

Universitatea POLITEHNICA din Bucureşti



Facultatea de Inginerie Electrică

PROIECT DE DIPLOMĂ

VERIFICAREA FUNCȚIONALĂ A DESIGNURILOR DIGITALE

Absolvent: Dragoş-Ioan PASCU

Conducător științific: Prof.dr.ing. George Călin SERIȚAN

BUCUREŞTI

Iulie 2023

Cuprins

1. Introducere.....	3
1.1 Tendințe în fabricarea circuitelor integrate	3
1.2 Efortul de verificare funcțională ale circuitelor integrate.....	5
1.3 Scopul lucrării.....	6
2. Context.....	7
2.1 AMBA Standard	7
2.3 AHB Protocol	10
2.3 Verificarea de design.....	19
2.3.1 Context, principii și metodologii.....	19
2.2.2 SystemVerilog.....	23
2.2.3 UVM.....	25
2.2.4 Unelte folosite	29
3. Verificare funcțională unui design de arbitru AHB.....	31
3.1 Planul de verificare.....	31
3.2 Mediul de verificare.....	40
3.3 Analiza rezultatelor.....	66
4. Concluzii.....	67
Bibliografie	69

1. Introducere

1.1 Tendințe în fabricarea circuitelor integrate

În prezent, circuitele integrate (CI) sunt esențiale în tehnologia modernă și sunt alimentate de progresele semiconductoarelor. Producătorii de cipuri se confruntă cu cerințe din ce în ce mai mari de performanță, eficiență și complexitate, într-un mediu în care tendințele semiconductoarelor dictează evoluția industriei.

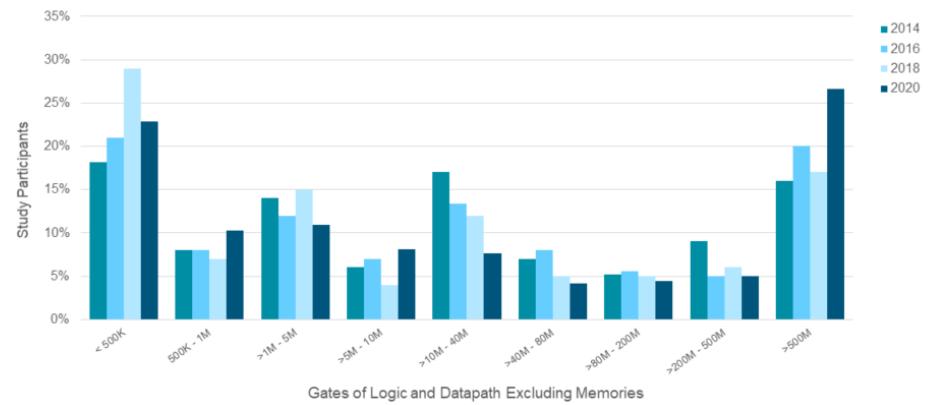
Unul dintre trendurile majore în domeniul semiconductoarelor și al circuitelor integrate este miniaturizarea. Producătorii se străduiesc să reducă dimensiunea componentelor individuale și să integreze mai multe funcționalități pe același cip. Această tendință a permis dezvoltarea de dispozitive electronice mai mici, mai ușoare și mai portabile, cum ar fi smartphone-urile și dispozitivele inteligente.

Un alt trend important este creșterea capacitații de procesare și a eficienței energetice. Producătorii de cipuri își concentrează eforturile asupra dezvoltării tehnologii avansate, cum ar fi procesoarele cu multiple nuclee și tehnologiile de fabricație de înaltă performanță, care permit obținerea de performanțe ridicate la consumuri de energie reduse.

În același timp, există o creștere a cererii pentru dispozitive integrate cu funcționalități complexe, cum ar fi senzori avansați, tehnologie wireless și inteligență artificială. Acest lucru conduce la dezvoltarea de cipuri specializate și personalizate pentru aplicații specifice, cum ar fi Internetul Lucrurilor (IoT), vehiculele autonome și dispozitivele medicale.

Datorită îmbunătățirilor tehnologice, cum ar fi creșterea capacitații de calcul a cipurilor, memorii mai mari și mai rapide, capacitatea de a executa algoritmi de inteligență artificială și reducerea costurilor de producție, s-a observat o schimbare radicală în modul în care trăim și lucrăm. Această îmbunătățire a adus provocări nemaîntâlnite anterior pentru companii, care au avut nevoie să producă cipuri tot mai complexe, menținând în același timp standarde ridicate de calitate și încercând să evite întârzierile excesive. Confirmând această difuziune, Alsop [1] a estimat o cifră de afaceri în anul 2020 din vânzarea sistemelor integrate de 361,23 miliarde de dolari americani și a prevăzut o creștere în anii următori. De acord cu aceeași opinie, Foster a estimat o creștere continuă a complexității în producția de circuite integrate în ultimii șapte ani, evidențiind două tendințe majore [2]. Pe de o parte, există o creștere a producției de design-uri cu mai puțin de 1M de porți logice (porți logice, cai de date, excluzând memorii); aceasta se datorează cipurilor de senzori mai mici pentru dispozitivele IoT și auto. Pe de altă parte, totuși, producția de design-uri mari este în continuare prezentă: 36% din proiectele care lucrează în jurul valorii de 80K porți și 31% care lucrează între 80K și 1M de porți. Cu toate că este doar o dimensiune care exprimă complexitatea designului, putem înțelege cum acestea devin provocări și cum există o nevoie de identificare a calității acestora.

ASIC Study Participation by Gate Count (Design Size)

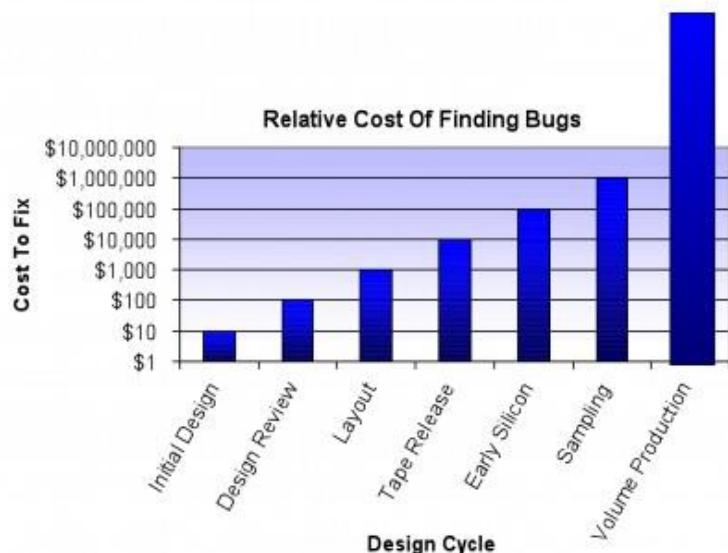


Source: Wilson Research Group and Mentor, A Siemens Business, 2020 Functional Verification Study
Page 1 © Siemens 2020 | 2020-10-15 | Siemens Digital Industries Software | Where today meets tomorrow.

SIEMENS

Fig 1.1 Dimensiune IC/ASIC per numar porti logice [1]

Un defect detectat în faza inițială de proiectare este de 10 ori mai ieftin de reparat decât un defect descoperit în faza de finalizare a proiectului și de sute sau mii de ori mai ieftin dacă este detectat de către consumator. Pentru a asigura că un circuit integrat funcționează corect și este fiabil, este necesar să existe proceduri de verificare paralele cu procesul de proiectare a produsului. [3]



Silicon Debug, Doug Josephson and Bob Gottlieb, (Paul Ryan)
D. Gizopoulos (ed.), *Advances in Electronic Testing: Challenges and Methodologies*, Springer, 2006

Fig 1.2 Costul relativ al găsirii defectelor in diferite faze ale proiectului [3]

Într-adevăr, detectarea și remedierea erorilor în faza inițială de proiectare a unui circuit integrat (CI) este mult mai rentabilă decât găsirea acestora în timpul fazei de finalizare a proiectului sau, mai rău, după ce produsul ajunge la consumatori. Costul corectării problemelor crește exponential în fiecare etapă din cauza resurselor suplimentare necesare și a impactului potențial asupra termenelor de producție.

În timpul fazei inițiale de proiectare, erorile pot fi identificate și rezolvate prin simulare, modelare și tehnici de verificare. Aceste metode permit designerilor să identifice și să remedieze potențiale erori înainte ca CI-ul să intre în producție. Prin investirea în proceduri riguroase de verificare paralele cu procesul de proiectare, companiile pot asigura că CI-urile lor funcționează corect și sunt fiabile, reducând riscul de defecțiuni ulterioare și de costuri suplimentare semnificative.

1.2 Efortul de verificare funcțională a circuitelor integrate

În figura 1.3 este prezentat numărul mediu de ingineri implicați în producția și verificarea unui circuit integrat (IC)/ASIC. Este observabil faptul că există o nevoie mai mare de verificatori și că, de-a lungul anilor, a existat o creștere a cererii pentru ambele funcții.

În procesul de dezvoltare a unui IC/ASIC, producția și verificarea sunt două aspecte esențiale. Producția se referă la etapele de fabricație fizică a cipurilor, cum ar fi etapizarea designului, fabricarea stratului semiconductor, adăugarea straturilor de metale și izolatori, precum și testarea finală a produsului finit. Pe de altă parte, verificarea constă în asigurarea că IC-ul funcționează conform specificațiilor și că nu conține erori sau defecte.

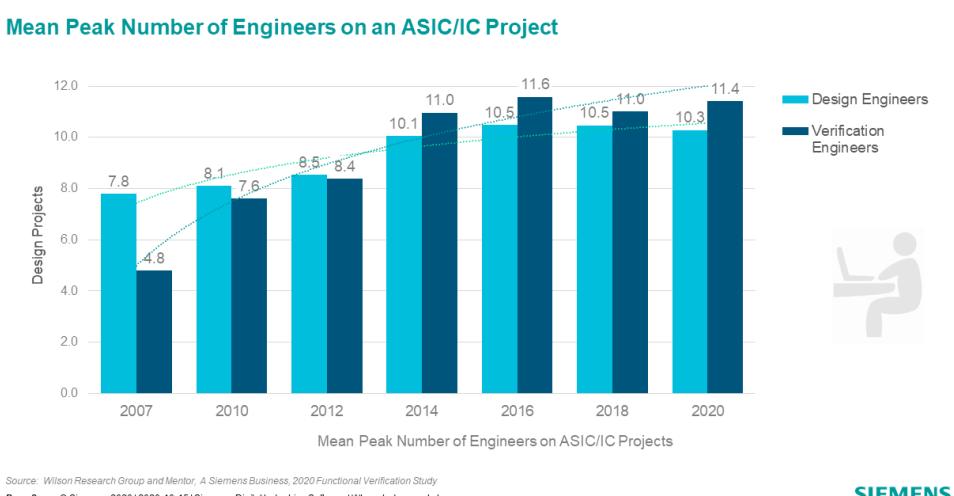


Fig 1.3 Numărul mediu de ingineri per proiect [4]

În figura 1.4, se poate observa, de asemenea, că nu toate proiectele necesită același timp de verificare. Aceasta se datorează în mod obișnuit capacitatea designerilor de a utiliza componente proprietare - proprietate intelectuală (PI) - deja verificate și de a le integra în

noile lor produse, care, prin urmare, nu necesită timp semnificativ pentru verificare, aşa cum poate fi cazul într-un design complet nou.

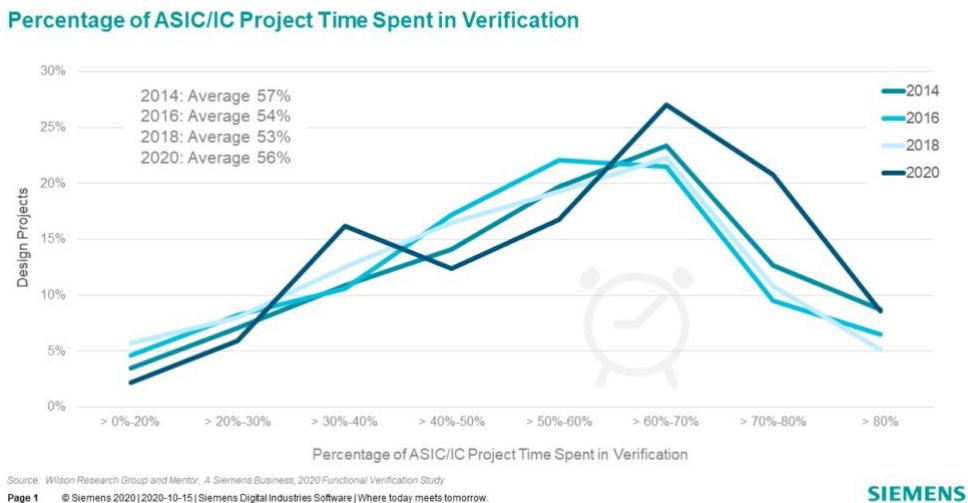


Fig 1.4 Procent din timpul din proiect petrecut in verificare [4]

Unul dintre avantajele utilizării componentelor IP (intellectual property) verificate este că acestea au trecut deja prin etapele de proiectare și verificare, confirmându-și astfel funcționalitatea și fiabilitatea. Atunci când designerii integreză aceste componente IP în noile produse, verificarea acestora poate fi mai rapidă și mai simplă, deoarece nu este necesar să se repete întregul proces de verificare pentru acele componente.

Cu toate acestea, în cazul unui design complet nou, în care toate componentele sunt dezvoltate de la zero, verificarea poate necesita mai mult timp și resurse. Acest lucru se datorează faptului că fiecare componentă trebuie proiectată și verificată individual, asigurându-se că funcționează în parametrii specificați și că se încadrează în cerințele de calitate și performanță.

1.3 Scopul lucrării

În această teză, este prezentată verificarea unui design, metodologii și tehnologii care ajuta în efortul de verificare, aplicându-le unui exemplu concret al unui protocol standard pentru magistrale pe cip, dezvoltat de ARM: Advanced High-performance Bus (AHB). Acesta este un standard deschis și disponibil gratuit pentru interconectarea și gestionarea nucleelor de proprietate intelectuală (IP) într-un sistem pe cip (SoC).

AHB permite dezvoltarea designurilor de cipuri multiprocesor într-un mod modular, reutilizabil și scalabil, ceea ce ajută la evitarea costisoarelor redesign-uri și reduce timpul de lansare pe piață al circuitelor integrate[2].

Este prezentată în detaliu cea de-a doua versiune a protocolului AHB, deoarece aceasta sta la baza protocolului și modificărilor ulterioare acestuia. De asemenea, designul pe care o să-l folosesc este facut să funcționeze pentru versiunea a doua a acestui protocol.

Ulterior, s-a lucrat la o componentă de verificare (VC). Aceasta poate verifica dacă un design AHB este într-adevăr consistent cu specificațiile. Procesul utilizat este cunoscut sub numele de verificare funcțională, larg adoptată în industria semiconductoarelor, prin crearea unui plan de verificare și extinderea unui mediu de verificare parțial implementat deja, bazat pe o metodologie standard cunoscută sub numele de Universal Verification Methodology

(UVM). Acesta din urmă a fost prezentat, iar SystemVerilog a fost adoptat pentru implementarea sa, aşa cum este sugerat de standard însuşi.

Apoi, au fost codificate câteva dintre atributele planului de verificare și au fost creați stimuli pentru a stimula designul, trăgându-se unele concluzii. Intenția concretă este de a conecta ulterior aceasta la designul unui cip real și de a verifica dacă designul AHB corespunde specificațiilor.

2. Context

2.1 AMBA Standard

AMBA, Advanced Microcontroller Bus Architecture, este unul dintre cele mai utilizate tipuri de busuri de comunicare în arhitecturile de procesoare. Dezvoltat de ARM Ltd în 1996, a fost inițial destinat să fie utilizat în microcontrolere pentru a susține eficient comunicarea între nucleele procesorului ARM. AMBA a fost conceput ca un set de specificații de interconectare care standardizează mecanismele de comunicare pe cip între diverse blocuri funcționale (sau IP-uri) pentru a construi design-uri SoC (System-on-Chip) de înaltă performanță. Unul dintre principiile pe care se bazează AMBA este independența față de tehnologie. Este un protocol descris doar la nivelul "ciclului de ceas", fără a detalia caracteristicile electrice, care sunt lăsate la discreția producătorilor.[6]

ACEste design-uri includ, în general, unul sau mai multe microcontrolere sau microprocesoare, împreună cu mai multe componente integrate pe un singur cip, cum ar fi memorie internă sau externă, bridge-uri, procesoare DSP, controlere DMA, acceleratoare și diferite periferice, precum USB, UART, PCIE, I2C etc. Motivația principală a protocolului AMBA este de a oferi o modalitate standard și eficientă de interconectare a acestor blocuri, cu posibilitatea de reutilizare în multiple design-uri.

AMBA a evoluat de-a lungul timpului, odată cu creșterea numărului de blocuri funcționale integrate în design-urile SOC. În anul 2003, odată cu lansarea versiunii AMBA 3, a fost introdus protocolul AXI (Advanced eXtensible Interface) ca o soluție de conectivitate punct-la-punct, care depășește limitele protocolului de bus partajat utilizat anterior (AHB/ASB). AXI a fost îmbunătățit în versiunea AMBA 4, iar ulterior a fost introdus protocolul ACE (AXI Coherency Extension) pentru a permite coerentă sistemului la nivel de memorie între mai multe procesoare și pentru a facilita tehnologii precum ARM big.LITTLE processing. Aceste protocole AXI și ACE au fost adoptate în design-urile SOC care necesită performanțe ridicate și coerentă a memoriei.[7]

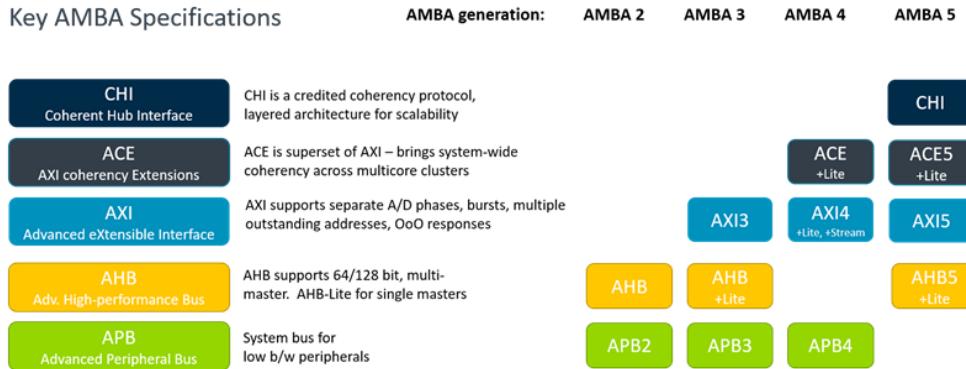


Fig 2.1.1 Evoluția protoalelor din standardul AMBA [8]

O altă evoluție importantă în domeniul SOC-urilor a avut loc în era telefoanelor mobile și a smartphone-urilor, odată cu integrarea de procesoare dual/quad/octa-core cu cache-uri partajate și necesitatea unei coerente hardware în subsistemul de memorie. Aceasta a dus la introducerea protocolului ACE în revizia AMBA 4.[7]

În prezent, în era calculului heterogen pentru piețele HPC și data center, tendința de integrare continuă cu un număr tot mai mare de nuclee de procesor, împreună cu mai multe elemente de calcul heterogene, cum ar fi GPU-uri, DSP-uri, controlere de memorie și subsisteme IO. În 2013, în revizia AMBA 5, a fost introdus protocolul CHI (Coherent Hub Interconnect) ca o redesenare a protocolului AXI/ACE. Protocolul CHI, bazat pe pachete de date, înlocuiește protocolul AXI/ACE bazat pe cicluri de ceas și oferă o soluție scalabilă pentru integrarea sistemelor SOC complexe, oferind o mai mare eficiență în consumul de energie, performanță sporită și facilități avansate de securitate.[7]

De menționat faptul că AMBA nu este doar un set de protoale de comunicație, ci și un cadru de lucru pentru proiectarea și verificarea sistemelor SOC. Aceasta oferă, de asemenea, specificații pentru modelarea și simularea interfețelor și arhitecturilor SOC, împreună cu un set de unelte și biblioteci software pentru dezvoltarea și testarea sistemelor SOC bazate pe aceste specificații.

Astfel, AMBA reprezintă un standard "de facto" în industria de proiectare de circuite integrate și joacă un rol crucial în dezvoltarea SOC-urilor de înaltă performanță, asigurând o conectivitate eficientă între diversele blocuri funcționale și facilitând dezvoltarea rapidă și reutilizarea componentelor de IP în design-uri complexe.

Dintre protoalele mentionate în standardul AMBA ar trebui să exemplifice câteva caracteristici definitorii, locul unde sunt utilizate și capabilitățile acestora.

APB (Advanced Peripheral Bus):

- Protocolul APB este folosit pentru conectarea perifericelor cu cerințe reduse în ceea ce privește lățimea de bandă, performanța, puterea și complexitatea.
- Este un protocol simplu, nepipelined, utilizat pentru comunicarea (citirea sau scrierii) între un master/bridge și mai multe sclavi/periferice prin intermediul unui bus partajat.
- Nu suportă transferuri de date în bloc (burst) și folosește același set de semnale pentru operațiunile de citire și scriere.
- Este utilizat pentru accesarea perifericelor I/O care necesită transferuri necomplexe. [6][8]

AHB (Advanced High-performance Bus):

- Protocolul AHB este folosit pentru conectarea componentelor care necesită o lătime de bandă mai mare pe un bus partajat.
- AHB este un protocol partajat pentru mai mulți masteri și sclavi, care permite transferuri de date în bloc (burst) și asigură performanțe mai ridicate.
- Este utilizat pentru accesul la memorie internă sau externă, interfețe de memorie, controlere DMA, procesoare DSP și alte periferice de înaltă performanță.
- AHB-Lite este o versiune simplificată a protocolului AHB, proiectată pentru design-uri cu un singur master, eliminând nevoie de arbitraj, retriere și tranzacții separate. [6][8]

AXI (Advanced eXtensible Interface):

- Protocolul AXI este utilizat pentru interconexiuni cu lătime de bandă mare și latență scăzută.
- Este un protocol punct-la-punct care depășește limitările busului partajat prin suportul pentru mai mulți agenți conectați.
- AXI permite transferuri de date în bloc (burst), are căi separate pentru citire și scriere și suportă transferuri de date multiple în paralel.
- Este folosit în design-uri SOC cu cerințe ridicate de performanță, cum ar fi interconectarea de procesoare puternice, acceleratoare, memorie cache partajată și subsisteme de comunicare IO. [6][8]

Un exemplu posibil al folosirii acestor protocoale este în construcția unui sistem pe cip (System-on-Chip, SoC), în care diverse componente (cum ar fi procesoare, memorii, controlere de periferice etc.) sunt integrate pe același cip. Acest sistem poate utiliza bus-ul AMBA pentru a facilita comunicarea între aceste componente.

Este posibil să vizualizăm bus-ul într-un mod ierarhic, împărțindu-l în două părți: un bus de sistem (AHB) și un bus periferic (APB), legate de un pod care gestionează datele și operațiile între cele două domenii.

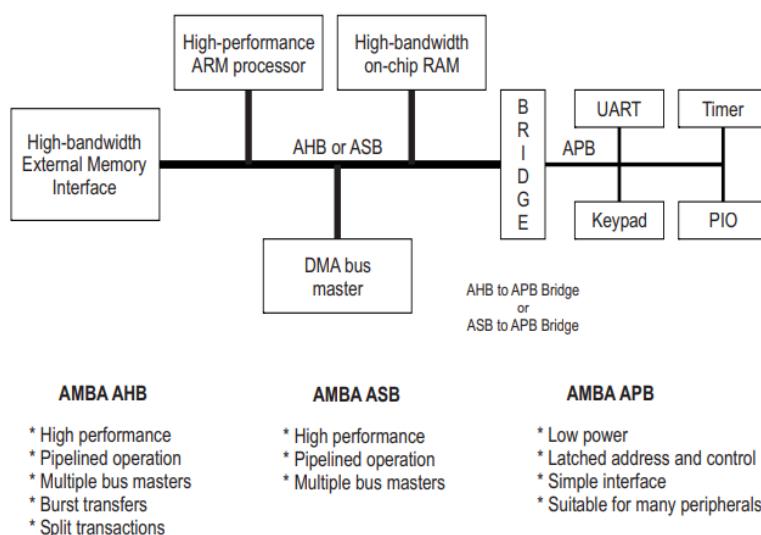


Figure 1-1 A typical AMBA system

Fig 2.1.2 Un exemplu al folosirii standardului AMBA intr-un sistem [9]

2.3 AHB Protocol

AHB, Advanced High-performance Bus, definește interfețele pentru master, slave și conexiunile dintre acestea, permășând performanțe și frecvențe de ceas ridicate, datorită următoarelor caracteristici:

- transferuri de tip burst
- tranzacții împărțite (split)
- transfer de control între masteri într-un singur ciclu de ceas
- funcționare pe un singur front activ de ceas
- implementare non-tristate
- configurații ale bus-ului de date mai largi (64/128 de biți)

Înainte ca un transfer AHB să poată începe, masterul care dorește să facă transferuri pe magistrală trebuie să primească "grant". Acest proces de cerere a magistralei se face ridicând semnalele de cerere de magistrală, care ajung la arbitru. Arbitrul are rolul de a acorda magistrala masterului cu cea mai mare prioritate, în funcție de algoritmul de prioritate implementat de către arbitru.

