2023

Proiectarea unei unități de management pentru o memorie virtuală

STUDENT: VOLCOV SABINA,

Anul III, licență

Facultatea: Automatică și Calculatoare

Specializarea: Calculatoare și Tehnologia Informației

UTCN, CLUJ-NAPOCA

Cuprins:

1. Introducere

1.1 Propunere de proiect

Proiectul presupune dezvoltarea în mediul VHDL a unei unități de management pentru o memorie virtuală. Memoria principală are dimensiunea de 1MB (2^{20} bytes), cea virtuală are dimensiunea de 1GB (2^{30} bytes), iar dimensiunea paginilor folosite pentru stocarea datelor este de 4KB (2^{12} bytes). Se consideră o magistrală de date pe 16 biți și maparea paginilor de tip asociativ.

Memoria principală reprezintă 256 de cadre, iar cea virtuală 2^{18} pagini, asocierea între ele efectuându-se cu ajutorul unei tabele TLB și a unui page table. Mecanismul de înlocuire a paginilor în memorie folosit este LRU (least recently used), iar pentru scrierea datelor se va aplica tehnica de write-back (motivată de costul înalt al unei scrieri write-through).

Adresa după care se efectuează căutarea datelor în memorie conține 18 biți pentru alegerea paginii virtuale (translatabili la 8 biți care determină numărul paginii din memoria fizică principală) și 12 biți de offset pentru adresarea individuală a fiecărui octet dintr-o pagină.

Unitatea de management va fi cea responsabilă de efectuarea translatării adreselor primite din formatul instrucțiunilor de lucru cu memoria (load/store sau read/write), dar și de încărcarea paginilor lipsă în cazul unui page fault. Un page fault va provoca o excepție și va gestionat ca atare. În etapa finală, proiectul se va testa pe câteva astfel instrucțiuni de citire și scriere.

1.2 Planificare

Elaborarea proiectului se va desfășura pe parcursul a 5 laboratoare, astfel:

1) **Sesiunea de proiect 1:** definirea propunerii de proiect și planificării, elaborarea introducerii din documentație + definirea unui nou proiect în Vivado cu entitățile necesare;
2) **Sesiunea de proiect 2:** definirea funcționalităților pentru fiecare entitate fizică în parte + elaborarea părții de analiză și design din documentație;
3) **Sesiunea de proiect 3:** continuarea descrierii în Vivado a funcționalităților proiectului;
4) **Sesiunea de proiect 4:** finalizarea și testarea proiectului, elaborarea părții de testare/concluzii din documentație;
5) **Sesiunea de proiect 5 (finală):** Redactarea documentației, completarea bibliografiei, concluziilor, testarea finală pe plăcuță a proiectului;

2. Studiu bibliografic

2.1 Mașini și memorii virtuale

Mașinile virtuale (eng. Virtual Machines, VMs) au fost dezvoltate inițial pe la jumătatea anilor 1960 și au rămas o parte importantă a mainframe-ului de calculatoare pe parcursul deceniilor care au urmat.

Popularitatea recentă a conceptului se explică prin mai multe realități actuale:

- Eșecurile sistemelor de operare standard în ceea ce privește securitatea și fiabilitatea;
- Importanța tot mai mare a izolării și protecției sistemelor moderne;
- Necesitatea de a împărți același sistem cu mai mulți utilizatori, în special în cazul Cloud computing;
- Creșterea dramatică a vitezei procesoarelor din ultimii ani, care fac overhead-ul cauzat de utilizarea unei mașini virtuale să fie acceptabil;

Mașinile virtuale au avantajul de a oferi iluzia că un utilizator are toate resursele calculatorului la dispoziția sa, acesta rulând mai multe VM-uri care suportă diverse sisteme de operare. Dacă convențional, un dispozitiv "deține" toate resursele hardware, în cazul virtualizării, mai multe sisteme de operare impart între ele aceste resurse.

Aceeași idee de distribuire a resurselor hardware apare și în cazul memoriilor virtuale, concept introdus în 1961 de Fotheringham, și care presupune extinderea memoriei principale cu un spațiu virtual de adrese, astfel încât mai multe programe să poată rula în același timp, fără a se suprapune.

Motivațiile principale care au dus la implementarea unui sistem în care memoria principală să acționeze ca un "cache" pentru stocarea de tip disc magnetic sunt:

1) Permiterea unei distribuiri eficiente și sigure a datelor între mai multe programe, cum ar fi a celor necesare pentru mai multe mașini virtuale în Cloud computing;
2) Permiterea ca un singur program să depășească dimensiunea limitată a memoriei principale;

De asemenea, memoria virtuală simplifică încărcarea programului pentru execuție, permițând relocarea (maparea unor adrese virtuale la alte adrese fizice înainte ca acestea să fie folosite în program). Mecanismul de relocare presupune încărcarea în memorie a unei porțiuni de cod la orice adresă.

