, iar în caz contrar, se generează un mesaj de eroare.

```
task verifyResets();
    forever begin
        @(posedge mem_intf.qspo_rst or posedge mem_intf.qspo_srst);

        if (mem_intf.qspo_rst == 1) begin
            #1;
            if (mem_intf.qspo != 0) begin
                `uvm_error("scoreboard", "Asynchronous reset failed");
            end else begin
                `uvm_info("scoreboard", "Asynchronous reset successful", UVM_NONE);
            end
        end

        if (mem_intf.qspo_srst == 1) begin
            @(posedge mem_intf.clk);
            #1;
            if (mem_intf.qspo != 0) begin
                `uvm_error("scoreboard", "Synchronous reset failed");
            end else begin
                `uvm_info("scoreboard", "Synchronous reset successful", UVM_NONE);
            end
        end
    end
endtask : verifyResets
```
Fig. 86

run_phase pornește task-urile de verificare (verifyResets, verifySyncRead, verifyAsyncRead) în paralel, folosind fork...join. Se utilizează raise_objection pentru a informa UVM că testul este în desfășurare și nu trebuie încheiat prematur. După finalizarea verificărilor, se apelează drop_objection, semnalând că testul poate fi finalizat.

```
virtual task run_phase(uvm_phase phase);
    phase.raise_objection(this);

    fork
        verifyResets();
        verifySyncRead();
        verifyAsyncRead();
    join

    phase.drop_objection(this);
endtask
```
Fig. 87

Forma de Undă

Scriere

```
writeMemory(6'h10, 16'h1234);
#25
writeMemory(6'h20, 16'h5678);
#55
writeMemory(6'h30, 16'h9ABC);
repeat(4) @(posedge env.agent.monitor.mem_intf.clk);
writeMemory(6'h2, 16'h3);
#15
writeMemory(6'h20, 16'h3333);
#15
writeMemory(6'h30, 16'h0101);
#15
```

Fig. 79

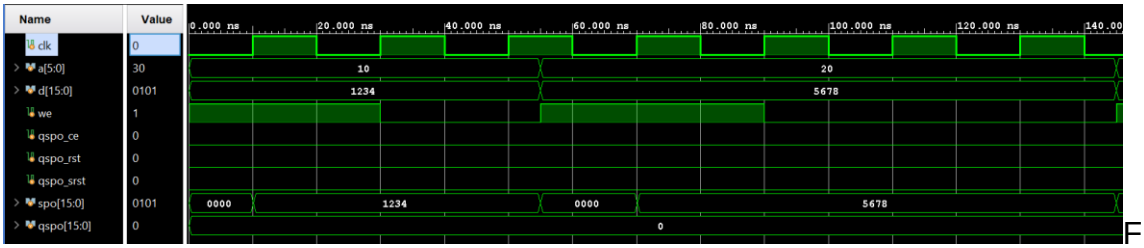

Fig. 80

Până la primul posedge al semnalului de ceas, valorile semnalelor sunt definite conform Fig. 48. La momentul 10ns, semnalul we devine 1, ceea ce declanșează o operație de scriere la adresa 10h. Înainte de acest moment, ieșirea spo avea valoarea 0, iar imediat după scriere se actualizează la 1234h. we va deveni 0 la următorul posedge clk, așa cum se poate vedea în Fig. 49. La 55ns, we revine la 0 și apoi urcă din nou la 1, ceea ce indică o nouă scriere, de această dată la adresa 20h. Când adresa de acces se schimbă de la 10h la 20h, spo trece temporar la 0, deoarece adresa 20h nu fusese utilizată anterior. Scrierile efective în memorie au loc doar la posedge-ul semnalului de ceas, moment în care we este activ. Astfel, la 70ns, scrierea la adresa 20h este finalizată, iar spo este actualizat la 5678h.

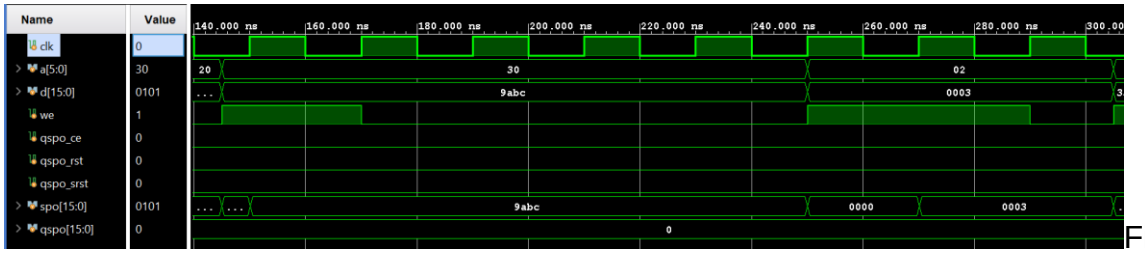

Fig. 81

La 250ns, în momentul în care apare un posedge al semnalului de ceas, we este încă 0, ceea ce înseamnă că nicio operație de scriere nu este inițiată în acel ciclu de ceas. După acest moment, we devine 1, indicând intenția de a efectua o scriere. Adică we se află în tranziția de la 0 la 1. Totuși, deoarece scrierile în memorie sunt procesate doar la posedge-ul ceasului, această operație va fi luată în considerare abia la următorul posedge al clk. Astfel, la 270ns, scrierea este efectuată, iar valoarea corespunzătoare este înregistrată în memorie.