Un master care deține magistrala poate începe un transfer trimînd semnale de control și adresă. Semnalele de control includ direcția, adică dacă operația este de a scrie datele (de la master la slave) sau de a citi datele (de la slave la master), lățimea transferului și dacă transferul este simplu sau în serie (incremental sau înfășurare la o anumită adresă). Transferurile constau din două faze:

- **Address phase:** Faza de adresa durează un singur ciclu de ceas în care adresa și semnalele de control sunt trimise.
- **Data phase:** Faza de date poate dura un ciclu de ceas sau mai multe. Un slave poate cere extensia fazei de date folosind semnalul HREADY care atunci când este "low" indică inserarea ciclurilor de așteptare.

Interfețele Master și Slave

Interfața Master

Masterul trimite informațiile necesare pentru scriere sau citire către slave. Fiecare slave este selectat folosind un semnal de selecție, HSEL. În acest fel, slaveul respectiv este asociat unui interval de adrese la care masterul vrea să facă transferul. De asemenea, masterul primește intrari semnalul de ceas și reset, răspunsul slaveului și datele citite în cazul în care a fost o operație de citire.

The interface diagram of an AHB bus master shows the main signal groups.

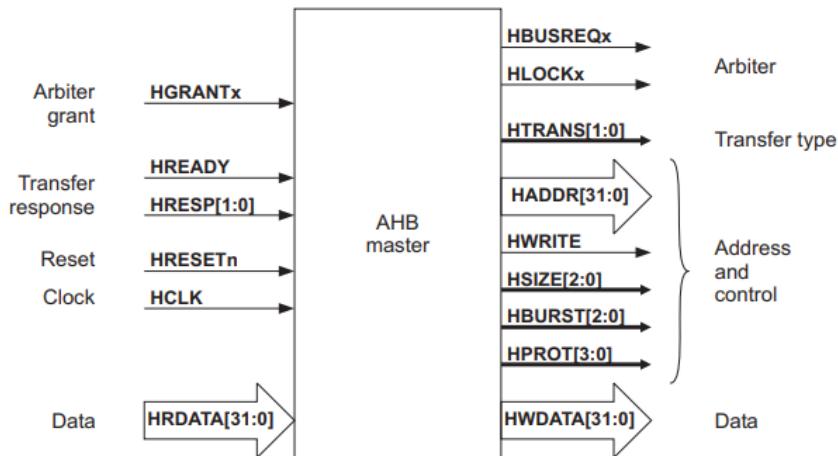


Figure 3-27 AHB bus master interface diagram

Fig 2.3.1 Interfața unui master AHB [9]

- **HCLK:** Semnalul HCLK reprezintă semnalul de ceas al interfeței AHB. Toate transferurile de date sunt sincronizate cu frontul pozitiv al acestui semnal. Aceasta furnizează temporizarea necesară pentru sincronizarea corectă a semnalelor și operațiilor între maestru și sclav în cadrul interfeței AHB.
- **HRESETn:** Semnalul HRESETn este generat de controlerul de reset și este utilizat pentru a reseta magistrala. Semnalul este activ la nivel scăzut (active low), ceea ce înseamnă că resetarea este declanșată atunci când semnalul este în starea logică "0". Prin resetarea magistralei, se restabilește starea inițială și se pregătește sistemul pentru a începe operațiile de transfer.
- **HADDR[31:0]:** Semnalul HADDR reprezintă magistrala de adrese a sistemului, cu o lățime de 32 de biți. Acest semnal este conectat la decodorul de adrese și indică adresa specificată pentru operațiile de citire sau scriere efectuate de maestru. Prin intermediul acestui semnal, se selectează dispozitivul sau locația în care se va realiza transferul de date.
- **HBURST[2:0]:** Semnalul HBURST indică tipul de transfer efectuat. Acesta este un semnal cu o lățime de 3 biți și poate avea diferite valori, în funcție de tipul de transfer. Valoarea acestui semnal specifică dacă transferul este de tip rafală (burst) și, în caz afirmativ, indică modul specific în care se realizează transferul în rafală, cum ar fi transferul incremental sau transferul cu înfășurare.
- **HMASTLOCK:** Semnalul HMASTLOCK indică faptul că maestrul curent efectuează o secvență blocată de transferuri. Acest semnal are aceeași sincronizare temporală cu semnalul de adresă și indică faptul că maestrul este angajat într-o secvență continuă de transferuri și trebuie să fie gestionat corespunzător pentru a asigura integritatea și corecitudinea datelor transferate.
- **HPROT[3:0]:** Furnizează informații suplimentare despre protecția și controlul accesului în cadrul interfeței AHB. Acest semnal indică regulile de acces pentru transferurile de date, specificând permisiunile și restricțiile asociate cu fiecare acces la bus.
- **HSIZE[2:0]:** Acest semnal indică dimensiunea transferului (byte, jumătate de cuvânt, cuvânt și extensii de până la 1024 de biți). Valoarea acestui semnal determină câte biți sunt transferați într-un singur ciclu.

- **HMASTER[3:0]:** Identifierul maestrului, unic pentru fiecare maestru. Acest semnal permite identificarea specifică a maestrului care inițiază transferul și poate fi utilizat pentru a gestiona prioritățile și politica de arbitraj în cadrul interfeței AHB.
- **HTRANS[1:0]:** Acest semnal indică tipul de transfer. Poate fi: IDLE (inactiv), BUSY (ocupat), NON SEQUENTIAL (nesecvențial) și SEQUENTIAL (secvențial). Valorile acestui semnal reflectă starea transferului și pot fi utilizate pentru a sincroniza și a controla fluxul de date între maestru și sclav.
- **HWDATA[31:0]:** Busul de date pentru scriere. Acest bus conține datele care vor fi scrise în sclav. Dimensiunea busului poate fi extinsă pentru a permite transferul unui număr variabil de biți de la maestru la sclav.
- **HWRITE:** Acest semnal indică direcția transferului. Dacă semnalul este înalt (1), se indică o operație de scriere, în caz contrar se indică o operație de citire. Acest semnal are aceeași sincronizare temporală cu semnalul adresei, dar este menținut constant pe durata unui transfer în serie (burst).

Interfața Slave

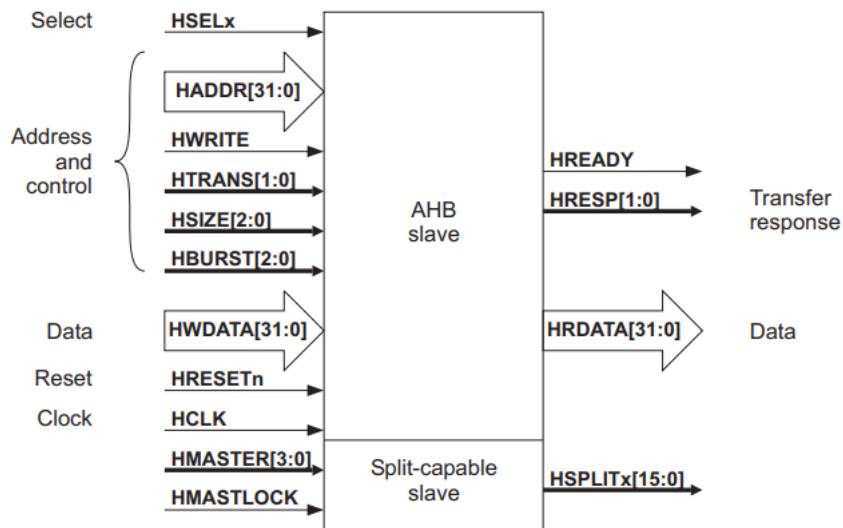


Figure 3-23 AHB bus slave interface

Fig 2.3.2: Interfata unui slave AHB [9]

- HRDATA[31:0]: Semnalul HRDATA reprezintă busul de citire. În timpul unei operații de citire, busul selectat produce datele către multiplexor, care le furnizează apoi maestrului. Busul poate fi extins și poate avea o lățime mai mare de 32 de biți, permitând transferul unui volum mai mare de date într-un singur ciclu.
- HREADYOUT: Semnalul HREADYOUT indică încheierea unui transfer (stare logică înaltă) sau extinderea unui transfer (stare logică scăzută). Acest semnal este utilizat pentru a sincroniza și a indica finalizarea unui transfer sau necesitatea unei extinderi a acestuia. Este important în gestionarea corectă a fluxului de date între maestru și sclav.
- HRESP: Semnalul HRESP indică starea de răspuns a transferului. Acest semnal furnizează informații despre rezultatul transferului și poate indica dacă transferul a fost efectuat cu succes (răspuns pozitiv) sau dacă a apărut o eroare sau o condiție de eșec (răspuns negativ). Este utilizat pentru a confirma dacă transferul a fost realizat corect și poate fi folosit pentru a gestiona situațiile de eroare în cadrul interfeței AHB.

Interconectarea

În cadrul interfeței AHB (Advanced High-performance Bus), un master și un slave sunt conectați utilizând un multiplexor (MUX) și un decodator.

Conexiunea între un master și un slave în AHB implică două componente principale: MUX și decodator.

1. Multiplexor (MUX): Un multiplexor este utilizat pentru a multiplexa diferite semnale provenite de la diferiți maștri într-un singur canal de date. În cazul AHB, semnalele de adresă, date și control provenite de la mai mulți maștri sunt multiplexate într-un singur flux de date care va fi transmis către slave. MUX selectează și comută semnalele de la maștrii activi în funcție de necesitate.
2. Decodator: Decodatorul este responsabil de decodarea semnalelor de adresă și de selectarea slave-ului adecvat în funcție de adresa specificată de către maestru.

Decodorul recepționează semnalele de adresă și identifică adresa destinatarului, permitând comunicarea directă între maestru și slave.

Astfel, atunci când un master dorește să efectueze o operație de citire sau scriere către un slave în cadrul interfeței AHB, semnalele de adresă și control sunt multiplexate și apoi transmise prin intermediul multiplexorului către decodor. Decodorul analizează semnalele de adresă și identifică slave-ul corespunzător, permitând transferul datelor între maestru și slave. Prin intermediul acestei arhitecturi bazate pe multiplexor și decodor, interfețele AHB permit comunicarea eficientă și gestionarea corectă a transferurilor de date între mai mulți maștri și slave într-un sistem.

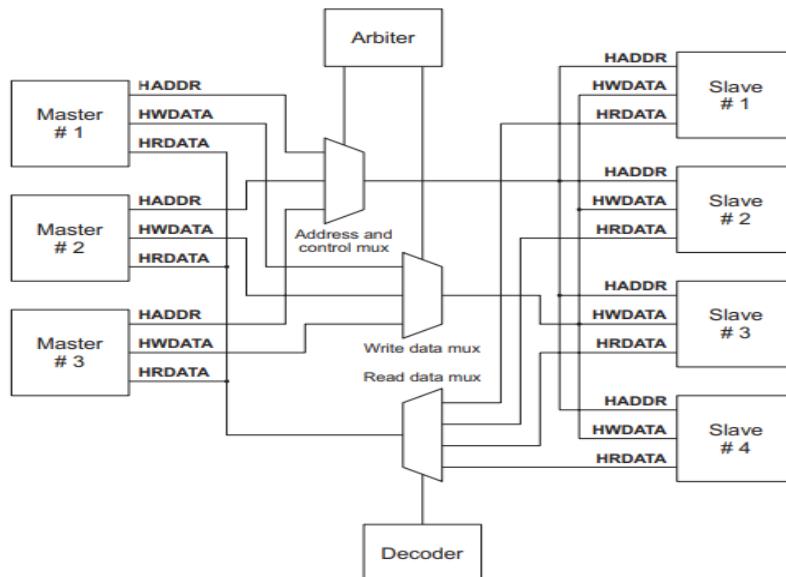


Fig 2.3.3: Interconectarea masterilor cu dispozitivele subordonate

- Semnalele decodorului:
 - HSELx: Fiecare slave are propriul său semnal HSEL și, în mod combinatoriu, acesta primește o adresă și produce un semnal care activează slave-ul țintă.
- Semnalele multiplexorului:
 - HRDATA [31:0]: Busul de date pentru citire, selectat de decodor.
 - HREADY: Indică maestrului și tuturor slave-urilor că transferul anterior s-a încheiat.
 - HRESP: Răspunsul transferului.

Protocolul oferă, de asemenea, posibilitatea de a avea mai multe selecții de slave. Un exemplu este atunci când un periferic are calea de date și registrele de control în locații diferite ale spațiului de adresă. În ceea ce privește spațiul de adresare, protocolul definește un slave implicit. Dacă nu toate adresele de memorie sunt mapate, atunci se furnizează un slave suplimentar fictiv pentru a răspunde la adresele care nu există. Acest slave generează un răspuns de eroare pentru transferurile NONSEQ și SEQ și un răspuns corect cu zero cicluri de așteptare pentru transferurile IDLE și BUSY.

Transferuri si tipuri de transfer

Transferuri simple

După cum am menționat anterior, un transfer în protocolul AHB constă din două faze distincte: faza de adresă și faza de date. Faza de adresă are loc într-un singur ciclu al semnalului HCLK, în timp ce faza de date poate dura mai multe cicluri de ceas. Prin intermediul semnalului HWRITE, putem determina direcția transferului: când HWRITE este înalt, indică un transfer de scriere pe magistrala HWDATA, iar când este scăzut, indică o operație de citire din magistrala HRDATA.

Cele mai simple transferuri, atât de citire cât și de scriere, pot fi realizate fără stări de așteptare. Astfel, atât faza de adresare, cât și faza de date pot fi finalizate într-un singur ciclu al semnalului HCLK.

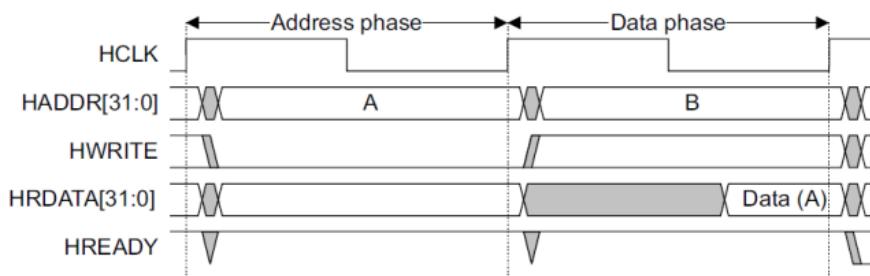


Fig 2.3.4: Transfer simplu de citire [9]

Un master trimite semnalele de adresă și de control după frontul ascendent al semnalului de ceas HCLK. Slave-ul, la următorul front ascendent al ceasului achiziționează aceste semnale. Slave-ul poate scrie HREADYOUT la nivelul “low”, în cazul în care are nevoie mai mult pentru a putea eșantiona datele. Toate HREADYOUT-urile sunt combinate prin interconectarea care

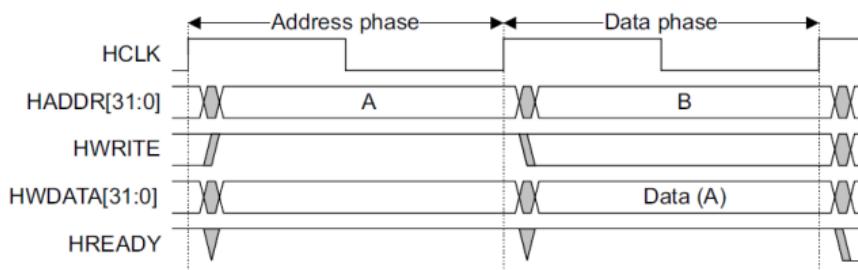


Fig 2.3.5: Transfer simplu de scriere [9]

generează un singur semnal HREADY pentru master, care înțelege că a apărut o stare de așteptare. De asemenea, este posibil să observăm că transferurile sunt pipeline: în timp ce o fază de date a unui transfer are loc, faza de adresare a următorului transfer are loc, ceea ce duce la performanțe mai ridicate ale magistralei.

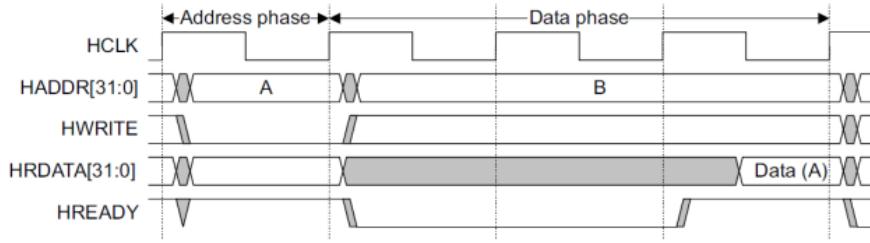


Fig 2.3.6 Transfer de citire cu două cicluri de așteptare [9]

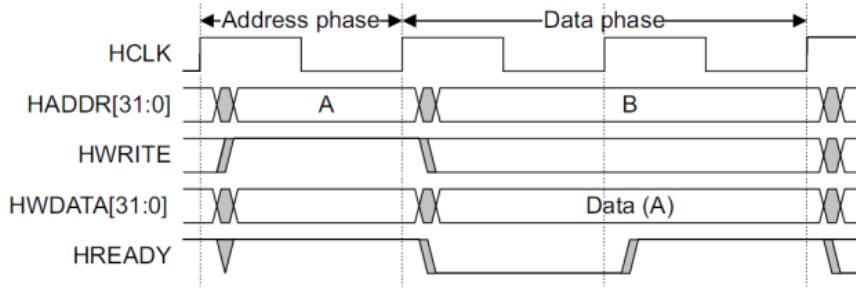


Fig 2.3.7 Transfer de scriere cu două cicluri de așteptare [9]

Prin utilizarea semnalelor HTRANS și HSIZE, este posibil să se definească combinații de transferuri. Semnalul HTRANS poate avea următoarele patru valori:

- IDLE (00_2): Indică că nu este necesar niciun transfer de date. Slave-ul răspunde cu un semnal de stare de așteptare OKAY de valoare zero.
- BUSY (01_2): Permite inserarea de cicluri idle atunci când se efectuează un burst. Aceasta înseamnă că transferul nu poate avea loc imediat, ci este necesar să se aștepte.
- NONSEQ (10_2): Indică primul transfer al unui burst sau un singur transfer.
- SEQ (11_2): Transferurile ulterioare ale unui burst sunt considerate secvențiale, iar semnalele de adresă și control sunt legate de transferul anterior.

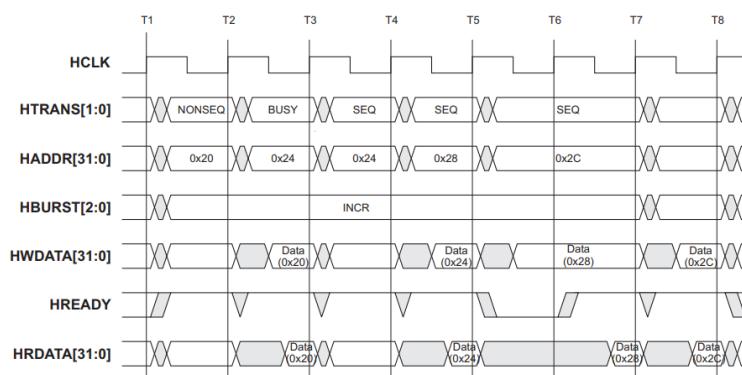


Fig 2.3.8 Exemplu de tipuri de transfer [9]

Tipuri de bursturi

Bursturile de transfer reprezintă transmiterea repetată a datelor, fără a parcurge toți pașii necesari în cazul transferurilor individuale. Bursturile pot fi de două tipuri: incrementale și circulare.

Un burst este incremental atunci când accesează locațiile de memorie în mod secvențial, incrementând valoarea adresei anterioare cu un decalaj. În schimb, un burst este

circular dacă valoarea adresei anterioare este incrementată până când se atinge o valoare limită (calculată ca numărul de beat-uri înmulțit cu dimensiunea transferului). Ulterior, adresa indicată revine la valoarea inițială. Două exemple sunt prezentate în figurile 2.12 și 2.13. Tipurile de bursturi, exprimate cu semnalul HBURST, pot fi SINGLE, INCR (lungime nedefinită), WRAP4-8-16, INCR4-8-16.

Transfer blocat

Un transfer blocat este adesea utilizat pentru a menține integritatea unui semafor. Prin acest tip de transfer, slave-ul țintă este blocat și nu primește alte comenzi de la alți masteri. Semnalul HMASTLOCK este utilizat de master pentru a semnala că se efectuează o secvență blocată de transferuri.

HMASTLOCK este un semnal care indică că masterul curent este angajat într-o secvență blocată de transferuri. Acest semnal asigură că slave-ul țintă nu răspunde la alte solicitări din partea altor masteri în timpul acestei secvențe. Este important de menționat că nu toți slave-ii din sistem sunt obligați să accepte semnalul HMASTLOCK conform protocolului. Protocolul sugerează inserarea unui transfer IDLE după un transfer blocat și impune ca aceste tipuri de transferuri să fie adresate în aceeași regiune de adresă a slave-ului. Acest lucru contribuie la menținerea coerentei și integrității sistemului, asigurând că operațiunile de citire și scriere se desfășoară corect și într-o ordine bine definită.

Transfer cu așteptare

Transferurile cu așteptare (waited transfers) au loc atunci când slave-ul are nevoie de mai mult timp pentru a prelua datele. Slave-ul utilizează semnalul HREADYOUT pentru a notifica master-ul și pentru a obține câteva cicluri de ceas suplimentare. În timpul acestor transferuri, master-ul este limitat în schimbarea semnalelor de control și adresă. Poate schimba doar tipul de transfer și adresa în trei cazuri:

- Transferuri IDLE: Masterul poate schimba HTRANS din IDLE în NONSEQ și adresa poate fi modificată până când HTRANS devine NONSEQ. Apoi, acestea trebuie să rămână constante până când HREADY devine ridicat.
- Transferuri BUSY cu burst de lungime fixă și nebună: Master-ul poate schimba HTRANS din BUSY în SEQ (pentru burst de lungime fixă) sau în orice alt tip de transfer (pentru burst de lungime nebună), dar apoi trebuie să rămână constant până când HREADY devine ridicat.
- După un răspuns de ERROR: În cazul în care slave-ul răspunde cu o eroare, master-ul poate schimba adresa în timp ce HREADY este scăzut.

Transferurile cu așteptare sunt utilizate atunci când slave-ul necesită mai mult timp pentru a prelua și procesa datele. Acest lucru poate fi cauzat de diferite condiții, cum ar fi viteza de acces a slaveului sau disponibilitatea resurselor. Slaveul utilizează semnalul HREADYOUT pentru a indica masterului că mai sunt necesare cicluri de ceas suplimentare pentru a finaliza transferul.

Masterul trebuie să fie conștient de restricțiile impuse în timpul transferurilor cu așteptare și să respecte cerințele protocolului AHB. Aceasta poate face modificări la tipul de transfer și la adresa în anumite momente cheie, cum ar fi tranzițiile între stări IDLE și NONSEQ sau în cazul unui răspuns de eroare din partea slave-ului. Respectarea acestor cerințe asigură o comunicare corectă și sincronizată între master și slave în cadrul magistralei AHB.

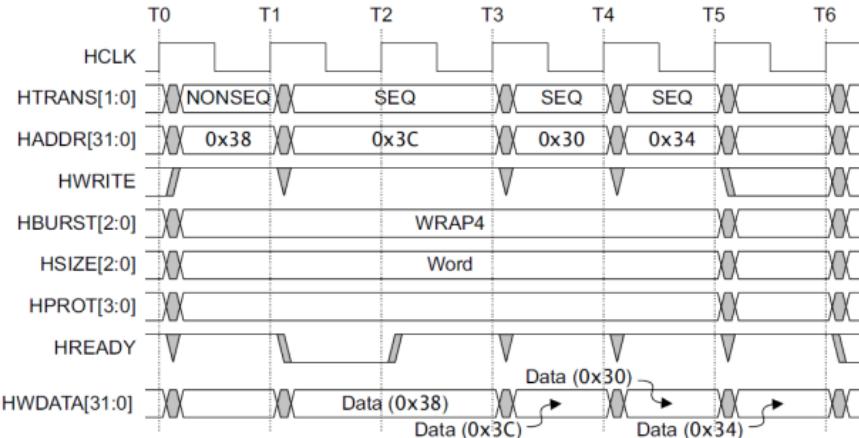


Fig 2.3.9 Transfer circular (WRAP4) cu mărimea transferului egala cu un cuvânt [9]

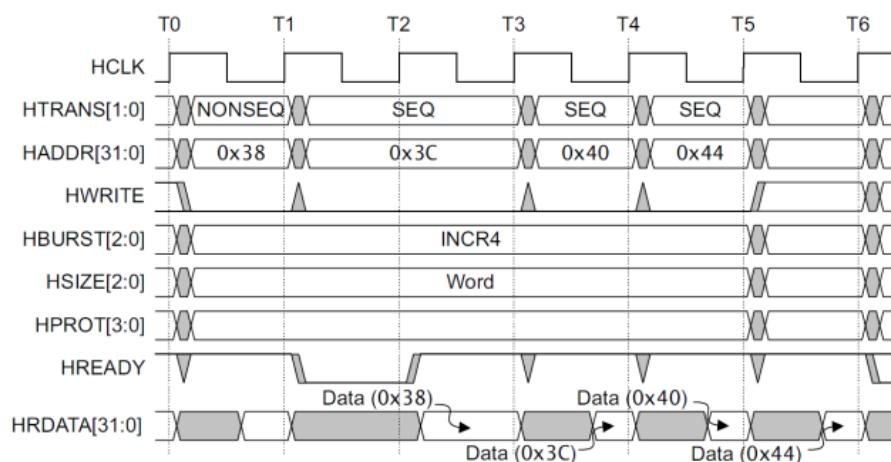


Fig 2.3.10 Transfer incremental (INCR4) cu mărimea transferului egala cu un cuvânt [9]

Răspunsul slaveului

După ce un master inițiază un transfer, acesta monitorizează progresul acestuia și așteaptă un răspuns de la slave-ul țintă. Răspunsul este compus din două semnale: HRESP și HREADYOUT.

Transferul poate fi în trei condiții:

1. Transfer finalizat: Transferul a fost finalizat imediat cu succes. HREADYOUT este 1 și HRESP este OKAY.
2. Eroare de transfer: Transferul nu a fost finalizat, raportând o eroare. HRESP este ERROR. Sunt necesare două cicluri pentru a recupera de la o eroare, iar al doilea ciclu conține HREADYOUT egal cu 1.
3. Transfer în așteptare: Transferul necesită unul sau mai multe cicluri de ceas (wait states) pentru a fi finalizat. HREADYOUT este 0, iar HRESP este setat la OKAY la sfârșitul transferului. Numărul recomandat de cicluri de așteptare este maxim 16, pentru a evita degradarea performanțelor.

Astfel, master-ul poate interpreta semnalele HRESP și HREADYOUT pentru a determina starea și rezultatul transferului și să întreprindă acțiunile corespunzătoare în funcție de acestea.

2.3 Verificarea de design

2.3.1 Context, principii și metodologii

Rolul unui inginer de verificare și fluxul de lucru în proiectarea circuitelor integrate

Introducere

În domeniul proiectării circuitelor integrate semiconductoare (IC), verificarea funcțională joacă un rol critic în asigurarea corectitudinii și fiabilității proiectelor digitale complexe. Acest document oferă o explorare detaliată a rolului unui inginer de verificare și a fluxului de lucru implicat în verificarea funcțională, concentrându-se în special pe utilizarea SystemVerilog și a Metodologiei de verificare universală (UVM).

Rolul unui inginer de verificare

În dezvoltarea unui sistem hardware complex, prezența unui număr egal de proiectanți și verificatori este esențială. Inginerii de verificare constituie adesea o parte semnificativă, aproximativ 60-70%, din ciclul de dezvoltare datorită importanței verificării funcționale. Rolul unui inginer de verificare necesită o gamă largă de abilități și competențe pentru a naviga în procesul complicat.

Înțelegerea specificațiilor și a designului

Un inginer de verificare nu trebuie doar să înțeleagă specificațiile, ci și să aibă capacitatea de a analiza proiectele dintr-o perspectivă distinctă. În timp ce proiectanții se concentrează în primul rând pe modelarea designului pentru a îndeplini cerințele specificate, verificatorii sunt responsabili pentru examinarea designului din toate unghiurile posibile, inclusiv scenarii care nu sunt descrise în mod explicit în specificații. Luând în considerare toate cazurile potențiale, chiar și cele aparent infinite ca număr, inginerii de verificare se străduiesc să evite timpul prelungit de lansare pe piață, asigurând în același timp integritatea designului.

Retrageri multiple și verificarea designului

Dezvoltarea cipurilor complexe implică adesea mai multe iterații, sau tapeouts, înainte de prezentarea finală a cipurilor către clienți. Verificarea proiectării este indispensabilă în acest context. Întrucât proiectarea cipurilor rămâne un proces creativ cu imperfecțiuni inerente, iar fabricarea cipurilor este un efort care necesită mult resurse, verificarea designului ajută la minimizarea riscurilor și garantează producția de cipuri de înaltă calitate.

Relația de proiectare și verificare

Procesul de proiectare începe cu specificații, subliniind funcționalitatea dorită a cipului. Un proiectant rafinează specificațiile prin diferite etape, inclusiv specificația funcțională, descrierea algoritmică, nivelul de transfer de regisztru (RTL), lista de rețea de poartă, lista de rețele de tranzistori și, în sfârșit, aspectul. Designerii iau decizii critice în fiecare etapă, selectând descrierea adecvată care duce la o realizare concretă. Este important de reținut că implementările multiple la nivel de poartă pot reprezenta același design RTL.

Verificarea ca proces în sens opus

Spre deosebire de procesul de proiectare, verificarea începe cu o implementare și verifică dacă specificațiile sunt îndeplinite. Verificarea este necesară în fiecare etapă a procesului de proiectare, asigurând consistența și corectitudinea. De exemplu, este crucial să se verifice dacă o descriere algoritmică se potrivește cu specificația funcțională sau că un aspect reflectă cu acuratețe lista de porți. Verificarea proiectării cuprinde o gamă largă de activități, inclusiv verificarea funcțională, verificarea timpului, verificarea aspectului și verificarea electrică.

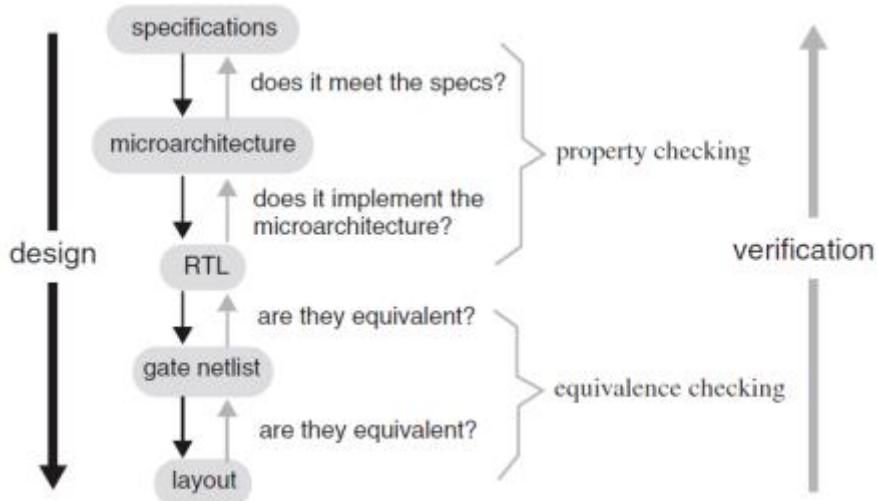


Fig 2.3.1.1 Relație între design și verificarea designului [10]

Tipuri de verificare funcțională

Verificarea funcțională se concentrează pe asigurarea faptului că proiectarea funcționează corect conform specificațiilor. În mod obișnuit, sunt efectuate mai multe tipuri de verificări funcționale, inclusiv:

1. Verificarea echivalenței:
 - Verifică dacă două versiuni ale unui design sunt echivalente din punct de vedere funcțional.
 - Verificarea echivalenței poate fi aplicată între proiecte la același nivel de abstractizare sau niveluri diferite, cum ar fi implementările RTL și la nivel de poartă.
 - Verificarea implementării și verificarea modelului/proprietății:
2. Verifică dacă implementarea proiectării îndeplinește specificațiile și respectă cerințele definite de model.
 - Această verificare este esențială atunci când se face tranziția între diferite niveluri de abstractizare, deoarece sunt introduse detalii de nivel inferior care pot să nu fie specificate în mod explicit la nivelurile superioare.

Erori în proiectare și verificare

Inginerii de verificare sunt responsabili pentru identificarea a două tipuri principale de erori: erori de implementare și erori legate de specificații.

1. Erori de implementare:

- Erorile de implementare apar în timpul fazei de implementare a proiectării, rezultate din greșeli umane sau erori ale instrumentelor software.
- Tehnicile de redundanță, cum ar fi implementarea acelorași specificații în moduri diferite și compararea rezultatelor, sunt utilizate în mod obișnuit.
- Utilizarea diferitelor limbaje descriptive, cum ar fi VHDL/Verilog pentru proiectare și SystemVerilog/C/C++ pentru verificare, minimizează riscul de propagare a erorilor.

2. Erori legate de specificații:

- Erorile legate de specificații apar din cerințe conflictuale, caracteristici nerespectate sau detalii funcționale nespecificate.
- Deoarece aceste erori apar în vârful ierarhiei de abstractizare, verificarea directă bazată pe model nu este fezabilă.
- Întâlnirile regulate, revizuirile de proiectare și comunicarea continuă între echipa de ingineri sunt cruciale pentru abordarea problemelor arhitecturale.
- probleme și rezolvarea cerințelor nerealizabile sau conflictuale.

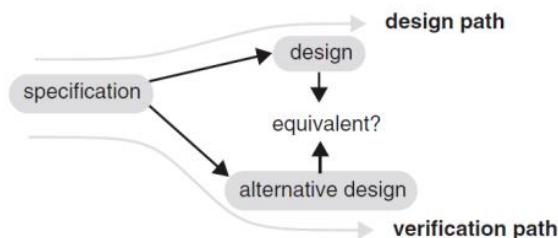


Fig 2.3.1.2 Verificarea prin redundanță [10]

Metodologii

Verificarea bazată pe simulare

Verificarea bazată pe simulare constă în utilizarea unei bănci de teste, unde se plasează designul și se aplică stimuli de intrare, iar ieșirile designului (Design Under Test) sunt colectate și comparate cu cele așteptate. Atât intrările cât și ieșirile așteptate pot fi calculate în avans și încărcate în timpul simulării sau pot fi generate dinamic. În primul rând, un design este de obicei supus analizei statice a codului, numită lint, care caută erori și încălcări potențiale, cum ar fi un bus fără un driver, lățimea unui port a unei instanțe care nu se potrivește cu definiția modulului sau intrări în așteptare ale unei porți. În ceea ce privește generarea vectorilor de intrare, se poate vorbi despre teste direcționate, când vectorii de intrare sunt aleși direct de către ingineri. Problema cu această abordare este că limitează alegerea doar pe baza cunoștințelor lor, ceea ce adesea lasă neexplorate regiuni și, prin urmare, cu potențiale erori. Pentru a evita acest lucru, de obicei se furnizează o valoare de

inițializare (seed) și se produc vectori aleatori de intrare și ieșiri așteptate, având astfel teste și în jurul testelor direcționate. Aceste teste sunt utilizate de un simulator care efectuează simulările. Există multe astfel de simulațioare pe piață, care pot fi simulațioare hardware, software bazate pe cicluri sau bazate pe evenimente. Un simulator bazat pe evenimente efectuează evaluări ori de câte ori intrările unei porți sau variabilele de sensibilitate se schimbă într-un bloc de instrucțiuni. Schimbarea este numită eveniment. Un simulator bazat pe cicluri partionează un circuit și evaluează la fiecare margine a impulsului de ceas. Clar, acest tip de simulator poate fi utilizat doar pentru circuite sincrone. În schimb, un simulator hardware utilizează elemente hardware precum FPGA-uri (Field Programmable Gate Arrays) pentru a modela designul, adică acestea sunt programate pentru a emula comportamentul porților designului. Utilizând simularea, se obțin ieșiri și, dacă acestea nu se conformează celor așteptate, se investighează cauza problemei. Aceasta se face prin analizarea codului și verificarea semnalelor prin observarea schimbărilor valorilor asumate de variabile în timp, identificând anomalii. De obicei, când se găsește o eroare, aceasta este comunicată designerului pentru a o remedie. Acest lucru se face utilizând un sistem de urmărire a erorilor (bug tracking system), care informează persoana responsabilă de design și urmărește progresul acestuia, de la verificare până la rezolvare. În proiecte mari, se utilizează și software-uri de control al reviziei pentru a arbitra gestionarea între mai mulți utilizatori, astfel încât fișierele să fie întotdeauna accesibile tuturor membrilor echipei și să se evite pierderea modificărilor.

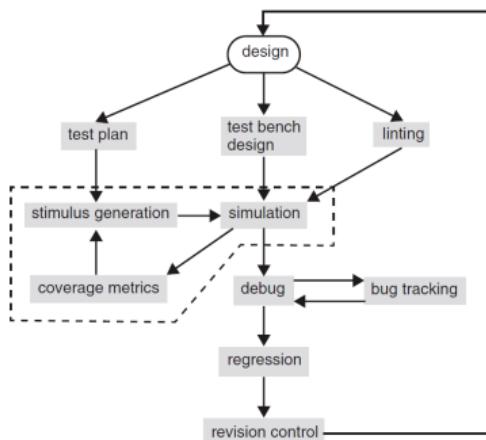


Fig 2.3.1.3 Verificare bazată pe simulare [10]

Verificarea formală

Deși această metodă nu a fost adoptată pentru verificarea AHB, merită menționată în scopul completitudinii. În verificarea bazată pe metode formale nu este nevoie de crearea de vectori de testare și se deosebește în: verificare de echivalență și verificare de proprietăți. Verificarea de echivalență verifică dacă două implementări sunt echivalente din punct de vedere funcțional. Există două abordări pentru abordarea acestei probleme:

1. Prima, numită SAT (satisfabilitate), constă în găsirea sistematică a unui vector de intrare pentru care cele două circuite diferă. [10]
2. A doua abordare constă în reprezentarea celor două circuite (funcțiile logice) într-o formă canonică, de exemplu, ROBDD (Reduced Ordered Binary Decision Diagram) și verificarea dacă sunt echivalente, ceea ce înseamnă că cele două reprezentări sunt izomorfe. Un exemplu de aplicare poate fi verificarea integrității unui aspect fizic al unui IC (Integrated Circuit) în comparație cu versiunea sa RTL. Verificarea poate fi realizată extrăgând netlistul de tranzistori din aspectul fizic și comparându-l cu versiunea RTL.

Dacă o verificare eșuează, atunci se generează un vector de intrare care demonstrează că cele două circuite sunt diferite. Apoi, verificatorul poate înțelege dacă eșecul se datorează unei erori reale sau dacă au fost stimulat condiții neintenționate. Un alt doilea tip de verificare se numește verificare de proprietăți. Aceasta presupune luarea unui design și a unei proprietăți și demonstrarea faptului că designul o respectă sau nu. Un program care verifică această condiție se numește verificator de modele. Ideea este de a explora întregul spațiu de stări în căutare de puncte care să facă ca proprietatea să eșueze. Acest punct, dacă există, este un contraexemplu și, prin analizarea semnalelor, verificatorul poate depăși eșecul. Pentru a evita erori care pot să nu fie relevante, este necesar să se restrângă explorarea spațiului stărilor și a intrărilor presupuse. De exemplu, un design inserat într-un sistem complex poate să nu presupună toate stările sau valorile de intrare posibile, astfel că ar putea apărea erori irelevante. Cu toate acestea, stabilirea constrângерilor nu este trivială și poate necesita eforturi considerabile. O altă abordare se numește demonstrare de teoreme și folosește metode deductive. O proprietate este specificată ca o propoziție matematică, iar designul este descris cu un set de axiome, adică entități matematice. Ideea este de a putea determina dacă propoziția poate fi dedusă din axiome. Dacă este posibil, atunci proprietatea este demonstrată. Această abordare necesită o cunoaștere profundă a funcționării interne a uneltelor și familiarizare cu procesul matematic de demonstrare, dar poate accepta proprietăți mai complexe. [10]

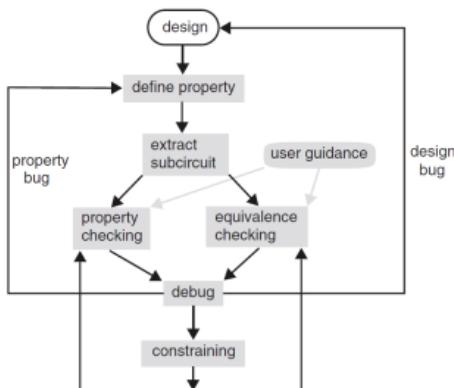


Fig 2.3.1.4 Verificare formala [10]

2.2.2 SystemVerilog

SystemVerilog este un limbaj de descriere hardware și verificare utilizat pe scară largă în industria semiconductorilor pentru proiectarea și verificarea sistemelor digitale complexe. Aceasta combină caracteristicile limbajului Verilog HDL cu construcții puternice pentru proiectare și verificare, devenind astfel o unealtă esențială pentru inginerii implicați în procesul de verificare. Această secțiune explorează utilizarea SystemVerilog și evidențiază caracteristicile sale importante, inclusiv Programarea Orientată pe Obiecte (OOP), cu accent pe simularea bazată pe evenimente.

SystemVerilog pentru verificare

SystemVerilog oferă mai multe funcționalități care îmbunătățesc procesul de verificare și măresc eficiența și fiabilitatea mediului de verificare. Aceste funcționalități includ: [12]

- Verificare bazată pe aserțiuni (ABV): SystemVerilog suportă aserțiuni, care sunt declarații utilizate pentru a specifica proprietăți care trebuie să fie valabile în timpul

simulării. Asetările permit inginerului de verificare să definească cerințele funcționale și să verifice automat dacă acestea sunt îndeplinite. Aceasta reduce efortul manual și îmbunătățește eficacitatea detectării erorilor.

- Acoperire funcțională: SystemVerilog include construcții pentru specificarea acoperirii funcționale, care măsoară gradul de acoperire al procesului de verificare. Prin definirea obiectivelor de acoperire și colectarea datelor de acoperire în timpul simulării, inginerii pot asigura că toate scenariile relevante au fost testate, crescând încrederea în corectitudinea designului.
- Verificare cu generare aleatoare restrânsă: SystemVerilog suportă generarea aleatoare restrânsă, permitând inginerilor să genereze o gamă largă de stimuli de intrare pentru testare. Prin definirea constrângerilor asupra variabilelor de intrare, cum ar fi intervalele și relațiile, inginerii pot asigura că cazurile de test generate acoperă diferite scenarii în mod eficient. Această abordare ajută la descoperirea cazurilor particulare și a erorilor potențiale care pot fi trecute cu vederea în cazul testării dirijate exclusiv.
- Interfață de Programare Directă (DPI): Funcționalitatea DPI în SystemVerilog permite integrarea cu alte limbaje de programare, cum ar fi C sau C++, permitând inginerilor de verificare să folosească biblioteci și instrumente software existente. Acest lucru facilitează crearea de bănci de teste complexe și permite verificarea eficientă a sistemelor hardware-software mixte.
- Programare Orientată pe Obiecte (OOP): SystemVerilog suportă conceptele de programare orientată pe obiecte, cum ar fi clasele și obiectele, care permit proiectarea modulară și reutilizabilă a codului. OOP în SystemVerilog promovează încapsularea, moștenirea și polimorfismul, permitând o mai bună organizare și gestionare a componentelor de verificare. Prin utilizarea OOP, inginerii de verificare pot construi arhitecturi ierarhice pentru băncile de teste, pot crea componente de verificare reutilizabile și pot îmbunătăți întreținerea și scalabilitatea codului.

Simularea bazată pe evenimente

Simularea bazată pe evenimente este un aspect esențial al verificării SystemVerilog și joacă un rol crucial în modelarea precisă a comportamentului sistemelor digitale. În simularea bazată pe evenimente, evaluările au loc ori de câte ori apare o schimbare în intrările sau variabilele de sensibilitate ale unui bloc. Iată câteva aspecte importante ale simulării bazate pe evenimente: [12]

- Sensitivity List: SystemVerilog utilizează o listă de sensibilitate pentru a specifica intrările sau variabilele care declanșează evaluările atunci când acestea se schimbă. Prin definirea corespunzătoare a listei de sensibilitate, designerii pot asigura că simularea reflectă în mod precis comportamentul dorit al designului.
- Modelarea Evenimentelor: În simularea bazată pe evenimente, evenimentele reprezintă schimbările în valorile semnalelor sau atribuirile variabilelor. SystemVerilog oferă construcții, cum ar fi blocurile procedurale și sarcinile de sistem, pentru a modela evenimentele în mod eficient. Acest lucru permite designerilor să surprindă cu precizie comportamentul secvențial și concurrent al designului.
- Simulare bazată pe timp: SystemVerilog suportă simularea bazată pe timp, în care pot fi introduse întârzieri pentru a modela întârzierile de propagare și constrângerile de timp. Prin încorporarea informațiilor de sincronizare în simulare, designerii pot evalua performanța și comportamentul temporal al designului.
- Debug și analiză a formelor de undă: În timpul simulării bazate pe evenimente, designerii pot analiza forme de undă pentru a identifica și depărtă problemele. SystemVerilog oferă instrumente și funcționalități pentru vizualizarea și analiza

formelor de undă, permitând inginerilor să urmărească modificările în valorile semnalelor în timp și să identifice eventualele discrepanțe sau probleme.

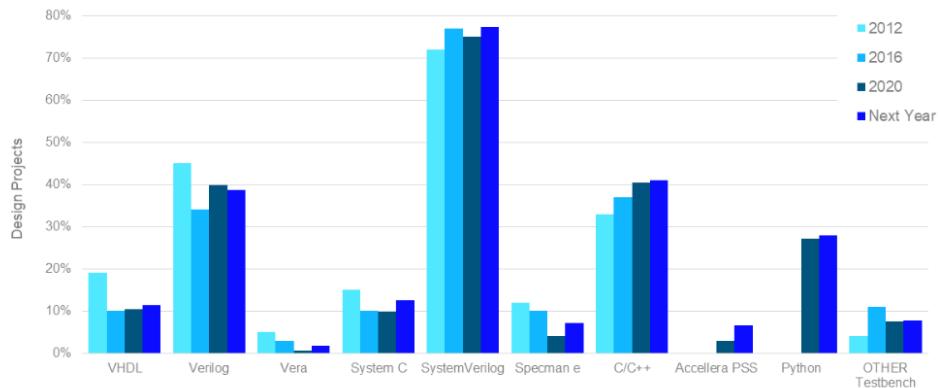


Fig 2.2.2.1 Trenduri HVL pentru verificare de ASIC [11]

ASIC/IC Design Language Adoption

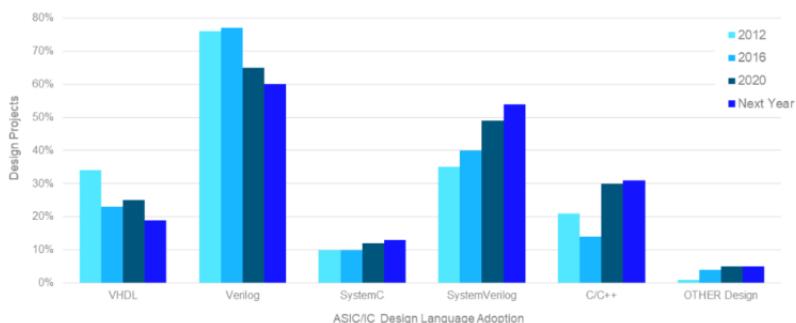


Fig 2.2.2.1 Trenduri HDL pentru design de ASIC [11]

2.2.3 UVM

UVM Factory

UVM Factory este o componentă fundamentală în metodologia UVM care facilitează crearea și configurarea dinamică a obiectelor. Aceasta permite testbench-ului să creeze instanțe ale componentelor UVM pe baza tipurilor acestora și oferă o modalitate flexibilă de a configura aceste componente cu parametri specifici. Fabrica este responsabilă de gestionarea relațiilor ierarhice dintre componente, permitând comunicarea și colaborarea eficientă între diferitele părți ale bancului de testare. [13]

UVM Factory funcționează pe baza unui mecanism de înregistrare, în care componente sunt înregistrate la fabrică folosind un nume de tip unic. Când este solicitată o componentă, fabrica își folosește numele tipului pentru a crea în mod dinamic o instanță a acelei componente. Acest lucru permite testbench-ului să instanțieze și să configureze componente în timpul execuției, făcând testbench-ul mai modular și adaptabil la diferite scenarii.