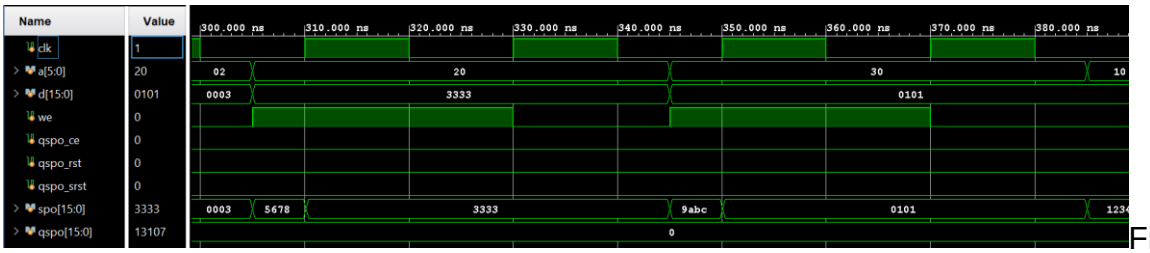

Fig. 82

La 305ns, adresa și datele de scriere se schimbă, iar spo devine 5678h, reflectând valoarea stocată anterior la adresa 20h. Această valoare rămâne neschimbată până la primul posedge al semnalului clk, moment în care we este activ (1), semnalând o operație de scriere. Imediat după acest posedge, spo este actualizat la 3333h, confirmând că noua valoare a fost scrisă în memorie.

Un fenomen similar are loc la 345ns, când adresa și datele de scriere se modifică din nou. În această etapă, spo afișează valoarea anterioară stocată și rămâne constantă până la următorul posedge clk. La acest moment, we este încă 1, ceea ce declanșează o nouă scriere. Imediat după propagarea prin ceas, spo este actualizat la noua valoare, indicând că scrierea a fost efectuată corect.

Citire Asincronă

```
// Citire asincronă
readMemoryAsync(6'h10, readdata);
$display("Async Read Data after Write 1: %0h", readdata);
#35;
readMemoryAsync(6'h20, readdata);
$display("Async Read Data after Write 2: %0h", readdata);
#20;
readMemoryAsync(6'h30, readdata);
$display("Async Read Data after Write 3: %0h", readdata);
#30;
readMemoryAsync(6'h10, readdata);
$display("Async Read Data after Write 1: %0h", readdata);
#45;
```

Fig. 83

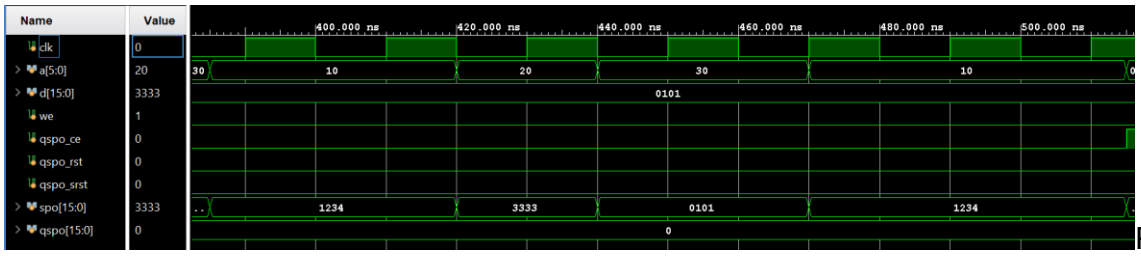

Fig. 84

La 385ns, adresa se schimbă la 10h și se inițiază o citire asincronă prin spo. Se poate observa că această operație are loc instantaneu, fără a fi necesară așteptarea unui ciclu de ceas. Același comportament se observă și pentru următoarele două citiri asincrone. Ultima citire asincronă, realizată la 470ns, confirmă că posedge clk nu influențează procesul de citire și că acesta se desfășoară corect. De asemenea, se remarcă faptul că valoarea datelor rămâne constantă, menținând ultima valoare validă.