Fazele UVM

Fazele sunt un aspect crucial al metodologiei UVM care oferă o abordare structurată pentru controlul și sincronizarea activităților într-un banc de testare. Fazele definesc o succesiune de etape bine definite care guvernează ordinea și executarea sarcinilor specifice. Standardul UVM definește un set de faze predefinite, cum ar fi construirea, conectarea, rularea și curățarea, care acoperă pașii esențiali ai unui proces de verificare.

Etaparea asigură că diferitele componente ale bancului de testare funcționează într-o manieră coordonată. Fiecare componentă poate executa sarcini specifice în timpul fazelor corespunzătoare, permitând alocarea și sincronizarea eficientă a resurselor. Etaparea permite, de asemenea, un control mai bun asupra calendarului și secvenței evenimentelor din bancul de testare, asigurându-se că componentele sunt inițializate, conectate și operate în ordinea dorită. [15]

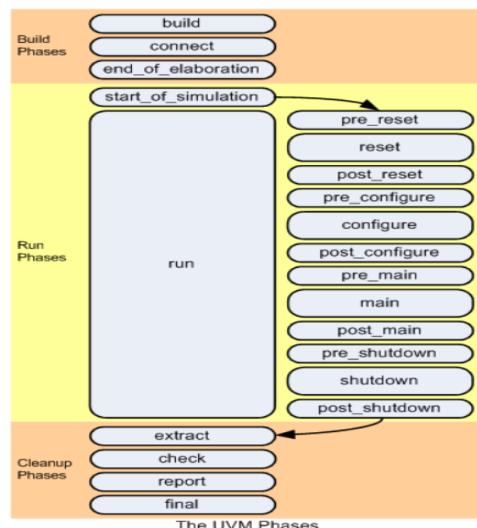


Figura 2.2.2.2 Fazele UVM [13]

Driver UVM

Driverul UVM este o componentă cheie a arhitecturii UVM responsabilă pentru stimularea proiectării testate (DUT). Acesta acționează ca o interfață între bancul de testare și interfețele de intrare ale DUT, transformând stimuli de nivel înalt, la nivel de tranzacție, în tranzacții la nivel de semnal de nivel inferior pe care DUT-ul le poate înțelege.

Driverul UVM funcționează împreună cu UVM Sequencer, care este responsabil pentru generarea secvenței tranzacțiilor care trebuie aplicate DUT. Driverul preia tranzacțiile generate de Sequencer și le conduce pe interfețele corespunzătoare ale DUT. Gestionează sincronizarea și succesiunea tranzacțiilor, asigurându-se că acestea sunt trimise la DUT cu acuratețe și în conformitate cu scenariile de testare definite. [15]

Driverul UVM joacă, de asemenea, un rol critic în colectarea răspunsurilor de la DUT, care sunt apoi transmise înapoi la bancul de testare în scopuri de analiză și verificare. Prin coordonarea generării stimuli și a colectării răspunsurilor, driverul UVM permite o comunicare eficientă între bancul de testare și DUT.

Monitor UVM

Monitorul UVM este o componentă care observă și surprinde activitatea de pe interfețele DUT în timpul simulării. Acesta acționează ca un observator pasiv, monitorizează tranzacțiile care au loc pe interfețele de intrare și ieșire ale DUT și le captează pentru analiză și verificare.

Funcția principală a monitorului UVM este de a intercepta și înregistra tranzacțiile care trec prin interfețele DUT. Captează informații importante, cum ar fi momentul, datele și detaliile specifice protocolului ale tranzacțiilor. Aceste informații capturate pot fi utilizate pentru analiză, colectarea acoperirii funcționale și verificarea protocolului. [15]

Monitorul UVM funcționează de obicei prin conectarea la interfețele DUT prin porturi de analiză sau interfețe virtuale. Monitorizează continuu semnalele sau tranzacțiile de pe interfețe și utilizează tehnici de simulare bazate pe evenimente pentru a detecta și captura tranzacțiile relevante. Tranzacțiile capturate pot fi apoi utilizate pentru analiză, comparare cu comportamentul așteptat și procesare ulterioară de către alte componente ale bancului de testare.

În general, UVM Factory, Phasing, UVM Driver și UVM Monitor sunt componente esențiale ale metodologiei UVM care contribuie la construirea de bancuri de testare modulare, scalabile și reutilizabile. Înțelegerea și utilizarea eficientă a acestor componente poate spori semnificativ eficiența și eficacitatea procesului de verificare funcțională.

Agent UVM

Agentul UVM este o abstractizare la nivel înalt care reprezintă un mediu de verificare autonom pentru un anumit bloc funcțional sau modul al proiectului în curs de testare (DUT). Aceasta încapsulează un set de componente înrudite, cum ar fi driverul UVM, monitorul UVM, secvențiatorul UVM și alte componente necesare pentru verificarea blocului respectiv.

Agentul UVM servește ca o entitate reutilizabilă și configurabilă care poate fi instanțiată de mai multe ori într-un banc de testare pentru a verifica în mod independent diferite părți ale DUT. Oferă o abordare modulară a verificării, permitând integrarea și scalabilitatea mai ușoară a bancului de testare. [15]

Fiecare agent UVM conține propriile instanțe ale driverului UVM și monitorului UVM, permitându-i să genereze stimuli și să capteze răspunsuri în mod specific pentru blocul său asociat. De asemenea, gestionează secvențierea tranzacțiilor prin UVM Sequencer și facilitează comunicarea cu alți agenți sau cu mediul testbench. [13]

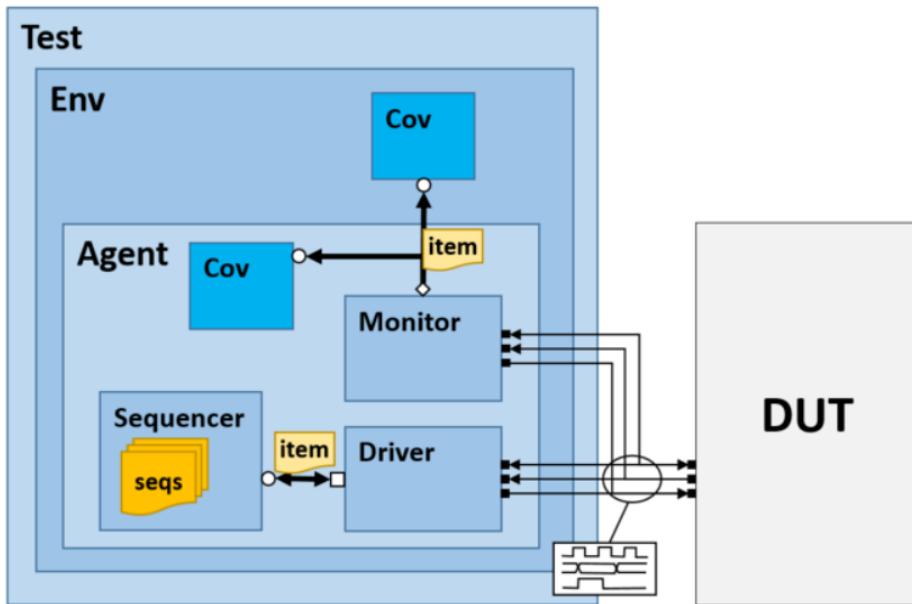


Fig 2.2.2.3 Exemplu arhitectural al unui mediu de verificare UVM [13]

Secvențe UVM

Secvențele UVM sunt componente responsabile pentru generarea unei secvențe de tranzacții care reprezintă un scenariu de testare specific sau un comportament care trebuie aplicat DUT. Ele oferă o modalitate de nivel înalt și reutilizabilă de a descrie modelele de stimuli și scenariile de testare.

Secvențele definesc ordinea, momentul și conținutul tranzacțiilor care urmează să fie generate, inclusiv întârzierile necesare, detaliile specifice protocolului și datele tranzacțiilor. Acestea încapsulează logica scenariului de testare și pot fi randomizate, permitând variații în secvențele de testare și o acoperire crescută a testului. [13]

Secvențele UVM funcționează împreună cu Sequencerul UVM, care gestionează secvențierea și livrarea secvențelor către driverul UVM. Sequencerul asigură că secvențele sunt executate în ordinea dorită și controlează fluxul de tranzacții dintre bancul de testare și DUT.

Elemente de secvență UVM

Elementele de secvență UVM reprezintă tranzacții individuale sau bucăți de date care sunt trimise sau primite de la DUT. Acestea încapsulează informațiile și proprietățile fiecărei tranzacții, cum ar fi valorile datelor, semnalele de control și câmpurile specifice protocolului. Elementele UVM Sequence sunt utilizate de UVM Sequences pentru a construi secvențe semnificative de tranzacții. Acestea oferă o modalitate structurată și reutilizabilă de a defini modelele de tranzacție și de a permite generarea ușoară de stimuli și verificarea răspunsurilor așteptate.

Elementele de secvență UVM pot fi extinse sau specializate pentru a gestiona diferite tipuri de tranzacții și protocole. Ele sunt adesea folosite împreună cu constrângeri pentru a defini intervalele și relațiile permise ale proprietăților tranzacțiilor, permitând generarea eficientă și eficientă de stimulente. [13]

Baza de date de configurare UVM (uvm_config_db)

Baza de date de configurare UVM, cunoscută și sub numele de uvm_config_db, este un depozit centralizat care stochează informații de configurare utilizate pentru a parametriza și configura componentele UVM. Acesta oferă un mecanism flexibil și ierarhic pentru transmiterea și preluarea datelor de configurare în timpul execuției.

Uvm_config_db permite componentelor să acceseze și să partajeze setările de configurare fără cuplare strânsă sau dependențe directe. Oferă o modalitate convenabilă de a configura și configura componente pe baza cerințelor de rulare, făcând bancul de testare mai flexibil și mai adaptabil la diferite scenarii. [15]

Componentele își pot înregistra setările de configurare în baza de date și le pot prelua după cum este necesar în timpul simulării. Acest lucru permite parametrizarea și reconfigurarea ușoară a componentelor fără modificarea codului sursă. Uvm_config_db promovează modularitatea, configurabilitatea și reutilizarea în testbench.

Utilizarea packageurilor

Pachetele sunt o caracteristică importantă în SystemVerilog și sunt utilizate pe scară largă în metodologia UVM. Un pachet este un spațiu de nume care încapsulează o colecție de declarații înrudite, cum ar fi definiții de clasă, definiții de tip, constante, funcții și sarcini. În UVM, pachetele sunt folosite pentru a organiza și grupa componentele UVM, funcțiile utilitare și setările de configurare. Ele oferă o abordare modulară pentru organizarea codului testbench și facilitează reutilizarea și lizibilitatea codului. [12,13]

Prin utilizarea pachetelor, diferite componente și funcționalități UVM pot fi grupate în mod logic.

2.2.4 Unelte folosite

Instrumentele Electronic Design Automation (EDA) joacă un rol crucial în verificarea și implementarea sistemelor digitale. Aceste instrumente ajută la proiectarea, simularea, analiza și sinteza componentelor hardware. Acestea permit inginerilor să verifice corectitudinea, funcționalitatea și performanța proiectelor lor înainte de implementarea fizică, reducând timpul și costul dezvoltării.

Fluxul arhitectural al unui sistem digital cuprinde diverse etape, începând de la specificare și terminând cu implementarea fizică. Fluxul presupune mai multe etape, inclusiv explorarea arhitecturală, sinteza la nivel înalt, proiectarea RTL, verificarea funcțională, sinteza, locul și traseul și, în final, verificarea fizică. Fiecare etapă se concentrează pe diferite aspecte ale procesului de proiectare, asigurându-se că proiectul îndeplinește specificațiile dorite și poate fi implementat eficient.

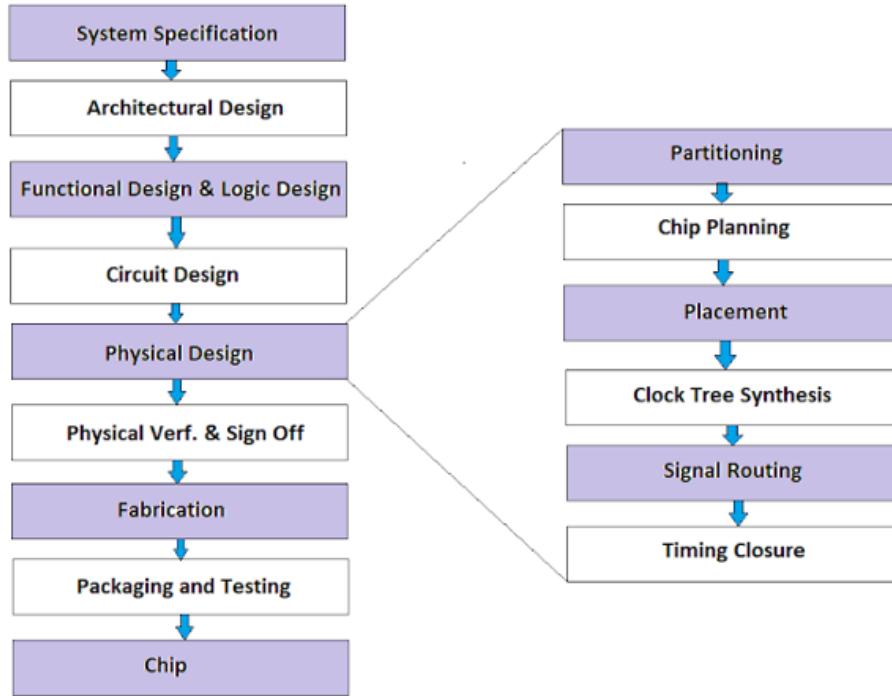


Fig 2.2.4.1 Fluxul designului unui ASIC [17]

Pe parcursul acestui flux arhitectural, instrumentele EDA oferă suport critic. Acestea permit inginerilor să modeleze și să simuleze sistemul la diferite niveluri de abstractizare, să analizeze comportamentul și performanța designului și să efectueze diferite forme de verificare pentru a asigura corectitudinea acestuia.

Visual Studio: Visual Studio este un mediu de dezvoltare integrat (IDE) utilizat pe scară largă pentru dezvoltarea de software. În proiectul dvs., acesta a servit probabil ca instrument principal pentru codare și dezvoltare, oferind funcții precum editarea codului, depanarea și managementul proiectelor.

Git: Git este un sistem popular de control al versiunilor folosit pentru urmărirea modificărilor codului sursă în timpul dezvoltării software. Permite mai multor dezvoltatori să colaboreze la un proiect, să gestioneze diferite versiuni ale codului și să urmărească modificările în timp. Git ajută la asigurarea integrității codului și oferă un istoric al modificărilor aduse proiectului.

SimVision: SimVision este un vizualizator de forme de undă de simulare și un instrument de analiză utilizat în mod obișnuit în industria EDA. Oferă o reprezentare grafică a rezultatelor simulării, permitând inginerilor să analizeze și să depaneze comportamentul proiectelor lor. SimVision permite vizualizarea formei de undă, urmărirea semnalului și identificarea problemelor sau discrepanțelor în proiectare.[19]

Xcelium: Xcelium este un simulator de înaltă performanță oferit de Cadence Design Systems. Acceptă limbaje de descriere hardware, cum ar fi Verilog, VHDL, SystemVerilog și SystemC. Xcelium oferă capabilități avansate de simulare, permitând inginerilor să verifice cu exactitate funcționalitatea și performanța proiectelor lor.[19]

vManager: vManager este un instrument de analiză a acoperirii care ajută la măsurarea eficacității eforturilor de verificare. Acesta colectează și analizează datele de acoperire, permitând inginerilor să evaluateze cât de amănunțit a fost testat designul. vManager permite

identificarea zonelor netestate și oferă perspective asupra calității și completității procesului de verificare.[19]

3. Verificare funcțională unui design de arbitru AHB

Procesul de verificare funcțională se desfășoară de obicei în paralel cu procesul de proiectare și necesită ca inginerii să citească specificațiile și să creeze un plan de verificare pentru a verifica dacă dispozitivul își atinge obiectivul propus. Acesta este alcătuit din patru etape:

1. Scrierea unui plan de verificare.
2. Implementarea unui mediu de verificare.
3. Realizarea unei activități de punere în funcțiune a dispozitivului.
4. Efectuarea uneia sau mai multor regresii ale dispozitivului.

În cele ce urmează, aceste patru etape vor fi prezentate, descriind o abordare generică pentru un circuit integrat. Aceasta diferă ușor de verificarea unui IP reutilizabil, cum ar fi AHB, iar diferențele vor fi menționate când este necesar.

3.1 Planul de verificare

Un plan de verificare este un document redactat în limbaj natural, de obicei în limba engleză, care exprimă două scopuri principale: ce trebuie verificat și cum trebuie verificat. Pentru a cuantifica verificarea unui component, se adoptă metrii de acoperire bazate pe elementele specificațiilor care trebuie implementate în cadrul mediului de verificare.

Extragerea caracteristicilor În primul rând, are loc o analiză a specificațiilor de proiectare, în care se extrag cele mai importante caracteristici ale designului, urmând abordări de sus în jos sau de jos în sus. Acest proces necesită utilizarea tuturor documentelor disponibile, de la specificațiile principale de proiectare la informații de piață și standarde pe care se bazează arhitectura.

În practică, acest lucru poate însemna parcurgerea analizei:

- Specificațiile de proiectare: specificațiile unui circuit integrat (IC).
- Specificațiile cerințelor: document tehnic care specifică modul în care un sistem este proiectat pe baza documentațiilor de cerințe de afaceri.
- Specificațiile de proiectare la nivel de bloc sau documentul de proiectare la nivel înalt: document care oferă o vedere de ansamblu asupra sistemului.

Functionalitatile designului

- Transferuri de date în conformitate cu protocolul AHB
- Decodarea adresei slaveului
- Arbitrare

Arhitectura mediului de verificare

Mediul constă într-un număr variabil de N agenți master și M agenți slave. Numărul poate fi modificat prin schimbarea parametrilor din integration_pkg.sv.

Agenții master sunt activi, asta înseamnă ca ei fac drive la elementele către interfață și, de asemenea, le monitorizează și le trimit la Scoreboard.

Agenții slave sunt reactivi, ei raspund la cererile masterilor. Au un monitor care verifică transferurile valide pe interfață, apoi trimit cererea la sequencer și începe o secvență care este răspunsul pentru master pe Driver. O imagine care prezintă mediul poate fi găsită în figura următoare.

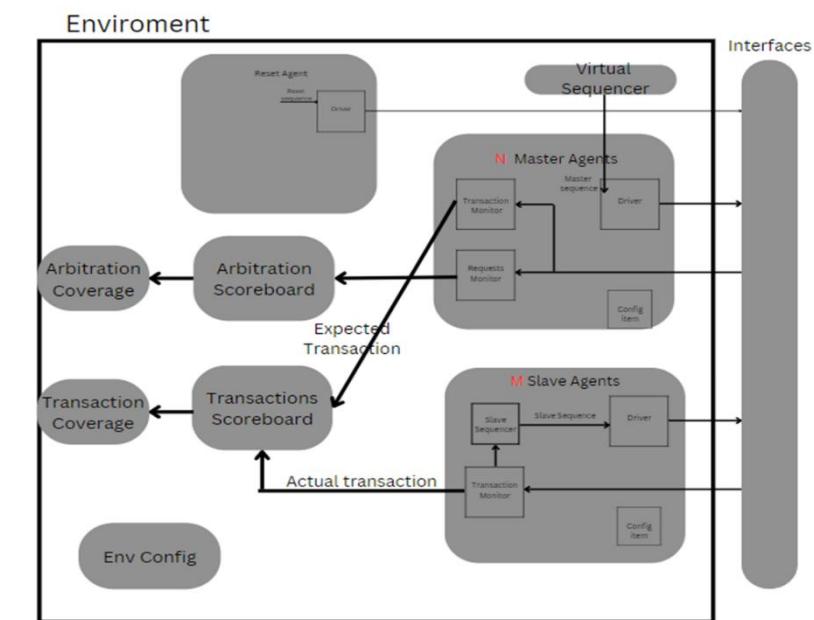


Fig 3.1.1 Arhitectura mediului de verificare

Checkere

Checkerele în verificare reprezintă componente esențiale care au rolul de a verifica corectitudinea și conformitatea comportamentului designului cu specificațiile. Acestea sunt responsabile pentru analizarea și validarea semnalelor, datelor și evenimentelor în timpul simulării.

Rolul principal al checkerelelor este de a compara valorile semnalelor și datele generate de design cu valorile așteptate definite în specificații. Ele sunt concepute pentru a detecta și raporta erori, discrepanțe sau devieri de la comportamentul dorit. Acest lucru ajută la identificarea potențialelor probleme în timpul procesului de verificare și la asigurarea calității și fiabilității designului.

Există două tipuri de checkere care trebuie implementate, checkere de protocol și checkere cross interface. Checkerele de protocol vor fi implementate în agenți, în timp ce checkerele cross interface vor fi implementate în scoreboard, deoarece este necesară informație suplimentară

Checkere de protocol

Checkerele de protocol sunt legate de configurația semnalelor de interfață.

Checker Name	Feature	Description
RISING_EDGE_OPERATION	Clock	If difference between the simulation time when a clock posedge happened and @ (signal) > clocking block delay then DUT works on single edge.
NON_TRISTATE_IMPLEMENTATION	Non-tristate implementation	If the inputs to DUT are different than x or z, the outputs should be different than x or z.
INCR_ADDR	AHB protocol	Address Check for INCR/INCRx transfer Address = Address of previous transfer + size in bytes.
ONE_KB	AHB protocol	1KB boundry for INCR burst.
WRAP4_WORD_ADDR	AHB protocol	Address Check for htrans = WRAP4 and hsize= WORD.
WRAP8_WORD_ADDR	AHB protocol	Address Check for htrans = WRAP8 and hsize= WORD.
WRAP16_WORD_ADDR	AHB protocol	Address Check for htrans = WRAP8 and hsize= WORD.
ADDR_ALIGNMENT_WORD	AHB protocol	Check for address alignment for word.
ADDR_ALIGNMENT_HALFWORD	AHB protocol	Check for address alignment for halfword.
SINGLE_NO_BUSY	AHB protocol	NONSEQ Single transfers should not be followed by BUSY or SEQ.
SAME_CTRL_SIG	AHB protocol	Control signals are identical to the first transfer .
ADDR_PHASE_DURATION	AHB protocol	Address lasts for a single HCLK cycle unless its extended by the previous bus transfer
FIRST_HTRANS_VALUE	AHB protocol	Check that HTRANS is NONSEQ when it is a single transfer or first in a burst.
BURST_HTRANS_VALUE	AHB protocol	Check that HTRANS is different than NONSEQ

		when it is a burst transfer. (except first cycle)
WAITED_SAME_TRANSFER_TYPE	AHB protocol	When hready == 0, the master must not change transfer type . (except for IDLE or BUSY)
NO_BUSY_AFTER_SINGLE	AHB protocol	After single transfer, we can't have htrans BUSY.
IDLE_BUSY_RESPONSE	AHB protocol	Slave response is OKAY to IDLE or BUSY.
SLAVE_WAIT_STATES	AHB protocol	Check that when a slave inserts a number of wait states prior to completing the response, it must drive HRESP to OKAY.

Cross-Interface Checkers

Checkerele cross interface se bazează pe un flux cauză-efect, astfel încât o configurație a semnalelor pe intrări are ca rezultat o configurație a semnalelor de ieșire. Această listă de verificatori include și verificarea integrității datelor.

Checker Name	Feature	Description
Arbitration checkers		
DEFAULT_BUS_MASTER	Arbitration	When no master requests the bus, the master lowest priority receive the bus . (increments from master_number to 0).
ONLY_ONE_HGRANT	Arbitration	Only one master should own the bus at a time so only one hgrant is active at a time.
ARBITRATION_SCOREBOARD	Arbitration	The master is granted according to the priority mechanism considering locked transfers also.
HMASTER_CHECK	Arbitration – auxiliary signal	When a master is granted the bus (HGRANT & HREADY == 1), HMASTER changes after one clock cycle.
HMASTLOCK_TIMING	Arbitration – auxiliary signal	If the master is granted, it's hlock is propagated combinational to hmastlock.
Data transfer checkers		
AHB_TRANSFER_SCOREBOARD	Data transfer	Checks that the data transfer happened correctly and according to AHB protocol.
Slave address decoding checkers		

SLAVE_ADDR_DECODING	Slave address decoding	The slave is selected based on the address range. It was implemented in AHB_TRANSFER_SCORE BOARD
ONLY_ONE_SELECTED_SLAVE	Slave address decoding	Only one slave is selected at a time.
IDLE_BUSY_NONEXISTENT_LOCATIONS	Slave address decoding	An IDLE or BUSY transfers to nonexistent locations result in a zero wait state OKAY response.
DEFAULT_SLAVE_FOR_NONEXISTENT_LOCATIONS	Slave address decoding	Check that If a NONSEQUENTIAL or SEQUENTIAL transfer is attempted to a nonexistent address location then the default slave provides an ERROR response.

Planul pentru coverage

Coverage/Acoperirea funcțională este un aspect critic al procesului de verificare funcțională în domeniul proiectării electronice. Ea implică măsurarea gradului în care diferite funcționalități ale unui design au fost exercitat sau testate în timpul procesului de verificare. Scopul principal al acoperirii funcționale este de a se asigura că toate caracteristicile și comportamentele importante de proiectare au fost verificate temeinic, reducând riscul erorilor nedetectate sau a lacunelor funcționale.

Verificarea este planificată să acopere toate stările posibile ale proiectului și să permită toate tranzițiile valide ale stărilor.

Coverage Item : ahb_master_cg	Feature	Possible Values	Description
read_write	Data transfer – Read or Write	WRITE or READ	
haddr	Data transfer	5 address bin ranges	
htrans	Data transfer	SEQ OR NONSEQ	
hburst	Data transfer	3 bins : (increment, wrap and single)	

hsize	Data transfer – Data size	3 bins (BYTE, HALFWORD , WORD)	
hwdata	Data transfer	6 evenly distributed bins	
read_writeXburstXsizeXtrans	Data transfer	$2 * \text{hsize} * \text{hburst} * \text{htrans}$	Combination of different transfer sizes along with direction.
Coverage Item : ahb_slave_cg	Feature	Possible Values	Description
hrdata	Data transfer	6 evenly distributed bins	
hready	Data transfer	1	For the 0 value, we can check RTL Code Coverage because sampling is made only when hready is 1.
hresp	Data transfer	OKAY , ERROR	
hrespXhready	Data transfer	(1,OKAY),	
hsel	Address Decoding		Because we have bins to sample the addresses we sent to slave. We can check on RTL Code Coverage if the corresponding slaves were selected.

Coverage Item : arbitration_cg	Feature	Possible Values	Description
hbusreq	Arbitration	9 bins corresponding the master number.	Check to see if all masters requested the bus.
busreqXhlock	Arbitration	9 bins.	Check to see if all masters locked the bus.
hgrant	Arbitration		Because we have bins to sample the busreq we can check on RTL Code Coverage if the corresponding masters had the bus.

Testele

Testele sunt scenarii pe care intenționăm să le implementăm pentru a exercita designul AHB Arbiter. Am împărțit lista de teste în câteva grupuri pentru a organiza mai bine verificarea.

Teste de bază

Testele de bază sunt utilizate pentru stabilizarea mediului și pentru testarea funcționalității de bază a designului. O listă cu testele planificate și descrierea scopului acestora și descrierea scenariului este dată în tabelul de mai jos.

N o.	Test Name	Test Purpose	Feature	Description
1.	single_write_t est	The test purpose is to exercise the single write functionality of the design.	Single write + Arbitration + Address decoding	
2.	single_read_t est	The test purpose is to exercise the single read functionality of the design.	Single read + Arbitration + Address decoding	
3.	incr_write_4_ test	The test purpose is to exercise increment type transfers.	INCR4 + Arbitration + Address decoding	
4.	incr_write_8_ test	The test purpose is to exercise increment type transfers.	INCR8 + Arbitration + Address decoding	
5.	incr_write_16_ _test	The test purpose is to exercise increment type transfers.	INCR16 + Arbitration + Address decoding	
6.	wrap_write_4_ _test	The test purpose is to exercise wrap transfer type.	WRAP4 + Arbitration + Address decoding	
7.	wrap_write_8_ _test	The test purpose is to exercise read transfer type.	WRAP8 + Arbitration + Address decoding	
8.	wrap_write_1 6_test	The test purpose is to exercise wrap transfer type.	WRAP16 + Arbitration + Address decoding	
9.	incr_read_4_t est	The test purpose is to exercise increment type transfers.	INCR4 + Arbitration + Address decoding	

10	incr_read_8_test	The test purpose is to exercise increment type transfers.	INCR8 + Arbitration + Address decoding	
11	incr_read_16_test	The test purpose is to exercise increment type transfers.	INCR16 + Arbitration + Address decoding	
12	wrap_read_4_test	The test purpose is to exercise wrap transfer type.	WRAP4 + Arbitration + Address decoding	
13	wrap_read_8_test	The test purpose is to exercise wrap transfer type.	WRAP8 + Arbitration + Address decoding	
14	wrap_read_16_test	The test purpose is to exercise wrap transfer type.	WRAP16 + Arbitration + Address decoding	

Teste random

Testele random aduc designul rapid în stări care sunt greu de atins utilizând testarea dirijată. De asemenea, tranzțiile între stări sunt mai ușor acoperite. O listă cu testele planificate și descrierea lor scopului și scenariului este prezentată în tabelul de mai jos.

No.	Test Name	Test Purpose	Feature	Description	
1	randomize_test	The test purpose is to exercise the design in random conditions	Data Arbitration + Arbitration + Address decoding	Random AHB transfer burst combinations	

Teste de eroare

Scopul testelor de eroare este exersarea designului în unele configurații eronate pentru a verifica dacă designul nu se blochează sau nu are un comportament neașteptat. O listă cu testele planificate și descrierea scopului acestora și descrierea scenariului este dată în tabelul de mai jos.

No.	Test Name	Test Purpose	Feature	Description
1.	error_test	The test purpose is to send out of bounds addresses to slaves and see how	Address decoding	The test purpose is to send out of bounds addresses to slaves. (haddr > 350)

		the decoding is handled.		
--	--	-----------------------------	--	--

3.2 Mediul de verificare

În urma planului de verificare, va fi întocmit mediul de verificare în care vor apărea componente sub formă de cod pentru simulare și verificarea funcțională a designului. Arhitectura mediului de verificare este ilustrată în Figura 3.1.1.

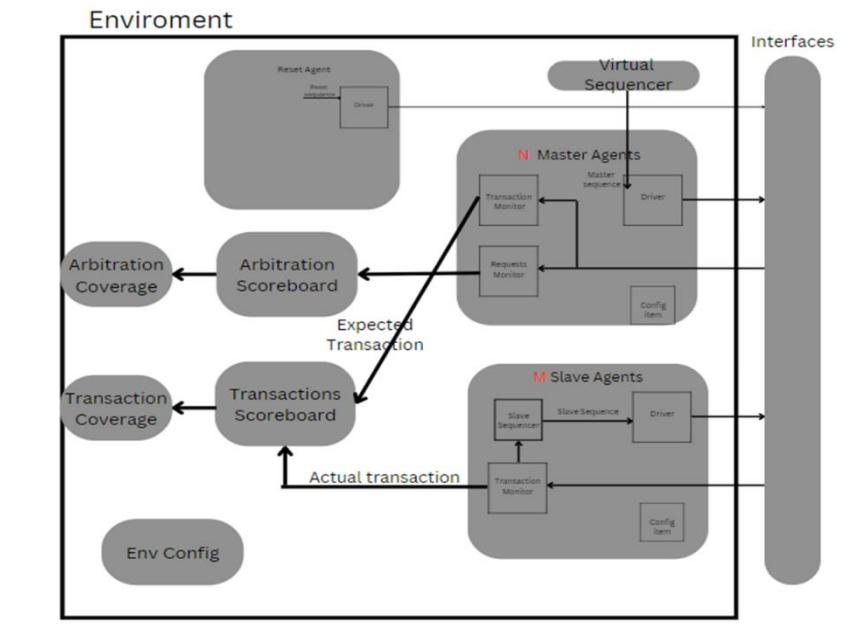


Fig 3.1.1 Arhitectura mediului de verificare

Rolul agenților în cadrul acestui proiect este de a încapsula mai multe funcționalități. În acest caz, ei sunt responsabili de gestionarea tranzacțiilor primite de la sequencer pe interfață. Agenții monitorizează și construiesc tranzacții care vor fi trimise către Scoreboard, unde vor fi comparate cu tranzacțiile așteptate (expected) pentru a verifica corectitudinea transferului datelor.

Masterul trimite o tranzacție, o tranzacție așteptată (expected transaction), care ajunge la Dispozitivul Unit de Testare (DUT), iar răspunsul actual (actual response) al acestuia este capturat de slave.

În Scoreboard-ul de tranzacții, se verifică corectitudinea transferului datelor, decodarea adreselor și selecția slave-ului. În Scoreboard-ul de arbitraj, se verifică dacă cererile de magistrală și arbitrarea sunt realizate conform algoritmului de arbitrare al DUT-ului.

Secvențele care rulează pe sequencerii agenților vor fi controlate prin secvențe și sequencer virtual.

Structura proiectului este următoarea: codul poate fi reutilizat prin preluarea fișierelor *.pkg.sv și prin importarea acestora pentru a reutiliza agenții, mediul de testare (env) și teste.

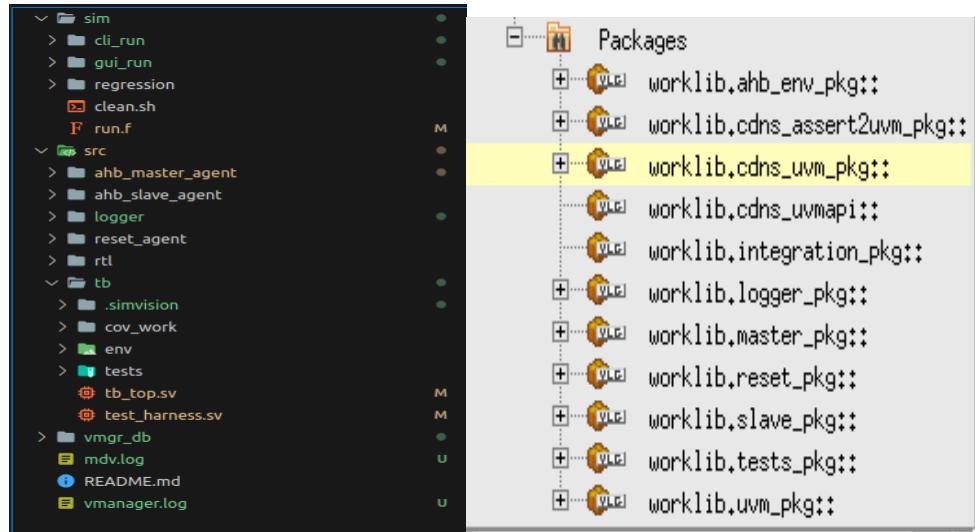


Fig 3.2.1 Packageurile folosite în proiect și structura de directoare

În fișierul run.f, se adaugă parametrii pentru simulare, flaguri pentru UVM și se specifică ce fișiere vor fi compilate și ce test va rula. Acest fișier are rolul de a controla setările și opțiunile de simulare.

```
m > F run.f
1   -UVM
2   -access +rwc
3   -clean
4   -timescale 1ns/1ns
5
6
7   //--linedebug
8   //--debug
9   //--uvmlinedebug
10  -cov_cgsample
11  -coverage all
12  -covoverwrite
13  //--cov_debuglog
14  -svseed random
15
16
17  +UVM_VERBOSITY=UVM_HIGH
18  //+UVM_PHASE_TRACE
19  //+UVM_OBJECTION_TRACE
20
21  /////////////////////////////////
22  // *** include test ***
23
24  //+UVM_TESTNAME=error_test
25
26  //+UVM_TESTNAME=single_write_test
27  //+UVM_TESTNAME=simple_read_test
28
29  //+UVM_TESTNAME=incr_write_test
30  //+UVM_TESTNAME=incr_read_test
31
32  +UVM_TESTNAME=incr_write_4_test
33  //+UVM_TESTNAME=incr_write_8_test
34  //+UVM_TESTNAME=incr_write_16_test
35
36  //+UVM_TESTNAME=incr_read_4_test
37  //+UVM_TESTNAME=incr_read_8_test
38  //+UVM_TESTNAME=incr_read_16_test
39
40  //+UVM_TESTNAME=wrap_write_4_test
41  //+UVM_TESTNAME=wrap_write_8_test
42  //+UVM_TESTNAME=wrap_write_16_test
43
44  //+UVM_TESTNAME=wrap_read_4_test
45  //+UVM_TESTNAME=wrap_read_8_test
46  //+UVM_TESTNAME=wrap_read_16_test
47
48  //+UVM_TESTNAME=random_test
```

Fig 3.2.2 flaguri pentru simulator și teste

```

49
50 // /////////////////////////////////
51 // *** include compile files ***
52
53
54
55
56 -incdir ../../src
57 -incdir ../../src/ahb_master_agent
58 -incdir ../../src/ahb_slave_agent
59 -incdir ../../src/reset_agent
60 -incdir ../../src/tb/env
61 -incdir ../../src/tb/
62 -incdir ../../src/tb/tests
63 -incdir ../../src/logger
64
65 ../../src/logger/logger_pkg.sv
66 #../../src/rtl/integration_pkg.sv
67 ../../src/rtl/integration_pkg.sv
68 ../../src/ahb_master_agent/master_pkg.sv
69 ../../src/ahb_slave_agent/slave_pkg.sv
70 ../../src/reset_agent/reset_pkg.sv
71 ../../src/tb/env/ahb_env_pkg.sv
72 ../../src/tb/tests/tests_pkg.sv
73 ../../src/reset_agent/reset_if.sv
74 ../../src/ahb_master_agent/request_if.sv
75 ../../src/ahb_master_agent/data_transfer_if.sv
76 ../../src/rtl/generic_arbiter_comb.sv
77
78 ../../src/tb/test_harness.sv
79 ../../src/tb/tb_top.sv

```

Fig 3.2.3 makefileul proiectului

Punctul de pornire al codului este funcția "run_test()", care este apelată din modulul "top". În această funcție, designul de arbitru AHB este instantiat, iar test_harness-ul este conectat la interfețele designului. De asemenea, interfețele sunt înregistrate în config_db pentru simulare.

Procesul de compilare, elaborare și simulare în zona EDA (Electronic Design Automation) este esențial în dezvoltarea și verificarea circuitelor integrate (CI) și a sistemelor digitale. Acest proces implică utilizarea unor instrumente software specializate pentru a transforma descrierile hardware și software într-o formă executabilă pe care o putem testa și verifica.

1. **Compilare (Compilation):** Primul pas în EDA implică utilizarea unui compilator pentru a transforma codul sursă al design-ului (de exemplu, în limbajul SystemVerilog) într-un format intermediar care poate fi utilizat de alte instrumente din fluxul de proiectare. Acest pas verifică sintaxa și semantica codului și generează o formă internă a design-ului.
2. **Elaborare (Elaboration):** Pasul de elaborare are loc după compilare și implică construirea ierarhiei și a interconexiunilor design-ului. În acest pas, modulele și interfețele sunt conectate pentru a forma structura ierarhică completă a design-ului. De asemenea, sunt rezolvate referințele și instanțele modulelor pentru a asigura conectivitatea corectă.
3. **Simulare (Simulation):** După etapa de elaborare, design-ul este gata pentru a fi simulat utilizând un simulator logic. În acest pas, se aplică semnale de intrare asupra circuitului și se monitorizează semnalele de ieșire pentru a verifica funcționalitatea corectă. Simularea poate fi realizată în mai multe moduri, cum ar fi simularea funcțională, simularea temporală sau simularea în cicluri de instrucțiuni (transaction-level simulation).

```

src > tb > tb_top.sv > top
  1  module top(
  2  );
  3    // import the UVM library
  4    import uvm_pkg::*;
  5
  6    // include the UVM macros
  7    `include "uvm_macros.svh"
  8    import integration_pkg::*;
  9    import tests_pkg::*;

10
11
12  logic clk;
13  logic reset;
14  time moment_reset;
15
16  reset_if r_if (.hclk(clk));
17
18  assign reset = r_if.hreset;
19
20  test_harness th (.hclk(clk), .hreset(reset));
21
22
23
24  initial begin
25    |  uvm_config_db #(virtual reset_if)::set(null,"", "reset_if", r_if);
26  end
27
28
29
30  > generic_arbiter_full DUT( .m_busreq(th.m_hbusreq), ...
56  );
57
58
59  initial begin
60    clk <= 0;
61    forever #5ns  clk = ~clk;
62  end
63
64
65  initial begin
66    |  run_test();
67  end
68
69 endmodule
70

```

Fig 3.2.4 Fișierul top unde sunt făcute instantierile, conectările la DUT și pornește simularea

În `test_harness`, sunt realizate legăturile pentru interfețele master, slave și interfața de cerere a magistralei (bus request), prin care se transmit cererile și răspunsurile pentru magistrală. Aceste legături sunt statice și sunt formate în faza de elaborare (elaboration phase).

Odată ce aceste legături sunt înregistrate în `config_db` (configurația bazată pe baze de date), ele devin obiecte dinamice numite interfețe virtuale. Aceste interfețe virtuale pot fi utilizate pentru a scrie și a citi semnalele pe Dispozitivul Unit de Testare (DUT) în cadrul simulării. Ele facilitează comunicarea între `test_harness` și DUT, permitând trimiterea și recepționarea de date între aceste componente.

```

1  interface test_harness(input hclk, input hreset);
2
3  import integration_pkg::*;
4
5  import uvm_pkg::*;
6
7  //master signals
8
9
10 wire[32*master_number-1:0] m_hwdata;
11 wire[31:0] m_hrdata;
12 wire[32*master_number-1:0] m_haddr;
13 wire[3*master_number-1:0] m_hburst;
14 wire[2*master_number-1:0] m_htrans;
15 wire[master_number-1:0] m_hbusreq;
16 wire[master_number-1:0] m_hlock;
17 wire[1:0] hresp;
18 wire hready;
19 wire[master_number-1:0] hgrant;
20 //wire[2:0] m_hsize; //= 2;
21 wire [3*master_number-1:0] m_hsize;
22 wire [master_number-1:0] m_hwrite;
23
24 generate
25   for(genvar i=0;i<master_number;i++)
26     begin: m_if
27       master_if master(.*);
28       //outputs
29       assign m_if[i].master.hresp      =hresp;
30       assign m_if[i].master.hready    =hready;
31       assign m_if[i].master.hgrant   =hgrant[i];
32       assign m_if[i].master.hrdata=m_hrdata;
33
34       //inputs
35       assign m_hwrite[(i+1)-1:i]     =m_if[i].master.hwrite;
36       assign m_hwdata[32*(i+1)-1:32*i]=m_if[i].master.hwdata;
37       assign m_haddr[32*(i+1)-1:32*i]=m_if[i].master.haddr;
38       assign m_hburst[3*(i+1)-1:3*i]=m_if[i].master.hburst;
39       assign m_htrans[2*(i+1)-1:2*i]=m_if[i].master.htrans;
40       assign m_hbusreq[i]           =m_if[i].master.hbusreq;
41       assign m_hlock[i]             =m_if[i].master.hlock;
42       //assign m_hsize = m_if[i].master.hsize;
43       assign m_hsize[3*(i+1)-1:3*i]=m_if[i].master.hsize;
44
45
46     initial begin
47       uvm_config_db #(virtual master_if)::set(null,"", $formatf("master[%0d]", i), master);
48     end

```

Fig 3.2.5 Test harness

Numărul de interfețe pentru master și slave poate fi controlat prin utilizarea unui fișier de parametri. Acest fișier de parametri poate conține informații referitoare la configurarea sistemului și poate include variabile care controlează numărul de interfețe pentru master și slave.

În fișierul de parametri, se pot defini variabile cum ar fi:

- numărul de interfețe și agenți pentru master (master_number)
- numărul de interfețe și agenți pentru slave (slave_number)

```

1 package integration_pkg;
2 timeunit 1ns/1ns;
3
4 //import uvm_pkg::*;
5
6
7 parameter master_number=9;
8 parameter slave_number=7;
9 parameter size_out=4;
10 parameter size_out_s=3;
11 parameter number_trans = 4; //numarul de secvențe pentru un master
12 [31:0] slave_low_address[] = {'d0, 'd32, 'd64, 'd96, 'd128, 'd160, 'd192, 'd224, 'd256, 'd288, 'd320};
13 parameter [31:0] slave_high_address[] = {'d31, 'd63, 'd95, 'd126, 'd159, 'd191, 'd223, 'd255, 'd287, 'd319, 'd351};
14
15 typedef enum bit[1:0] {IDLE, BUSY, NONSEQ, SEQ} transfer_t;
16 typedef enum bit {READ, WRITE} rw_t;
17 typedef enum bit [2:0] {SINGLE, INCR, WRAP4, INCR4, WRAP8, INCR8, WRAP16, INCR16} burst_t;
18 typedef enum bit [2:0] {BYTE, HALFWORD, WORD, WORDx2, WORDx4, WORDx8, WORDx16, WORDx32} size_t;
19 typedef enum bit [1:0] {OKAY, ERROR, RETRY, SPLIT} resp_t;
20
21 parameter ARBITRATION = 0;
22 parameter ADDRESS = 1;
23 parameter DATA = 2;
24
25 parameter MAX_SLAVE = 10;
26
27 //parameter IDLE = 0;
28
29 endpackage: integration_pkg

```

Fig 2.3.6 Fișierul cu definiții și parametrii

Rolul sequencerului este de a coordona și controla executarea secvențelor de tranzacții pe interfață specifică pe care este conectat. Sequencerul primește comenziile de la secvența virtuală și le traduce în tranzacții specifice interfeței respective. El se ocupă de generarea și gestionarea tranzacțiilor, inclusiv transmiterea acestora la DUT și așteptarea răspunsurilor corespunzătoare.

Secvența virtuală, la rândul ei, este responsabilă pentru crearea și gestionarea secvențelor de tranzacții. Ea decide când și în ce ordine să fie pornite secvențele, având acces la informațiile despre starea și configurarea DUT-ului. Secvența virtuală poate controla, de asemenea, parametrii și variabilele specifice testului și poate executa decizii logice sau ramificări în funcție de rezultatele obținute.

Un sequencer virtual poate conține pointeri către alți sequenceri pentru a controla executarea secvențelor virtuale pe interfețe diferite sau pentru a împărti responsabilitățile de gestionare a tranzacțiilor între mai mulți sequenceri. Aceasta permite o mai mare flexibilitate și modularitate în gestionarea tranzacțiilor și permite orchestarea complexă a secvențelor dintr-un singur loc centralizat.

În ansamblu, sequencerul și secvența virtuală lucrează împreună pentru a crea și a controla secvențele de tranzacții necesare pentru testarea și verificarea sistemului. Ele asigură o execuție eficientă și coordonată a secvențelor, permitând dezvoltatorului să se concentreze pe logica testului și pe analiza rezultatelor obținute.

```

1 class ahb_vsequencer extends uvm_sequencer#(ahb_transaction);
2
3   `uvm_component_utils(ahb_vsequencer)
4
5   ahb_sequencer master_seqr[master_number];
6   slave_sequencer slave_seqr[slave_number];
7
8   function new(string name = "ahb_vsequencer" , uvm_component parent);
9     super.new(name,parent);
10    endfunction
11  endclass

```

Fig 3.2.7 Sequencer virtual

În testul de bază, base_test, se va realiza configurarea mediului (environment) și vor fi pornite secvențele necesare pentru a executa testul.

Configurarea mediului (environment) implică crearea și conectarea componentelor necesare pentru testare, cum ar fi agentul, sequencerul, scoreboard-ul și alte module relevante. Aceasta poate implica, de asemenea, setarea parametrilor și configurarea interfețelor pentru a asigura comunicarea corectă între componentele sistemului.

După configurarea mediului, vor fi pornite secvențele necesare pentru a executa testul. Aceste secvențe pot fi secvențe predefinite, precum secvența de inițializare sau secvența de tranzacții, sau pot fi secvențe specifice testului care au fost definite în prealabil. Secvențele pot fi pornite secvențial sau în paralel, în funcție de nevoile testului.