Citire sincronă

```
// Citire sincronă
readMemorySync(6'h2, readdata);
$display("Sync Read Data: %0h", readdata);
#60
readMemorySync(6'h20, readdata);
$display("Sync Read Data: %0h", readdata);
#55
```

Fig. 85

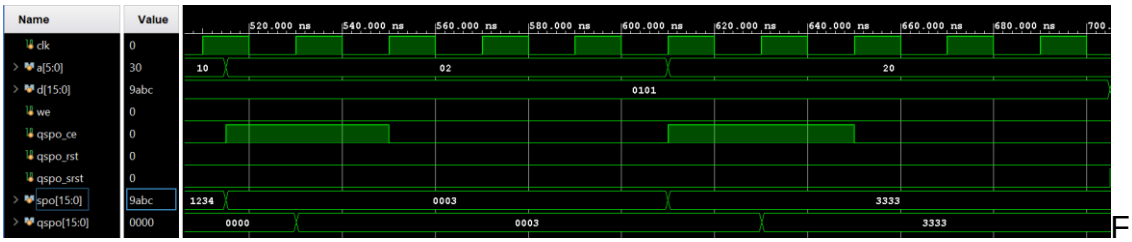

Fig. 86

La 515ns, qspo_ce din 0 devine 1 și la următorul posedge clk se va lua in considerare citirea sincronă qspo, adică la 530ns, iar această ieșire va deveni la 530ns valoarea de la adresa corespunzătoare, până la următoarea comandă. qspo_ce devine 0 la următorul posedge clk, așa cum se poate observa în Fig. 52. La 610ns pe posedge clk, qspo_ce se află în tranziția de la 0 la 1, deci qspo_ce = 1 la următorul posedge clk și atunci o să se și vadă și update ul lui qspo.

Reset Sincron

```
// Reset sincron
resetSync();
#40
readMemorySync(6'h10, readdata);
$display("Sync Read Data: %0h", readdata);
#20
resetSync();
#10
readMemorySync(6'h20, readdata);
$display("Sync Read Data: %0h", readdata);
#7
```

Fig. 87

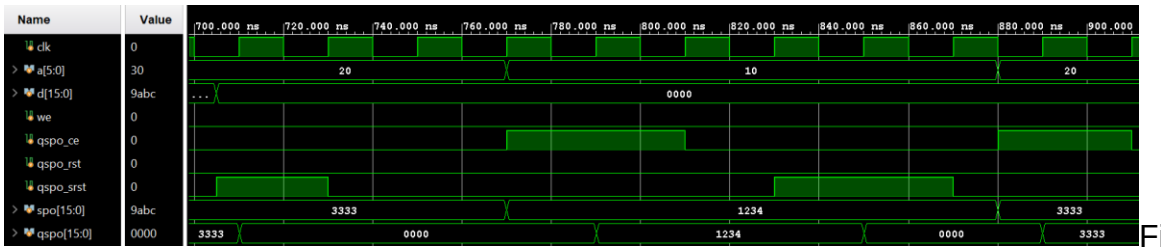

Fig. 88

La momentul 705ns, semnalul qspo_srst este activat, însă acesta nu afectează imediat ieșirea qspo, deoarece resetul sincron necesită un ciclu de ceas pentru a fi luat în considerare. Astfel, la 710ns, pe următorul posedge clk, qspo este setat la 0 și rămâne în această stare până la o actualizare ulterioară. qspo_srst devine 0 la următorul posedge clk, așa cum se poate observa în Fig. 50. Un comportament similar este observat și la 820ns, unde qspo_srst face tranziția de la 0 la 1. Totuși, acest semnal va avea efect doar la următorul **posedge clk**, adică la 850ns, moment în care qspo este resetat la 0, conform comportamentului resetului sincron.

Reset Asincron

```
// Reset asincron
resetAsync();
#13
readMemorySync(6'h30, readdata);
$display("Sync Read Data: %0h", readdata);
```

Fig. 89

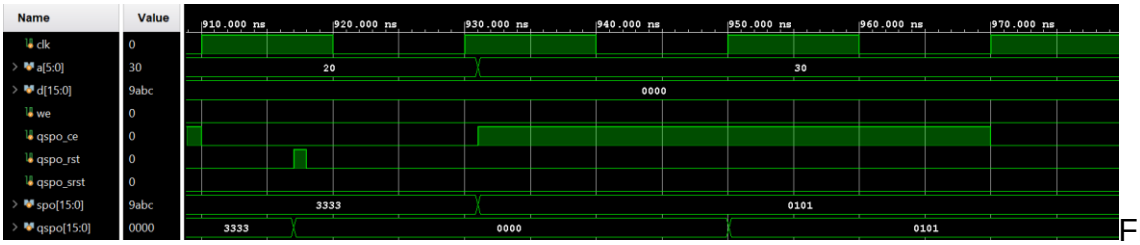

Fig. 90

La momentul 812ns, semnalul qspo_rst face tranziția la 1, ceea ce activează resetul asincron. Spre deosebire de resetul sincron, care necesită un posedge clk pentru a avea efect, resetul asincron este aplicat imediat, fără a depinde de ceas. qspo_rst devine 0 după 1ns, așa cum se poate observa în Fig. 51. Astfel, qspo devine 0 instantaneu, reflectând comportamentul așteptat al unui reset asincron. În această stare, qspo va rămâne 0 până când va apărea o nouă operație de scriere sau citire care să modifice conținutul său.

Bibliografie

1. https://docs.amd.com/v/u/en-US/dist_mem_gen_ds322
2. https://www.chipverify.com/tutorials/systemverilog
3. https://www.chipverify.com/tutorials/uvm