Pornirea secvențelor se poate face prin apelarea funcțiilor corespunzătoare în cadrul testului base_test. Aceste funcții vor comunica cu sequencerul și vor iniția execuția secvențelor.

```
1  class base_test extends uvm_test;
2    `uvm_component_utils(base_test)
3
4    ahb_env env; // my env
5    env_config env_cfg;
6
7    //base virtual sequence
8    virtual_base_sequence vseq_h;
9
10   //slave sequences
11   ahb_slave_base_seq slave_seq;
12
13   //virtual sequence base
14
15
16   //reset sequence
17   reset_seq reset_seq_h;
18
19   function new(string name="base_test", uvm_component parent = null);
20     super.new(name,parent);
21   endfunction
22
23
24   function void build_phase(uvm_phase phase);
25     super.build_phase(phase);
26
27     env_cfg = env_config::type_id::create("env_cfg", this);
28
29     env_cfg.enable_coverage = 1;
30     env_cfg.is_active = 1;
31
32     uvm_config_db#(env_config)::set(null, "", "env_config", env_cfg);
33
34     env = ahb_env::type_id::create("env",this);
35     `uvm_info(get_type_name(),"Build phase of test is executing",UVM_HIGH)
36
37
38
39
40  endfunction
```

Fig 3.2.8 Testul de bază

```

41     virtual task run_phase(uvm_phase phase);
42         vseq_h = virtual_base_sequence::type_id::create("vseq_h");
43
44
45
46
47         phase.raise_objection(this);
48         fork
49             begin
50                 for (int i=0; i<slave_number; i++) begin
51                     automatic int j=i;
52                     fork begin
53                         ahb_slave_base_seq slave_seq;
54                         slave_seq = slave_response_seq::type_id::create("slave_seq");
55
56                         slave_seq.start(env.s_agent[j].sequencer);
57                     end
58                     join_none
59                 end
60             end
61         end
62         begin
63             vseq_h.start(env.vsequencer);
64         end
65         begin
66             reset_seq_h = reset_seq::type_id::create("reset_seq");
67             reset_seq_h.start(env.reset_agent_h.reset_seqr_h);
68         end
69
70         join
71
72         phase.phase_done.set_drain_time(this, 50000ns);
73         phase.drop_objection(this);
74
75
76
77
78     endtask

```

Fig 3.2.9 Pornirea secvențelor virtuale

În cadrul fazei de execuție (run_phase()) a simulării UVM, secvențele specificate vor fi pornite. Pentru a asigura desfășurarea corectă a simulării și a evita oprirea prematură a acestia, este important să se utilizeze metodele raise_objection() și drop_objection(). Aceste metode permit simulării să consume timp și să continue desfășurarea, în loc să se oprească în mod neașteptat.

Secvențele vor apela raise_objection(), semnalând astfel că operațiile sunt în curs de desfășurare și simularea trebuie să continue. Apoi, acestea vor crea un obiect de tip tranzacție AHB și îl vor trimite către sequencerul agentului de master. Acest proces permite gestionarea și monitorizarea tranzacțiilor în cadrul mediului de verificare.

Slave-ul AHB a fost implementat folosind un slave reactiv, care are rolul de a monitoriza tranzacțiile plasate pe interfața de la master. Pentru a realiza acest lucru, se folosește un monitor care interceptează tranzacțiile și le transmite slave-ului informațiile relevante. Slave-ul reactiv trimite apoi cereri către sequencer pentru a genera răspunsurile corespunzătoare, care sunt apoi conduse către interfața AHB.

Prin intermediul acestei abordări, tranzacțiile sunt gestionate eficient și corect în cadrul mediului de verificare, iar simularea poate fi controlată într-un mod precis și coerent, asigurând astfel validarea și verificarea adecvată a designului integrat.

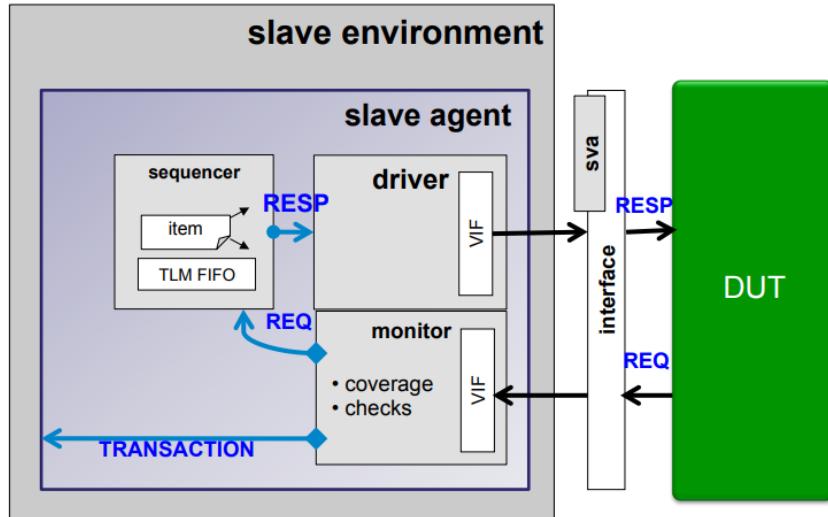


Fig 3.2.10 Slave reactiv [18]

Următoarele poze ilustrează clasele pentru agenți, care reprezintă componente cheie în cadrul sistemului. Configurarea agentului se realizează în aceste clase, iar modul în care un agent este definit și comportamentul său depind de natura sa: activ sau pasiv.

Dacă un agent este activ, acesta va conține un driver. Driver-ul are rolul de a genera și de a trimite tranzacții către interfața corespunzătoare. El gestionează interacțiunea cu mediul extern, respectând protocolul specificat.

În cazul în care un agent este pasiv, acesta va conține doar un monitor. Monitorul are rolul de a observa și de a captura tranzacțiile care trec prin interfața corespunzătoare. El monitorizează comunicarea dintre sistemul de test și designul verificat, capturând datele și informațiile relevante pentru analiză și verificare ulterioară.

Diferența între agenții activi și cei pasivi rezidă în responsabilitățile și acțiunile pe care le desfășoară. Agenții activi sunt implicați în generarea și transmiterea tranzacțiilor, în timp ce agenții pasivi se concentrează pe monitorizarea și capturarea acestora. Această abordare modulară și bine definită a agenților facilitează configurarea și controlul fluxului de date în cadrul sistemului verificat.

```

src > ahb_master_agent > ahb_master_agent.sv > ahb_master_agent > build_phase
  1  class ahb_master_agent extends uvm_agent;
  2
  3    `uvm_component_utils(ahb_master_agent)
  4
  5    ahb_master_driver ahb_mdriver;
  6    ahb_master_monitor ahb_mmonitor;
  7    ahb_request_monitor req_monitor;
  8    ahb_sequencer sequencer;
  9    ahb_magent_config config_h;
 10
 11
 12  function new(string name="ahb_master_agent",uvm_component parent = null);
 13    super.new(name,parent);
 14  endfunction //new()
 15
 16  virtual function void build_phase(uvm_phase phase);
 17    super.build_phase(phase);
 18
 19    if(!uvm_config_db#(ahb_magent_config)::get(null, this.get_name(), "ahb_magent_config", config_h))
 20      `uvm_fatal(get_full_name(), "Can't get env_config from db")
 21
 22    ahb_mmonitor = ahb_master_monitor::type_id::create("ahb_mmonitor",this);
 23    req_monitor = ahb_request_monitor::type_id::create("req_monitor",this);
 24
 25    if (config_h.is_active) begin
 26      sequencer = ahb_sequencer::type_id::create("sequencer",this);
 27      ahb_mdriver = ahb_master_driver::type_id::create("ahb_mdriver",this);
 28    end
 29
 30  endfunction
 31
 32
 33  virtual function void connect_phase(uvm_phase phase);
 34    super.connect_phase(phase);
 35
 36    if (config_h.is_active) begin
 37      ahb_mdriver.seq_item_port.connect(sequencer.seq_item_export);
 38    end
 39
 40  endfunction
 41
 42
 43
 44 endclass //ahb_master_agent extends uvm_agent

```

Fig 3.2.11 Agent de master

```

1  class ahb_slave_agent extends uvm_agent;
2
3    `uvm_component_utils(ahb_slave_agent)
4
5    ahb_slave_driver ahb_sdriver;
6    slave_sequencer sequencer;
7    ahb_slave_monitor ahb_smonitor;
8    ahb_sagent_config config_h;
9
10
11  function new(string name="ahb_slave_agent",uvm_component parent=null);
12    super.new(name,parent);
13  endfunction
14
15  virtual function void build_phase(uvm_phase phase);
16    super.build_phase(phase);
17    ahb_smonitor = ahb_slave_monitor::type_id::create("ahb_smonitor",this);
18    // config flags
19
20    if(!uvm_config_db#(ahb_sagent_config)::get(null, this.get_name(), "ahb_sagent_config", config_h))
21      `uvm_fatal(get_full_name(), "Can't get env_config from db")
22
23    if (config_h.is_active) begin
24      sequencer = slave_sequencer::type_id::create("sequencer",this);
25      ahb_sdriver = ahb_slave_driver::type_id::create("ahb_sdriver",this);
26    end
27
28
29
30  endfunction
31
32  virtual function void connect_phase(uvm_phase phase);
33    if (config_h.is_active) begin
34      //connect driver to sequencer port
35      ahb_sdriver.seq_item_port.connect(sequencer.seq_item_export);
36      //connect monitor port to export port
37      ahb_smonitor.reactive_transaction_port.connect(sequencer.m_request_export);
38    end
39
40
41  endfunction
42
43
44 endclass //ahb_slave_agent extends uvm_agent

```

Fig 3.2.12 Agent de slave

Pentru a beneficia de avantajele programării orientate pe obiecte, majoritatea obiectelor utilizate în verificare sunt moștenite din clasele de bază UVM. Această abordare ne permite să folosim funcționalitățile și metodele definite în clasele de bază și să le extindem pentru a se potrivi nevoilor noastre specifice.

În ceea ce privește sequence_item-urile și teste, dorim să putem modifica comportamentul ierarhiei noastre de obiecte de verificare. În funcție de tipul de transfer pe care dorim să-l efectuăm, vom avea un test de bază din care vom moșteni și, utilizând UVM factory override, vom schimba tipul obiectului.

Acest mecanism ne permite să personalizăm și să adaptăm comportamentul obiectelor noastre de verificare în funcție de cerințele și scenariile de testare. Astfel, putem obține o flexibilitate sporită în gestionarea diferitelor tipuri de transferuri și să ne asigurăm că testele noastre acoperă toate cazurile importante.

Această abordare modulară și flexibilă oferită de programarea orientată pe obiecte și de framework-ul UVM ne permite să dezvoltăm și să rulăm teste de verificare complexe și eficiente pentru a valida corectitudinea designului nostru de circuit integrat.

```

1  class incr_read_4_test extends base_test;
2    `uvm_component_utils(incr_read_4_test)
3
4    function new(string name="incr_read_4_test", uvm_component parent);
5      super.new(name,parent);
6    endfunction
7
8    function void build_phase(uvm_phase phase);
9      virtual_base_sequence::type_id::set_type_override(virtual_incr_read_4sequence::get_type());
10     super.build_phase(phase);
11   endfunction
12
13   task run_phase(uvm_phase phase);
14     super.run_phase(phase);
15   endtask
16 endclass
17

```

Fig 3.2.13 Exemplu de test extins din base_test si factory_override

Această abordare sprijină reutilizarea codului și îmbunătățește mentenabilitatea, deoarece putem crea o secvență și un test noi într-un fișier separat de fiecare dată când dorim să rulăm un test sau un scenariu nou într-un mediu specific. Astfel, putem gestiona cu ușurință diferitele cazuri de testare și putem reutiliza componente de verificare deja existente.

Pentru a implementa această funcționalitate, este necesar să modificăm fișierul "run.f". În acest fișier, trebuie să adăugăm "+UVM_TEST" la configurația existentă, indicând astfel că dorim să rulăm un test specificat într-un fișier separat. Acest test va include secvența corespunzătoare și alte componente necesare pentru a efectua testul dorit.

Prin adăugarea acestei configurații și utilizarea unui fișier separat pentru fiecare test, putem gestiona și rula cu ușurință diferite combinații de secvențe și teste, adaptându-ne nevoilor noastre specifice de verificare. Aceasta facilitează dezvoltarea, extinderea și întreținerea mediului de verificare într-un mod eficient și modular.

Un sequence_item este definit ca o clasă în cadrul mediului UVM (Universal Verification Methodology) și conține variabile și metode care descriu și controlează informațiile legate de tranzacție. Acesta poate include câmpuri precum adrese, date, direcția transferului, lățimea datelor și alte informații specifice protocolului.

În ahb_transaction este modelată cererea și răspunsul protocolului AHB.

```

class ahb_transaction extends uvm_sequence_item;
  `uvm_object_utils(ahb_transaction)

  //simulation time it was sampled at
  int sampled_at;
  //signifies from which transfer the beat is part of
  int tag;
  //id of the corresponding agent
  int id;
  //haddr, control and data
  rand logic [31:0] haddr[];
  rand size_t hsize;
  rand burst_t hburst;
  rand rw_t hwrite; // read/write
  rand transfer_t htrans[];
  rand logic [31:0] hdata[];

  //busy transfer
  rand int no_of_busy;
  rand int busy_pos;

  //bus req signals
  rand logic hlock;
  rand logic hbusreq;
  rand int lock_duration; // how many cycles lock will be 1.

  //slave response signals
  rand bit hready;
  rand resp_t hresp;
  rand bit [31:0] hdata;

  rand bit no_of_waits[];

  //slave select
  logic hsel;

  //****Add other signals for sampling****

  function new(string name = "ahb_transaction");
    super.new(name);
  endfunction

```

Fig 3.2.14 Modelarea unui transfer AHB (ahb_transaction)

UVM ne oferă, de asemenea, o serie de funcții utile pe care le putem rescrie în clasele noastre derivate, cum ar fi do_copy(), do_print(), do_compare() și convert2string(). Aceste funcții sunt implementate în clasa de bază UVM_OBJECT și ne permit să personalizăm comportamentul obiectelor noastre în cadrul metodologiei.

Funcția do_copy() este folosită pentru a crea copii independente ale obiectelor, iar funcția do_print() ne permite să afișăm informații specifice obiectului. Putem redefini logica funcției do_compare() pentru a compara două obiecte și putem utiliza funcția convert2string() pentru a converti obiectul într-un sir de caractere.

Aceste funcții utile ne ajută în procesul de verificare, oferindu-ne flexibilitatea de a personaliza comportamentul obiectelor în funcție de cerințele noastre specifice. Prin rescrierea acestor funcții, putem gestiona și manipula obiectele UVM într-un mod mai eficient și adaptat nevoilor noastre de verificare.

```

virtual function string convert2string();
    string s = super.convert2string();
    $sformat (s, "%s\n    ahb_transaction with id = %0d : ", s,id);
    $sformat (s, "%s\n      tag = %0d", s, tag);
    $sformat (s, "%s\n      hbusreq = %0d", s, hbusreq);
    $sformat (s, "%s\n      hlock = %0d", s, hlock);
    $sformat (s, "%s\n      lock_duration = %0d", s, lock_duration);
    $sformat (s, "%s\n      haddr = %p", s, haddr);
    $sformat (s, "%s\n      hwdata = %p", s, hwdata);
    $sformat (s, "%s\n      hburst = %0d", s, hburst);
    $sformat (s, "%s\n      htrans = %p", s, htrans);
    $sformat (s, "%s\n      hsize = %0d", s, hsize);
    $sformat (s, "%s\n      hready = %0d", s, hready);
    $sformat (s, "%s\n      hsel = %0d", s, hsel);
    $sformat (s, "%s\n      hresp = %0d", s, hresp);
    $sformat (s, "%s\n      hrdata = %h", s, hrdata);
    $sformat (s, "%s\n      no_of_waits = %p", s, no_of_waits);
    $sformat (s, "%s\n      busy_pos = %0d", s, busy_pos);
    $sformat (s, "%s\n      no_of_busy = %0d", s, no_of_busy);

    return s;
endfunction

```

Fig 3.2.15 convert2string()

```

virtual function bit do_compare(uvm_object rhs, uvm_comparer comparer);
    bit res;
    ahb_transaction tx_rhs;
    if(!$cast(tx_rhs,rhs))
        `uvm_fatal(get_type_name(),"Illegal rhs argument")

    res = super.do_compare(rhs,comparer) &&
        //the master that initiates the transaction
        //data that was sent by master vs received by slave
        (hburst === tx_rhs.hburst) &&
        (hsize === tx_rhs.hsize) &&
        (hwrite === tx_rhs.hwrite) &&
        //the slave response vs what master received
        (hready === tx_rhs.hready) &&
        (hresp === tx_rhs.hresp) &&
        //((hsel === tx_rhs.hsel) &&
        //  (hrdata === tx_rhs.hrdata));

    if (haddr.size() == tx_rhs.haddr.size()) begin
        for (int i = 0; i < haddr.size(); i++) begin
            if (haddr[i] != tx_rhs.haddr[i]) begin
                res = 0;
                break;
            end
        end
    end else begin
        res = 0;
    end
    if (htrans.size() == tx_rhs.htrans.size()) begin
        for (int i = 0; i < htrans.size(); i++) begin
            if (htrans[i] != tx_rhs.htrans[i]) begin
                res = 0;
                break;
            end
        end
    end else begin
        res = 0;
    end
    if (hwdata.size() == tx_rhs.hwdata.size()) begin
        for (int i = 0; i < hwdata.size(); i++) begin
            if (hwdata[i] != tx_rhs.hwdata[i]) begin
                res = 0;
                break;
            end
        end
    end else begin
        res = 0;
    end
    return res;
endfunction

```

Fig 3.2.16 do_compare()

SystemVerilog folosește generatoare de numere aleatorii pentru a ajuta la popularea câmpurilor obiectelor atunci când acestea sunt create. Pentru a ne asigura că aceste valori sunt în limitele admise, putem impune constrângeri asupra lor în cadrul clasei sequence_item sau putem utiliza constrângeri inline în interiorul codului pentru sevență. Declarația câmpurilor cu atributul "rand" indică faptul că acestea vor fi randomizate.

Este recomandat să declarăm ca "rand" doar intrările (input-urile) obiectului, deoarece acestea sunt valorile pe care le vom controla și pe care le vom genera în cadrul sevențelor. Atributul "rand" ne permite să beneficiem de generarea automată a valorilor aleatorii pentru aceste intrări, în conformitate cu constrângerile specificate.

Prin utilizarea generatoarelor de numere aleatorii și a constrângerilor, putem crea scenarii de testare mai variate și mai robuste, care să acopere un spectru mai larg de combinații și situații posibile. Aceasta ne ajută să identificăm și să rezolvăm eventuale erori sau probleme în timpul procesului de verificare al designului.

```

constraint haddr_size {
    //header Based on burst and hsize
    if(hburst == SINGLE)
        haddr.size == 1;
    if(hburst == INCR)
        haddr.size < 256; // problem using the above formula, number is for hsize == WORD
    if(hburst == WRAP || hburst == INCR4)
        haddr.size == 4;
    if(hburst == WRAP2 || hburst == INCR8)
        haddr.size == 8;
    if(hburst == WRAP16 || hburst == INCR16)
        haddr.size == 16;
}

constraint addr_wrap4_byte{
    if(hburst == WRAP4 || hsize == BYTE){
        foreach(haddr[i]){
            if(i != 0){
                haddr[i][1:0] == haddr[i-1][1:0] + 1;
                haddr[i][31:2] == haddr[i-1][31:2];
            }
        }
    }
}

constraint addr_wrap8_byte{
    if(hburst == WRAP8 || hsize == BYTE){
        foreach(haddr[i]){
            if(i != 0){
                haddr[i][2:0] == haddr[i-1][2:0] + 1;
                haddr[i][31:3] == haddr[i-1][31:3];
            }
        }
    }
}

constraint addr_wrap16_byte{
    if(hburst == WRAP16 || hsize == BYTE){
        foreach(haddr[i]){
            if(i != 0){
                haddr[i][3:0] == haddr[i-1][3:0] + 1;
                haddr[i][31:4] == haddr[i-1][31:4];
            }
        }
    }
}

constraint addr_wrap_halfword{
    if(hburst == WRAP4 || hsize == HALFWORD){
        foreach(haddr[i]){
            if(i != 0){
                haddr[i][2:1] == haddr[i-1][2:1] + 1;
                haddr[i][31:5] == haddr[i-1][31:5];
            }
        }
    }
}

constraint request1{
    // block == 0;
    hbusreq == 1;
    //hbusreq dist {0:/2,1:/1};
    if(hbusreq) {
        hlock.dist {0:/2,1:/1};
    }
}

constraint locked_cycles{
    if(hlock) {
        lock.duration < haddr.size;
        lock.duration > 0;
    }
}

//comment this constraint for no busy and post_randomize also.
constraint busy_pos_and_size{
    if(hburst != SINGLE){
        {
            busy_pos > 0;
            busy_pos < haddr.size - 1;
            no_of_busy >= 0;
            no_of_busy < 6;
        } else {
            busy_pos == 0;
            no_of_busy == 0;
        }
    }
}

constraint wait_size{
    no_of_waits.size >= 0;
    no_of_waits.size < 17;
    //octetos 0 and 16
}

constraint number_of_waits_values{
    foreach (no_of_waits[i])
        if (i == no_of_waits.size - 1) {
            no_of_waits[i]==1;
        } else {
            no_of_waits[i]==0;
        }
}

```

Fig 3.2.17 Constrângeri

Pentru a realiza modelarea temporală a protocolului, vom utiliza un driver care transformă obiectele de tip sequence_item în semnale pe interfața virtuală a masterului, care este conectată la DUT (Device Under Test). Driverul este parametrizat pentru a se adapta cerințelor și specificațiilor protocolului respectiv.

Protocolul implementat este de tip pipelined, ceea ce înseamnă că faza de date a unui transfer curent se va desfășura în timpul fazei de adresă a următorului transfer. Pentru a realiza acest lucru, am utilizat thread-uri și am creat un thread dedicat pentru faza de date și unul pentru faza de adresă. În fiecare fază, se citesc sau se scriu datele din obiectul ahb_transfer (sequence_item), iar prin intermediul unui mailbox, obiectul este transmis de la thread-ul pentru faza de adresă la cel pentru faza de date.

Această abordare ne permite să simulăm în mod corespunzător interacțiunea între master și slave în cadrul protocolului pipelined. Fiecare thread se ocupă de etapa corespunzătoare a transferului, asigurând sincronizarea corectă între cele două faze și realizând transmiterea datelor între ele. Astfel, putem obține o reprezentare fidelă a comportamentului protocolului în cadrul mediului de simulare.

```

class ahb_master_driver extends uvm_driver#(ahb_transaction);
  `uvm_component_utils(ahb_master_driver)
  virtual master_if vif;
  ahb_magent_config agent_config;
  mailbox mbx = new();
  int haddr_index=0;
  int was_busy = 0;
  function new(string name = "ahb_master_driver", uvm_component parent);
    super.new(name, parent);
  endfunction: new

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    if(!uvm_config_db #(ahb_magent_config)::get(null,get_parent().get_name(), "ahb_magent_config", agent_config))
      `uvm_fatal(get_type_name(), "Failed to get config inside Master Driver")
    if(!uvm_config_db #(virtual master_if)::get(this, "", $formatf("master[%0d]", agent_config.agent_id), vif))
      `uvm_fatal(get_type_name(), "Failed to get VIF inside Master Driver")
  endfunction

  task initialize();
    vif.m_cb.hbusreq <= 0;
    vif.m_cb.hlock  <= 0;
    vif.m_cb.haddr  <= 0;
    vif.m_cb.hwdata <= 0;
    vif.m_cb.hburst <= 0;
    vif.m_cb.htrans <= 0;
    vif.m_cb.hsize  <= 0;
    vif.m_cb.hwrite <= 0;
    repeat (1) begin
      @ (vif.m_cb);
    end
  endtask

  virtual task run_phase(uvm_phase phase);
    initialize();
    repeat(2) @vif.m_cb;
    forever begin
      initialize();
    end
  endtask

```

Fig 3.2.18 Driver de master ce modeleaza protocolul AHB

```

virtual task run_phase(uvm_phase phase);
  initialize();
  repeat(2) @vif.m_cb;
  forever begin
    initialize();
    wait(vif.hreset==1);
    fork
      address_phase();
      data_phase();
      reset_monitor();
    join any
    disable fork;
  end
endtask

task reset_monitor();
  wait(vif.hreset==0);
  if (req!=null) begin
    seq_item_port.put(req);
  end
endtask

task address_phase();
  forever begin
    seq_item_port.get(req);
    //dont drive when reset is 0
    while (!vif.hreset) @vif.m_cb;
    req.id = agent_config.agent_id;
    `uvm_info(get_type_name(), $formatf("Master Driver item : \n %s",req.convert2string()),UVM_MEDIUM);
    haddr_index = 0 ;
    vif.m_cb.hbusreq <= req.hbusreq;
    vif.m_cb.hlock <= req.hlock;
    for (int i=0; i<req.htrans.size(); ++i) begin
      //wait for bus to be granted
      while (!(vif.m_cb.hgrant && vif.m_cb.hready && vif.hreset)) begin
        vif.m_cb.htrans <= IDLE;
      end
    end
  end
endtask

```

Fig 3.2.19 Driver de master ce modeleaza protocolul AHB

```

if(req.htrans[i] == NONSEQ || req.htrans[i] == SEQ) begin
    vif.m_cb.haddr <= req.haddr[haddr_index];
    haddr_index++;
    /* construct to make a variable hlock */
    if (req.lock_duration > 0) begin
        req.lock_duration--;
    end
    vif.m_cb.htrans <= req.htrans[i];
    vif.m_cb.hwrite <= req.hwrite;
    vif.m_cb.hsize <= req.hsize;
    vif.m_cb.hburst <= req.hburst;
end else if(req.htrans[i] == BUSY ) begin
    while (req.no_of_busy>0) begin
        vif.m_cb.htrans <= req.htrans[i];
        vif.m_cb.haddr <= req.haddr[haddr_index];
        vif.m_cb.hwrite <= req.hwrite;
        vif.m_cb.hsize <= req.hsize;
        vif.m_cb.hburst <= req.hburst;
        req.no_of_busy--;
        i++;
        @vif.m_cb;
    end
    i--;
end

if(!was_busy) begin
    @(vif.m_cb iff(vif.m_cb.hready && vif.hreset));
    mbx.put(req);
end

//waitwa se executa in timp 0 daca expresia este true
if (haddr_index == req.haddr.size()-1 ) begin
    vif.m_cb.hbusreq <= 0;
    vif.m_cb.hlock <= 0;
end else if (haddr_index < req.haddr.size()-1) begin
    //request bus
    | vif.m_cb.hbusreq <= req.hbusreq;
    if (req.lock_duration == 0) begin
        vif.m_cb.hlock <= 0;
    end else begin
        vif.m_cb.hlock <= req.hlock;
    end
end
haddr_index = 0 ;
vif.m_cb.htrans <= IDLE;

end
endtask

```

Fig 3.2.20 Driver de master ce modeleaza protocolul AHB

Utilizând instrucțiunea "mbx.put(req)", se va debloca thread-ul "data_phase()" și va începe execuția fazei de date a protocolului AHB. Această instrucțiune plasează obiectul "req" în mailbox-ul asociat thread-ului "data_phase()", permitând transferul de date între threadurile pentru faza de adresă și faza de date.

Odată ce obiectul este plasat în mailbox, thread-ul "data_phase()" poate să-l preia și să-l utilizeze pentru a efectua operațiunile specifice fazei de date, cum ar fi citirea sau scrierea datelor pe interfața virtuală a masterului. Astfel, se asigură o sincronizare corectă între cele două faze ale transferului AHB, iar simularea protocolului poate avansa în mod adecvat.

Această abordare asigură o execuție secvențială a fazei de adresă și a fazei de date, reflectând comportamentul protocolului AHB și permitând verificarea corectitudinii și performanței acestuia în cadrul mediului de simulare.

```

uvm_master_agent > smd_master_driver.sv > smd_master_driver > UsecMonitor
repeat(2) @if(m_cb);
  forever begin
    initialize();
    wait(vif.hreset==1);
    fork
      address_phase();
      data_phase();
      reset_monitor();
    join_any
    disable fork;
  end
endtask

task reset_monitor();
  | wait(vif.hreset==0);
  | if (req!=null) begin
  |   seq_item_port.put(req);
  | end
endtask

task address_phase();-
endtask

task data_phase();
  ahb_transaction item;
  int i = 0;
  forever begin
    //drive data items
    mbx.get(item);
    if(item.hwrite == WRITE) begin
      | vif.m_cb.hdata <= item.hdata[i];
    end
    i++;
    if(item.haddr.size() == 1) begin
      seq_item_port.put(item);
      i=0;
    end
    `uvm_info(get_type_name(), $sformatf("Driver put item : \n"),UVM_MEDIUM);
  end
endtask

endclass //ahb_master_driver extends superClass

```

Fig 3.2.21 Fazele protocolului AHB

La finalul fazei de date, instrucțiunea "seq_item_port.put(item)" va trimite tranzacția către Scoreboard-ul de tranzacții, unde se vor efectua verificări pentru integritatea datelor și decodarea adresei. Scoreboard-ul de tranzacții are rolul de a compara tranzacțiile așteptate (expected transactions) cu tranzacțiile reale (actual transactions), asigurându-se astfel că transferul de date a fost realizat corect conform specificației protocolului AHB.

În cadrul interfețelor virtuale, am inclus checkerle pentru protocol. Acestea verifică forma protocolului, asigurându-se că modul în care se realizează comunicarea cu DUT respectă aspectele temporale și cerințele specificației AHB. Checkerle sunt responsabile de monitorizarea semnalelor și comportamentului protocolului, identificând eventualele abateri și generând alerte în cazul în care apar discrepanțe sau erori în execuția protocolului.

Prin includerea checkerelor în interfețele virtuale, se poate realiza o verificare continuă și automată a integrității protocolului, asigurându-se astfel că comunicarea între master și slave respectă cerințele și restricțiile impuse de specificație. Această verificare în timp real contribuie la identificarea și remedierea rapidă a eventualelor probleme sau discrepanțe, facilitând procesul de validare și testare al designului.

```

interface master_if(input hclk, input hreset);
    import integration_pkg::*;

    logic[31:0] haddr;
    logic[1:0] htrans;
    logic hwrite;
    logic[2:0] hsize;
    logic[2:0] hburst;
    logic[31:0] hdata;
    logic[31:0] hdata;
    logic hready;
    logic[1:0] hresp;
    logic hbusreq;
    logic hlock;
    logic hgrant;

    clocking m.cb @(posedge hclk);
        //default input #1step output `Tdrive;
        input hgrant,hready,hresp,hdata;
        inout htrans,haddr,hsize,hburst,hdata,hbusreq,hlock,hwrite;
    endclocking

    /*-----TRANSFER PROPERTIES*/

    //1KB Boundary Check Incrementing burst
    property kb_boundary_p;
        @(posedge hclk) disable iff(!hreset)
            (htrans == SEQ) |-> (haddr[10:0] != 11'b1_0000_0000);
    endproperty

    //Address Check for INCR/INCRx transfer
    property incr_addr_p;
        @(posedge hclk) disable iff(!hreset)
            (htrans == SEQ) && ((hburst == INCR)|| (hburst == INCR8)|| (hburst == INCR16)) &&
            ($past(htrans, 1) != BUSY) && ($past(hready, 1)) |-> (haddr == ($past(haddr, 1) + 2**hsize));
    endproperty

    //Address Check for WRAP4 hsize = WORD
    property wrap4_word_addr_p;
        @(posedge hclk) disable iff(!hreset)
            (htrans == SEQ) && (hburst == WRAP4) && (hsize == WORD) && ($past(htrans, 1) != BUSY) && ($past(hready, 1)) |->
            ((haddr[3:2] == ($past(haddr[3:2], 1) + 1)) && (haddr[31:4] == $past(haddr[31:4], 1)));
    endproperty

```

Fig 3.2.22 Interfață master și checkere de protocol

```

property burst_htrans_value_p;
    @(posedge hclk) disable iff(!hreset)
        hburst != SINGLE |=> htrans != NONSEQ ;
endproperty

ONE_KB: assert property(kb_boundary_p);
INCR_ADDR: assert property(incr_addr_p);
WRAP4_WORD_ADDR : assert property (wrap4_word_addr_p);
WRAP8_WORD_ADDR : assert property (wrap8_word_addr_p);
WRAP16_WORD_ADDR : assert property (wrap16_word_addr_p);
ADDR_ALIGNMENT_WORD : assert property(addr_alignment_word_p);
ADDR_ALIGNMENT_HALFWORD : assert property(addr_alignment_halfword_p);
SINGLE_NO_BUSY: assert property(no_busy_single_burst_p);
SAME_CTRL_SIG : assert property(ctrl_sig_same_p);
WAITED_SAME_TRANSFER_TYPE: assert property(same_transfer_tye_p);
NO_BUSY_AFTER_SINGLE : assert property(no_busy_after_single_p);

//FIRST_HTRANS_VALUE : assert property(first_htrans_value_p);
//ADDR_PHASE_DURATION : assert property(adr_phase_duration_p);

endinterface : master_if

```

Fig 3.2.22 Checkere de protocol din planul de verificare

Aceste aserțiiuni (assertions) rulează pe întreg parcursul simulării și sunt activate folosind construcția assert property(). Prin utilizarea unui label (etichetă), putem urmări cu ușurință locul unde acestea generează erori în timpul depanării codului.

Procesul de verificare funcțională implică identificarea funcționalităților pe care dorim să le testăm și evaluarea nivelului de acoperire (coverage) atins în cadrul acestor teste. Pentru a realiza acest lucru, folosim o componentă numită "coverage". În cadrul Scoreboard-ului, după compararea tranzacțiilor utilizând funcția do_compare(), tranzacțiile care sunt egale sunt trimise către componența de colectare a acoperirii (coverage).

Utilizarea acoperirii ne permite să evaluăm cât de bine au fost testate diferitele funcționalități ale designului. Cu ajutorul acoperirii, putem determina procentul de acoperire a diverselor cazuri și funcționalități, identificând zonele care nu au fost testate suficient și care necesită o atenție mai mare în procesul de testare. Aceasta facilitează monitorizarea și îmbunătățirea gradului de acoperire a testelor noastre, asigurându-ne că designul este supus unui set exhaustiv de scenarii și că funcționalitățile cheie sunt testate în mod corespunzător.

```

class ahb_scoreboard extends uvm_scoreboard;
`uvm_component_utils(ahb_scoreboard)
`uvm_analysis_imp_decl(_expected)
`uvm_analysis_imp_decl(_collected)

uvm_analysis_imp_expected #(ahb_transaction,ahb_scoreboard) item_expected;
uvm_analysis_imp_collected #(ahb_transaction,ahb_scoreboard) item_collected;

uvm_analysis_port #(ahb_transaction) coverage_port;
bit enable_coverage;

ahb_transaction expected_transactions[slave_number][$];
ahb_transaction coverage_queue[$];
ahb_transaction temp_cov;

ahb_transaction expected_tx;
ahb_transaction temp_tx;
ahb_transaction temp_tx1;
int match, mismatch;
int expected_transactions_counter;
int collected_transactions_counter;

int current_tag;
int previous_tag;
int flag_mismatch= 0 ;

CircularBuffer circular_buffer;

function new(string name = "ahb_scoreboard", uvm_component parent);
super.new(name, parent);
item_expected = new("item_expected",this);
item_collected = new("item_collected",this);
match = 0;
mismatch = 0;
expected_transactions_counter = 0;
collected_transactions_counter = 0;
circular_buffer = new;
endfunction

function void build_phase(uvm_phase phase);
super.build_phase(phase);
if (enable_coverage) begin
| coverage_port = new("coverage_port",this);
end
endfunction

function void write_expected(ahb_transaction master_item);
int i;

if (master_item.htrans[0] == NONSEQ) begin
| | | | expected_transactions_counter++;
end

for (i=0; i<slave_number; ++i) begin
if((master_item.haddr[0] >= slave_low_address[i]) && (master_item.haddr[0] <= slave_high_address[i] )) begin
| | | | expected_transactions[i].push_back(master_item); // i represents the slave number.
| |
end
endfunction

```

Fig 3.2.23 Scoreboard pentru transfer de date

```

for (i=0; i<slave_number; ++i) begin
    if((master_item.haddr[0] >= slave_low_address[i]) && (master_item.haddr[0] <= slave_high_address[i] )) begin
        expected_transactions[i].push_back(master_item); // i represents the slave number.
        `uvm_info(get_type_name(), $sformatf("Received from master[%0d] : \n %s", master_item.id,master_item.convert2string()), UVM_MEDIUM);
        break;
    end
end
if (i>=slave_number) begin
    `uvm_info(get_type_name(),$sformatf("Unmeped transaction not to be matched, haddr: %h",master_item.haddr[0]),UVM_MEDIUM);
end
endfunction

function void write_collected(ahb_transaction slave_item);
    `uvm_info(get_type_name(), $sformatf("Received from slave : \n %s",slave_item.convert2string()), UVM_DEBUG);

    //record when the collected came to SCB
    slave_item.sampled_at = $time;
    circular_buffer.add_transaction(slave_item);

    if (expected_transactions[slave_item.id].size == 0) begin
        `uvm_error(get_type_name(),"Queue is empty")
    end else begin
        temp_tx1 = expected_transactions[slave_item.id].pop_front();
        previous_tag = current_tag;
        current_tag = temp_tx1.tag;
        if ($slave_item.compare(temp_tx1)) begin
            coverage_queue.push_back(temp_tx1);
        end
        else begin
            flag_mismatch = 1;
            `uvm_error(get_type_name(),"MISMATCH : ");
            `uvm_error(get_type_name(), $sformatf("Expected : \n %s",temp_tx1.convert2string()));
            `uvm_error(get_type_name()," Received : \n %s",slave_item.convert2string());
        end
        //this means that a new transaction came and I can send to coverage.
        if (current_tag!=previous_tag) begin
            collected_transactions_counter++;
            if (flag_mismatch) begin
                //flush queue and increment mismatch;
                coverage_queue.delete();
                mismatch++;
                `uvm_info(get_type_name(), $sformatf("MISMATCH for tag : \n %d",previous_tag),UVM_MEDIUM);
            end else begin
                //send the coverage queue to coverage and increment match.
                match++;
                `uvm_info(get_type_name(), $sformatf("MATCH for tag : \n %d",previous_tag),UVM_MEDIUM);
                if (enable_coverage) begin
                    while (coverage_queue.size > 0) begin
                        temp_cov = coverage_queue.pop_front();
                        coverage_port.write(temp_cov);
                    end
                end
            end
        end
    end
end

```

Fig 3.2.23 Scoreboard pentru transfer de date, include checkerul pentru decodarea adresei

Prin utilizarea instrucțiunii coverage_port.write(temp_cov), transmitem obiectul de tranzacție către obiectul de acoperire (coverage). Aceasta permite înregistrarea acoperirii tranzacțiilor în cadrul obiectului de acoperire, astfel încât să putem evalua nivelul de acoperire obținut în timpul simulării.

La finalul simulării, avem faza check_phase() în care putem efectua diverse acțiuni, cum ar fi afișarea rezultatelor sau compararea obiectelor din cozi. Această fază este un moment potrivit pentru analizarea rezultatelor obținute în cadrul simulării și pentru a verifica dacă comportamentul obiectelor și rezultatele obținute corespund așteptărilor noastre. De exemplu, putem afișa mesaje de rezultate sau putem compara obiectele de rezultat cu obiectele așteptate pentru a verifica corectitudinea implementării noastre.

În general, faza check_phase() este utilizată pentru a verifica și valida rezultatele obținute în cadrul simulării și pentru a asigura că designul și testele funcționează conform așteptărilor noastre.

```

virtual function void check_phase(uvm_phase phase);
    foreach (expected_transactions[id]) begin
        while (expected_transactions[id].size > 0) begin
            temp_tx = expected_transactions[id].pop_front();
            `uvm_info(get_type_name(), $sformatf("Scoreboard received an unmatched : %s",temp_tx.convert2string()), UVM_MEDIUM);
        end
    end

    if (expected_transactions_counter!=collected_transactions_counter) begin
        `uvm_error(get_type_name(),$sformatf(" Number of master/slave transactions mismatch; nr of master_tx = [%0d] , nr of slave_tx = [%0d] ",expected_transactions_counter, collected_transactions_counter)
    end else begin
        `uvm_info(get_type_name(),$sformatf("Scb recived %0d transactions .",expected_transactions_counter),UVM_MEDIUM);
    end

    `uvm_info(get_type_name(),$sformatf("Matches: %0d ",match),UVM_MEDIUM);
    `uvm_info(get_type_name(),$sformatf("Mismatches: %0d ",mismatch),UVM_MEDIUM);

endfunction

virtual function void report_phase(uvm_phase phase);
    `uvm_info(get_type_name(),"Inside report_phase, flushing the collected transaction buffer",UVM_MEDIUM);

    for (int i = 0; i < circular_buffer.get_size(); i++) begin
        ahb_transaction transaction = circular_buffer.get_transaction(i);
        `uvm_info(get_type_name(),$sformatf("collected transaction %0d at time %0d .",transaction, transaction.sampled_at),UVM_MEDIUM);
    end

endfunction

endclass

```

Fig 3.2.24 check_phase()

În interiorul clasei de acoperire (coverage), avem definiții binii pentru diferite seturi de valori și combinații ale acestora. Fiecare combinație și punct de acoperire (coverpoint) reprezintă un scenariu al valorilor pe care dorim să știm că le-am transmis pe interfață. Fiecare combinație de valori va influența starea DUT-ului (Design Under Test).

Prin definirea binilor în cadrul clasei de acoperire, putem monitoriza și înregistra numărul de evenimente care se încadrează în fiecare bin. Aceasta ne permite să evaluăm acoperirea fiecărui scenariu și să determinăm dacă testele noastre sunt suficient de cuprinzătoare și au explorat toate combinațiile importante de valori.

Prin urmărirea acoperirii în cadrul clasei de acoperire, putem obține informații valoroase despre gradul de testare și exhaustivitatea testelor noastre. Acest lucru ne ajută să identificăm zonele care necesită o mai mare atenție și să dezvoltăm strategii de testare mai eficiente pentru a asigura o acoperire mai completă a scenariilor și stărilor DUT-ului nostru.

```

class ahb_coverage extends uvm_subscriber#(ahb_transaction);
  `uvm_component_utils(ahb_coverage)

  /****** COVERAGE FOR DATA TRANSFER ACCORDING TO AHB *****/
  ahb_transaction tx;

  covergroup ahb_master_cg ;
    option.per_instance = 1;
    type option.merge_instances = 1;
    read_write: coverpoint tx.hwrite {
      bins write_bin = {WRITE};
      bins read_bin = {READ};
    }
    htrans: coverpoint tx.htrans[0]{
      bins nonseq = {NONSEQ};
      bins seq = {SEQ};
      ignore_bins ignore_vals = {IDLE,BUSY};
    }
    haddr : coverpoint tx.haddr[0]{
      bins range_0 = {'[d0:d69]};
      bins range_1 = {'[d70:d140]};
      bins range_2 = {'[d141:d210]};
      bins range_3 = {'[d211:d281]};
      bins range_4 = {'[d282:d350]};
    }
    hburst : coverpoint tx.hburst {
      option.at_least = 1;
      /*A minimum number of hits for each bin. A bin with a hit count that is less than the number is not considered covered. the default value is '1'
      bins increment = {INCR, INCR4, INCR8, INCR16};
      bins wrap = {WRAP4, WRAP8, WRAP16};
      bins single = {SINGLE};
    }
    hsize: coverpoint tx.hsize{
      bins bytes_bin = {BYTE};
      bins halfword_bin = {HALFWORD};
      bins word_bin = {WORD};
    }
    hwdatal: coverpoint tx.hwdatal[0]{option.auto_bin_max = 6};
    read_writeXburstXhsizeXtrans: cross read_write, hburst, hsize , htrans;
  endgroup

  covergroup ahb_slave_cg ;
    hrdata: coverpoint tx.hrdata iff(tx.hwrite == READ)
      (option.auto_bin_max = 6);
    hready: coverpoint tx.hready{
      ignore_bins ignore_vals = {0};
    }
    hresp: coverpoint tx.hresp{
      ignore_bins ignore_vals = {RETRY,SPLIT};
      bins okay_bin = {OKAY};
      bins error_bin = {ERROR};
    }
    hreadyXhresp : cross hready, hresp;
  endgroup

  virtual function void write(ahb_transaction t);
    tx = t;
    ahb_master_cg.sample();
    ahb_slave_cg.sample();
  endfunction

  function new(string name = "ahb_coverage", uvm_component parent);
    super.new(name, parent);
    ahb_master_cg = new();
    ahb_slave_cg = new();
  endfunction

  //Report
  function void report_phase(uvm_phase phase);
    `uvm_info(get_type_name(), $sformatf("Master data coverage is: %f", ahb_master_cg.get_coverage()), UVM_MEDIUM)
    `uvm_info(get_type_name(), $sformatf("Slave data coverage is: %f", ahb_slave_cg.get_coverage()), UVM_MEDIUM)
  endfunction
endclass

```

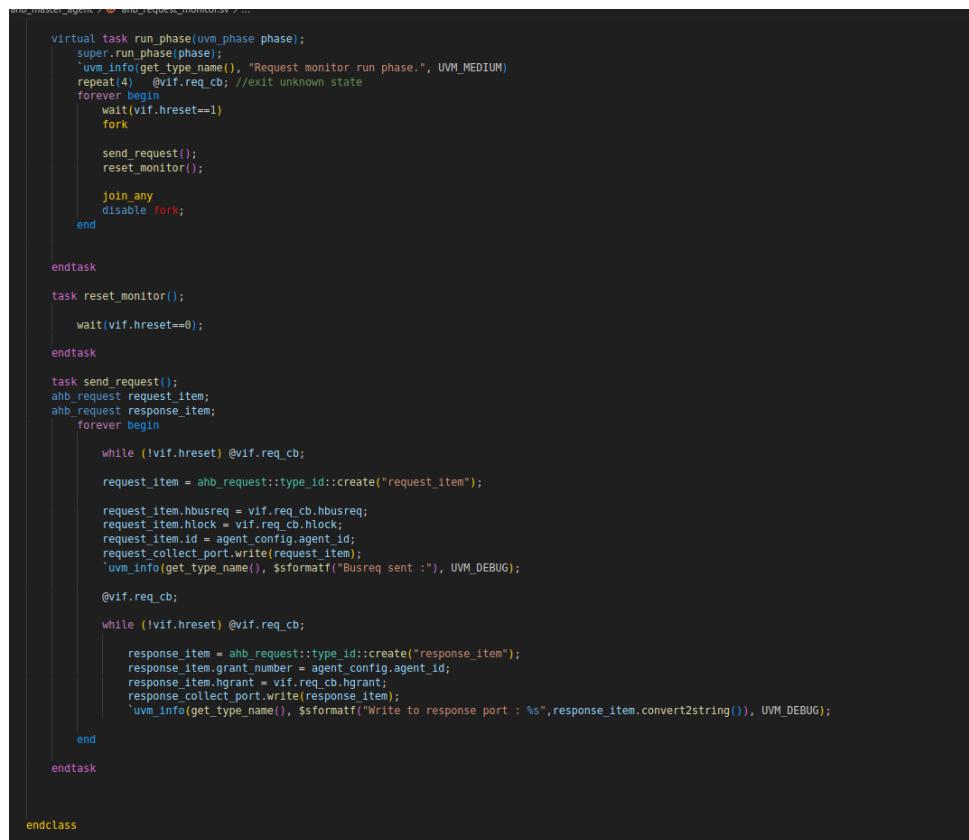
Fig 3.2.25 Clasa de coverage

Verificarea funcționalității de arbitraj a designului se realizează prin monitorizarea tranzacțiilor care conțin cererile pentru magistrală trimise de către master și trimiterea acestora către Scoreboard-ul de arbitraj.

Monitorul este responsabil de capturarea și analizarea tranzacțiilor care conțin cererile pentru magistrală. Acesta monitorizează activitatea magistralei și identifică tranzacțiile

relevante în funcție de cererile primite de la master. Aceste tranzacții sunt apoi trimise către Scoreboard-ul de arbitraj.

În cadrul Scoreboard-ului de arbitraj, tranzacțiile sunt comparate și evaluate conform algoritmului de arbitraj specific designului. Acesta verifică corectitudinea cererilor de pe magistrală și asigură că procesul de arbitraj se desfășoară conform specificațiilor. Scoreboard-ul de arbitraj poate detecta și raporta orice discrepanțe sau erori în procesul de arbitraj, oferind astfel informații importante pentru verificarea și validarea funcționalității de arbitraj a designului.



```

uvm_monitor.sv:16: virtual task run_phase(uvm phase phase);
uvm_info(get_type_name(), "Request monitor run phase.", UVM_MEDIUM)
repeat(4) @vif.req_cb; //exit unknown state
forever begin
    wait(vif.hreset==1)
    fork
        send_request();
        reset_monitor();
    join_any
    disable fork;
end

endtask

task reset_monitor();
    wait(vif.hreset==0);
endtask

task send_request();
    ahb_request request_item;
    ahb_request response_item;
    forever begin
        while (!vif.hreset) @vif.req_cb;
        request_item = ahb_request::type_id::create("request_item");
        request_item.hbusreq = vif.req_cb.hbusreq;
        request_item.hlock = vif.req_cb.hlock;
        request_item.id = agent_config.agent_id;
        request_collect_port.write(request_item);
        `uvm_info(get_type_name(), $sformatf("Busreq sent :"), UVM_DEBUG);
        @vif.req_cb;
        while (!vif.hreset) @vif.req_cb;
        response_item = ahb_request::type_id::create("response_item");
        response_item.grant_number = agent_config.agent_id;
        response_item.hgrant = vif.req_cb.hgrant;
        response_collect_port.write(response_item);
        `uvm_info(get_type_name(), $sformatf("Write to response port : %s", response_item.convertToString()), UVM_DEBUG);
    end
endtask

endclass

```

Fig 3.2.26 Monitor pentru cereri/răspunsuri pentru controlul magistralei

Scoreboardul de arbitraj realizează preluarea semnalelor necesare arbitrajului într-un moment specific al simulării, cum ar fi hbusreq, hlock și hgrant.

În interiorul scoreboard-ului de arbitraj, sunt declarate și configurate câmpurile necesare pentru funcționalitatea acestuia. Aceasta include construirea FIFO-urilor care vor reține cererile și răspunsurile asociate cu arbitrajul. FIFO-urile permit stocarea și gestionarea tranzacțiilor într-un mod ordonat și sincronizat.

De asemenea, scoreboard-ul de arbitraj are rolul de a configura și utiliza componența de coverage. Aceasta implică definirea și monitorizarea binurilor de acoperire relevante pentru evaluarea și raportarea informațiilor legate de procesul de arbitraj. Prin intermediul funcționalității de coverage, se poate obține o imagine detaliată a gradului de acoperire a diferitelor scenarii și combinațiilor de cereri și răspunsuri în cadrul procesului de arbitraj.

Scoreboardul de arbitraj are un rol esențial în verificarea și validarea corectitudinii și eficienței procesului de arbitraj al designului, asigurându-se că cererile de pe magistrală sunt tratate în mod corespunzător conform specificațiilor și regulilor de arbitraj.

```

class arbitration_scoreboard extends uvm_scoreboard;
  `uvm_component_utils(arbitration_scoreboard)

  uvm_tlm_analysis_fifo #(ahb_request) request_fifo[master_number];
  uvm_tlm_analysis_fifo #(ahb_request) response_fifo[master_number];

  bit enable_coverage;

  int match_nr = 0;
  int mismatches = 0;
  //ap for coverage
  uvm_analysis_port #(ahb_request) coverage_port;

  ahb_request predicted_transactions[$];
  ahb_request predicted_response;

  ahb_request requests_array[master_number];

  ahb_request response_array[master_number];
  ahb_request slave_response_array[slave_number];

  bit busreq_map[master_number];
  bit hlock_map[master_number];
  bit req_and_lock[master_number];

  int previous_granted_master = master_number - 1;

  function new(string name = "arbitration_scoreboard", uvm_component parent);
    super.new(name, parent);

    for (int i=0; i<master_number; ++i) begin
      request_fifo[i] = new($sformatf("request_fifo[%0d]",i),this);
    end
    for (int i=0; i<master_number; ++i) begin
      response_fifo[i] = new($sformatf("response_fifo[%0d]",i),this);
    end

  endfunction

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);

    if (enable_coverage) begin
      coverage_port = new("coverage_port",this);
    end
  endfunction

```

Fig 3.2.27 Scoreboard de arbitrage

Scoreboard-ul de arbitraj va avea două thread-uri pornite în faza run_phase(), în care se va utiliza un algoritm de predicție a viitorului grant, hgrant, în funcție de hbusreq și hlock.

```

task run_phase(uvm_phase phase);
    fork
        predictor();
        evaluator();
    join

    endtask

task predictor();
    forever begin
        //fork
        for ( int i=0; i<master_number; ++i) begin
            automatic int j = i;
            //https://verificationacademy.com/verification-methodology-reference/uvm/docs_1.1d/html/files/tlm1/uvm_tlm_ifs-svh.html#uvm_tlm_ifs_svh
            request_fifo[j].get(requests_array[j]);
        end
        //join

        for (int i=0; i<master_number; ++i) begin
            `uvm_info(get_type_name(), $sformatf("Request from master[%0d] : \n %s", requests_array[i].id,requests_array[i].convert2string));
        end
        store_in_map();
        predict_grant();
        clear_maps();
    end
    `uvm_info(get_type_name(),"/////////////////", UVM_DEBUG);

endtask

function void store_in_map();
    for (int i=0; i<master_number; ++i) begin
        if (requests_array[i].ibusreq == 1) begin
            busreq_map[requests_array[i].id] = 1;
        end
        if (requests_array[i].hlock == 1) begin
            hlock_map[requests_array[i].id] = 1;
        end
        req_and_lock[requests_array[i].id] = busreq_map[requests_array[i].id] & hlock_map[requests_array[i].id];
    end
    `uvm_info(get_type_name(), $sformatf("busreq_map : %p \n ", busreq_map), UVM_DEBUG);
    `uvm_info(get_type_name(), $sformatf("hlock_map : %p \n ", hlock_map), UVM_DEBUG);
    // `uvm_info(get_type_name(), $sformatf("req_and_lock : %p \n ", req_and_lock), UVM_HIGH);
endfunction

task evaluator();
    ahb request temp_predicted;
    ahb request temp_actual; //= ahb_request::type_id::create("temp_actual");
    forever begin
        for ( int i=0; i<master_number; ++i) begin
            automatic int j = i;
            response_fifo[j].get(response_array[j]);
        end

        temp_predicted = predicted_transactions.pop_front();

        if (response_array[temp_predicted.grant_number].hgrant == 1) begin
            match_nr++;
            if (enable_coverage) begin
                coverage_port.write(temp_predicted);
            end
        end else begin
            mismatches++;
            `uvm_info(get_type_name(), $sformatf("Bus request was unmatched "), UVM_MEDIUM);
            `uvm_info(get_type_name(), $sformatf("Predicted response was : %s",temp_predicted.grant_number), UVM_MEDIUM);
            `uvm_info(get_type_name(), $sformatf("Actual response : %s", response_array[temp_predicted.grant_number].convert2string()), UVM_MEDIUM);
        end
    end
endtask

function void predict_grant();
    int highest_priority_master = master_number - 1;
    for (int i=0; i<master_number; ++i) begin
        if (busreq_map[i] == 1 ) begin
            if( highest_priority_master > i )
            begin
                highest_priority_master = i;
                break;
            end
        end
    end
    if (!(busreq_map[previous_granted_master] && hlock_map[previous_granted_master])) begin
        previous_granted_master = highest_priority_master;
    end
    predicted_response = ahb_request::type_id::create("predicted_response");
    predicted_response.grant_number = previous_granted_master;
    // busreq signals to send for coverage
    predicted_response.busreq_map = busreq_map;
    predicted_response.hlock_map = hlock_map;
    `uvm_info(get_type_name(), $sformatf("Predicted grant is : %d \n ", predicted_response.grant_number), UVM_DEBUG);
    predicted_transactions.push_front(predicted_response);
endfunction

virtual function void check_phase(uvm_phase phase);
    `uvm_info(get_type_name(),$sformatf("BUSREQ SCB: "),UVM_MEDIUM);
    `uvm_info(get_type_name(),$sformatf("Matches: %0d ",match_nr),UVM_MEDIUM);
    `uvm_info(get_type_name(),$sformatf("Mismatches: %0d ",mismatches),UVM_MEDIUM);
endfunction

```

Fig 3.2.28 Scoreboard de arbitrage

Simularea se poate realiza folosind o interfață grafică precum SimVision. Aici se poate efectua depanarea în cod, se pot vedea conținutul obiectelor, ierarhia de obiecte și formele de undă.

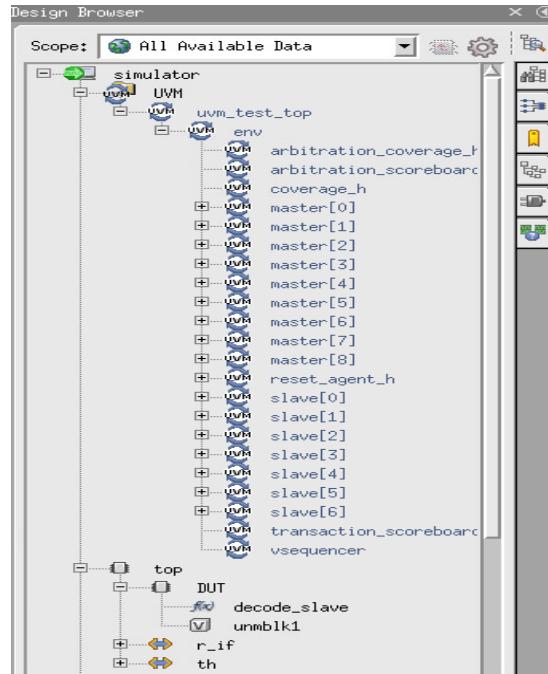


Fig 3.2.29 Ierarhia de obiecte

Analizarea formelor de undă în SimVision este o caracteristică esențială a instrumentului de depanare. Permite utilizatorilor să vizualizeze și să analizeze semnalele în timpul simulării. SimVision oferă o interfață grafică intuitivă care facilitează observarea și înțelegerea valorilor semnalelor, tranzițiilor și relațiilor dintre ele.

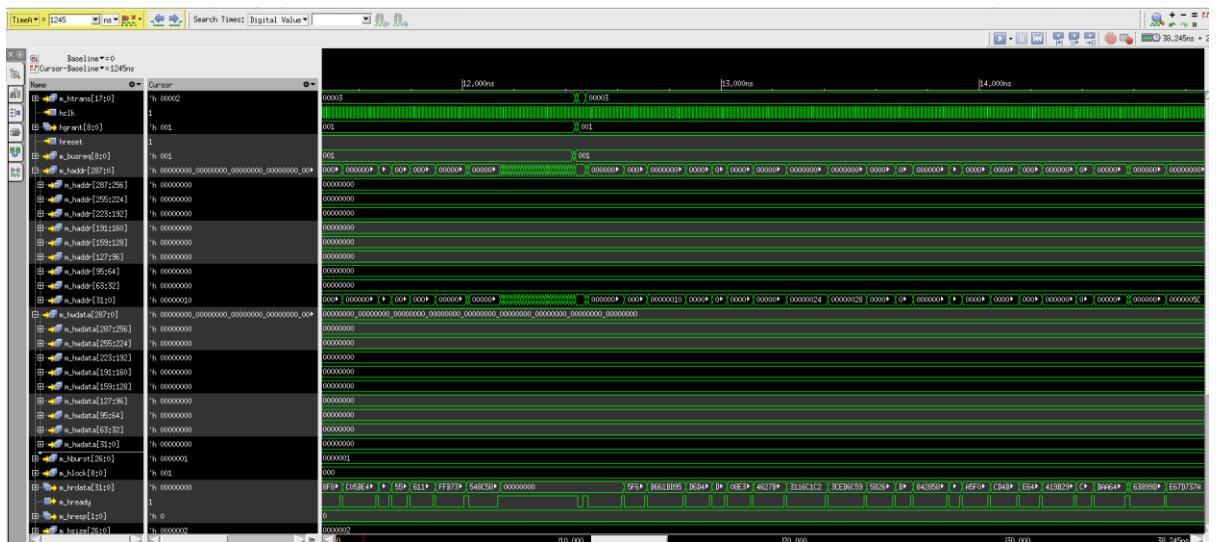


Fig 3.2.30 Vizualizarea simulării și a formelor de undă

Analiza rezultatelor și vizualizarea acoperirii (coverage) vor fi realizate folosind IMC (Incisive Metric-Driven Verification) de la Cadence. Acest instrument oferă capacitați puternice pentru analiza și evaluarea acoperirii în timpul verificării. Utilizând IMC, se pot observa detaliile legate de proprietățile SVA (Assertion-Based Verification) care au trecut sau au eșuat în timpul simulării. De asemenea, se pot examina obiectele de acoperire care au adunat date relevante despre acțiunile și comportamentul sistemului verificat. IMC permite vizualizarea acoperirii în diferite aspecte, cum ar fi acoperirea de cod (Code Coverage), acoperirea stărilor mașinilor de stare (State Coverage), precum și monitorizarea activității de toggle a bitilor. Aceste funcționalități facilitează evaluarea acoperirii și analiza rezultatelor pentru a asigura o verificare exhaustivă și completă a sistemului.

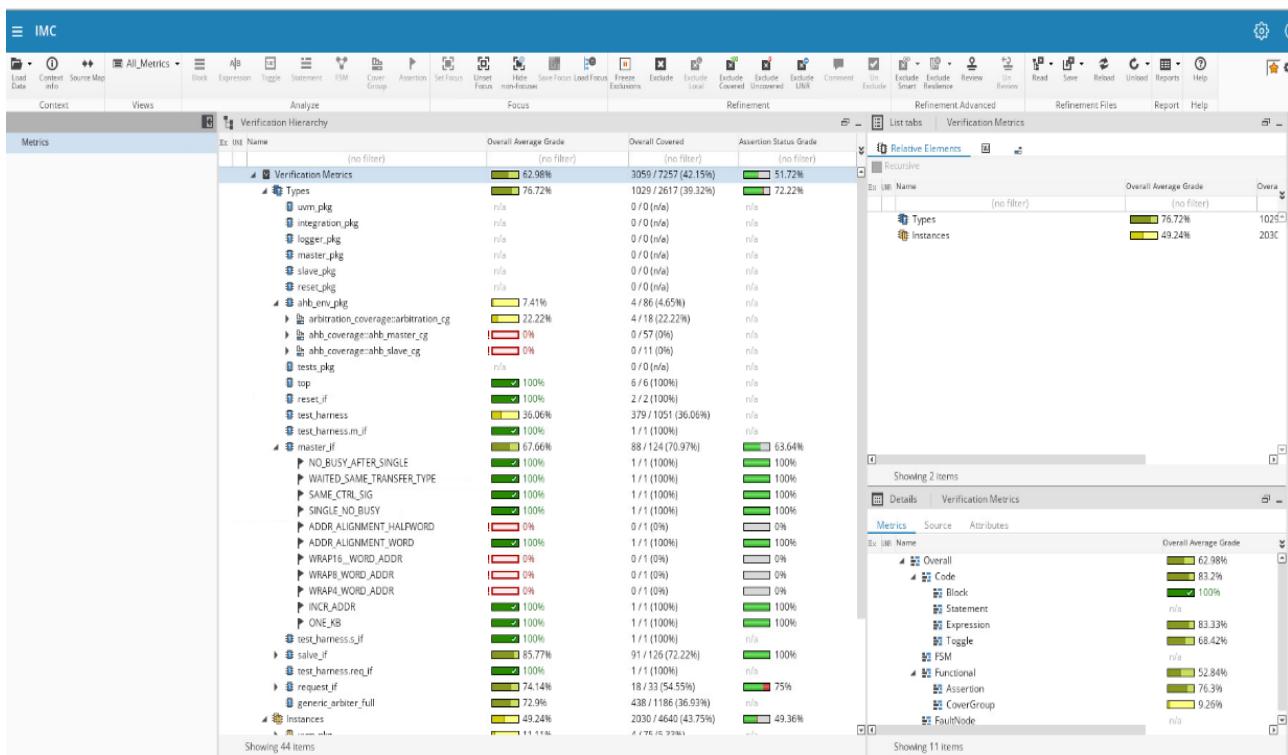


Fig 3.2.31 Analiza rezultatelor de coverage in IMC

3.3 Analiza rezultatelor

În procesul de verificare a designului (design verification), analizarea rezultatelor joacă un rol crucial în evaluarea calității și adecvării testelor de verificare aplicate.

IMC oferă un set de funcționalități puternice pentru analiza rezultatelor verificării. Acestea includ vizualizarea detaliată a acoperirii (coverage), examinarea rezultatelor testelor, evaluarea proprietăților SVA (Assertion-Based Verification) și monitorizarea activității de toggle a bitilor. Utilizând IMC, poți obține o înțelegere mai profundă a performanței testelor și a calității acoperirii, identificând zonele cu acoperire insuficientă și posibile probleme în design.

vManager este o platformă de gestionare a regresiilor care te ajută să organizezi și să rulezi o suită de teste în mod automat și eficient. Acest instrument permite planificarea,

programarea și executarea testelor într-un mod reproducibil și controlat. Poți configura seturi de teste și criterii de acceptare pentru a asigura o verificare completă și exhaustivă a designului. vManager facilitează raportarea rezultatelor și identificarea posibilelor erori sau discrepanțe între rezultatele așteptate și cele obținute.

În timpul procesului de debug, logurile și instrumente precum Simvision joacă un rol important. Logurile de simulare capturează informații despre activitatea sistemului și pot fi utilizate pentru analiza și identificarea erorilor sau discrepanțelor. Simvision oferă o interfață grafică pentru analizarea formelor de undă, permitând observarea comportamentului sistemului într-un mod vizual și intuitiv. Aceasta facilitează identificarea problemelor, debugarea și obținerea de insight-uri cu privire la interacțiunile și starea sistemului.

Pe măsură ce procesul de verificare avansează, este adesea necesar să ajustezi și să revizuieni planul de verificare. Aceasta poate fi adaptat în funcție de rezultatele obținute, prioritățile identificate, feedback-ul echipei și cerințele de verificare. Astfel, planul de verificare devine un instrument flexibil și iterativ care permite optimizarea procesului de verificare și obținerea unei acoperiri adecvate a funcționalităților și scenariilor critice din design.

În concluzie, analiza rezultatelor în design verification este un proces continuu și multidimensional, în care se utilizează instrumente precum IMC, vManager, logurile și Simvision pentru evaluarea calității testelor, identificarea problemelor și obținerea de insight-uri esențiale pentru validarea și îmbunătățirea designului. Ajustarea și revizuirea planului de verificare este necesară.

4. Concluzii

În concluzie, procesul de verificare a designului reprezintă o etapă esențială în dezvoltarea sistemelor de proiectare complexe. Aceasta implică utilizarea unor tehnici și instrumente avansate pentru a asigura funcționalitatea, corectitudinea și robustețea designului. Pe parcursul acestui proces, diversele componente și etape ale verificării sunt critice pentru atingerea obiectivelor și calității finale a produsului.

Un aspect important în verificarea designului este reprezentat de definirea unui plan de verificare cuprinzător și eficient. Planul de verificare trebuie să identifice și să acopere toate funcționalitățile critice, să includă scenarii de test relevante și să stabilească obiective clare de acoperire și calitate. Flexibilitatea și adaptabilitatea planului de verificare sunt esențiale pentru a aborda schimbările și provocările pe parcursul proiectului.

În cadrul verificării, utilizarea de tehnologii și metodologii avansate, precum UVM și SVA, aduce beneficii semnificative. Acestea facilitează modelarea și simularea

comportamentului designului, permit crearea de secvențe de test flexibile și ușor de gestionat, facilitează verificarea proprietăților și asigură o acoperire eficientă a scenariilor critice.

Un alt aspect cheie în procesul de verificare este analiza și interpretarea rezultatelor. Instrumente precum IMC, vManager și Simvision furnizează insight-uri importante și facilitează identificarea erorilor, depurarea codului și evaluarea acoperirii testelor. Utilizarea acestor instrumente în mod eficient poate reduce timpul de debug și crește eficiența generală a procesului de verificare.

Pe măsură ce domeniul verificării continuă să evolueze, noi provocări și cerințe apar constant. Acest lucru necesită o abordare continuă de învățare și adaptare, precum și o colaborare strânsă între echipele de proiectare și verificare. Comunicarea eficientă, documentarea adecvată și gestionarea eficientă a resurselor sunt elemente cheie în atingerea succesului în verificarea de design.

În final, verificarea designului este un proces complex și necesar pentru asigurarea calității și funcționalității sistemelor de proiectare. Prin utilizarea metodologiilor și instrumentelor adecvate, o planificare și o execuție riguroasă a testelor, analiza și interpretarea rezultatelor și adaptarea continuă, este posibilă obținerea unui produs verificat și validat în mod corespunzător. Verificarea joacă un rol esențial în asigurarea conformității cu specificațiile și cerințele impuse, contribuind la dezvoltarea de produse fiabile și de înaltă calitate.

Bibliografie

- [1] T. Alsop, Integrated circuits semiconductor market size worldwide from 2009 to 2022 Available: statista.com, <https://www.statista.com/statistics/519456/forecast-of-worldwide-semiconductor-sales-of-integrated-circuits/>.
- [2] H. Foster, Siemens, IC/ASIC Design Trends Available: siemens.com, <https://blogs.sw.siemens.com/verificationhorizons/2020/12/22/part-7-the-2020-wilson-research-group-functional-verification-study/>.
- [3] H.Foster, Siemens, Redefining Verification Performance Available:siemens.com, <https://blogs.sw.siemens.com/verificationhorizons/2010/08/08/redefining-verification-performance-part-2>.
- [4] H.Foster, Siemens, Redefining Verification Performance Available:siemens.com <https://blogs.sw.siemens.com/verificationhorizons/2021/01/06/part-8-the-2020-wilson-research-group-functional-verification-study/>
- [5] Wikipedia.org, Advanced Microcontroller Bus Architecture Available: Wikipedia.org, https://en.wikipedia.org/wiki/Advanced_Microcontroller_Bus_Architecture
- [6] ARM Holdings, AMBA standard overview Available: arm.com, <https://developer.arm.com/Architectures/AMBA>
- [7] Ramdas Mozhikunnath, Understanding AMBA Bus Architechture and Protocols Available: ansysilicon.com, <https://ansysilicon.com/understanding-amba-bus-architecture-protocols/>
- [8] Unspecified author, AMBA (Advanced Microcontroller Bus Architecture) Wiki, Available: semiwiki.com, <https://sem/wiki/semiconductor-ip-wikis/amba-advanced-microcontroller-bus-architecture/>
- [9] ARM Holdings , AMBA™ Specification (Rev 2.0) Available: arm.com, <https://developer.arm.com/documentation/ihi0011/latest/>
- [10] W. K. Lam (2008), Hardware Design Verification: Simulation and Formal Method-Based Approaches, Prentice Hall, ISBN: 978-0137010929
- [11] Harry Foster,Siemens,The 2020 Wilson Research Group Functional Verification Study - IC/ASIC Language and Library Adoption Trend Available: siemens.com, <https://blogs.sw.siemens.com/verificationhorizons/2021/01/20/part-10-the-2020-wilson-research-group-functional-verification-study/>
- [12] Chris Spear , Greg Tumbush, SystemVerilog for Verification Available: springer.com <https://link.springer.com/book/10.1007/978-1-4614-0715-7>

- [13] Ray Salemi, Mentor Graphics, UVM Cookbook - A Guide to Applying the Universal Verification Methodology for SystemVerilog, 2015. Available: verificationacademy.com, <https://verificationacademy.com/cookbook>
- [14] Ray Salemi, Mentor Graphics, UVM Coverage Cookbook - A Guide to Applying the Universal Verification Methodology for SystemVerilog Coverage, 2016. Available: verificationacademy.com, <https://verificationacademy.com/cookbook>
- [15] "IEEE Standard for Universal Verification Methodology Language Reference Manual," in IEEE Std 1800.2-2020 (Revision of IEEE Std 1800.2-2017) , vol., no., pp.1-458, 14 Sept. 2020, doi: 10.1109/IEEESTD.2020.9195920.
- [16] "IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language - Redline," in IEEE Std 1800-2009 (Revision of IEEE Std1800-2005) - Redline , vol., no., pp.1-1346, 11 Dec. 2009.
- [17] Vivek Arya , " VLSI Design" <https://www.linkedin.com/pulse/vlsi-design-flow-vivek-arya/>
- [18] Jeff Montesano , Verilab Inc. - Mastering Reactive Slaves in UVM
- [19] Cadence Design Systems Available: cadence.com, https://www.cadence.com/en_US/home/tools/tools-a-z.html