Caracteristica distinctivă a utilizării memoriilor virtuale o constituie împărțirea acesteia și a memoriei principale, în unități de aceeași dimensiune (de obicei, 4KB sau 16KB), numite pagini (eng. pages) în primul caz, și cadre (eng. frames) în cel de-al doilea. Astfel, la nevoie datele sunt accesate direct din memoria principală, iar dacă nu se regăsesc acolo, pagina căutată se aduce de pe hard disk și după mai multe cicluri de ceas, datele pot fi iarăși accesate. Pe de altă parte, transferul de date între RAM și disc magnetic este lent, efectuându-se doar în blocuri întregi, de o dimensiune fixă.

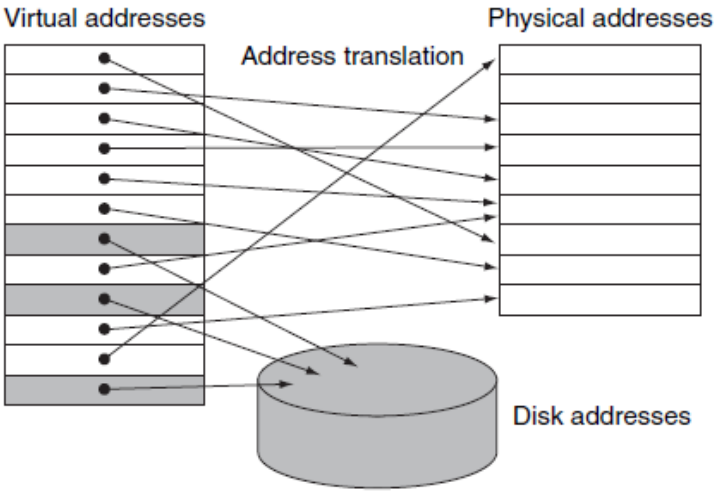

Figura 1. Mecanismul din spatele unei memorii virtuale

Pentru a face legătura dintre paginile reale și cele virtuale, e necesară o unitate de management care să efectueze o translatare a adreselor și să păstreze informații despre corespundența între cele două memorii.

Adresa după care se face căutarea este formată din 2 câmpuri: numărul paginii virtuale și offset-ul din cadrul paginii unde trebuie găsite datele. De exemplu, pe arhitectura RISC-V de 64 de biți, adresa virtuală e mapată pe 48 biți. Partea superioară a adresei reprezintă numărul paginii fizice din memorie, iar ceilalți sunt offset-ul din interiorul paginii. Biții de offset dau totodată și dimensiunea paginii de pe sistem. De exemplu, o pagină de 4KB va conține în formatul adresei 12 biți pentru adresarea fiecărui octet în parte.

Un lucru interesant este că numărul paginilor virtuale poate să difere de numărul de pagini din memoria principală, și anume să fie mai mare. Acest fapt stă la baza iluziei de spațiu nelimitat disponibil în memorie.

2.2 Implementarea memoriilor virtuale

Multe dintre deciziile de implementare în ceea ce ține de sisteme cu memorie virtuală sunt motivate de costurile uriașe ale unui eșec de găsire al datelor (page fault). Un asemenea eșec ar dura milioane de cicluri de ceas pentru a fi procesat, de aceea au ajuns să se aplice anumite idei de bază pentru proiectarea unor astfel de memorii:

- Paginile trebuie să fie destul de mari ca să amortizeze timpul de acces mare, dar destul de mici pentru a evita fragmentarea internă a datelor;
- Mecanisme precum plasarea asociativă a paginilor în memorie sunt atractive, deoarece reduc șansa de page fault;
- Page fault-urile pot fi rezolvate în software, pentru că overhead-ul va fi limitat față de timpul de acces la disk. De asemenea, se pot alege algoritmi software mai eficienți pentru plasarea paginilor, care să reducă și mai mult rata de neregăsire a informațiilor;
- Tehnica de write-through, deseori aplicată pentru memoriile cache, nu va funcționa în cazul memoriilor virtuale, deoarece o operație de write ar dura prea mult; în schimb, sistemele cu memorie virtuală implementează tehnica de write-back;

Dar chiar și în cazul plasării asociative, trebuie să se păstreze informații despre cadrele din memorie, care corespund paginilor din memoria virtuală. Aceste informații se regăsesc într-un page table, o structură de date registru, care pe lângă numărul paginii fizice presupune un bit de validitate (0 -> pagina nu există în memoria principală => page fault, 1 -> pagina există în memorie și datele se pot accesa). Spre deosebire de memoriile cache, un page table nu necesită un tag, deoarece corespondența se face pentru fiecare pagină virtuală posibilă.

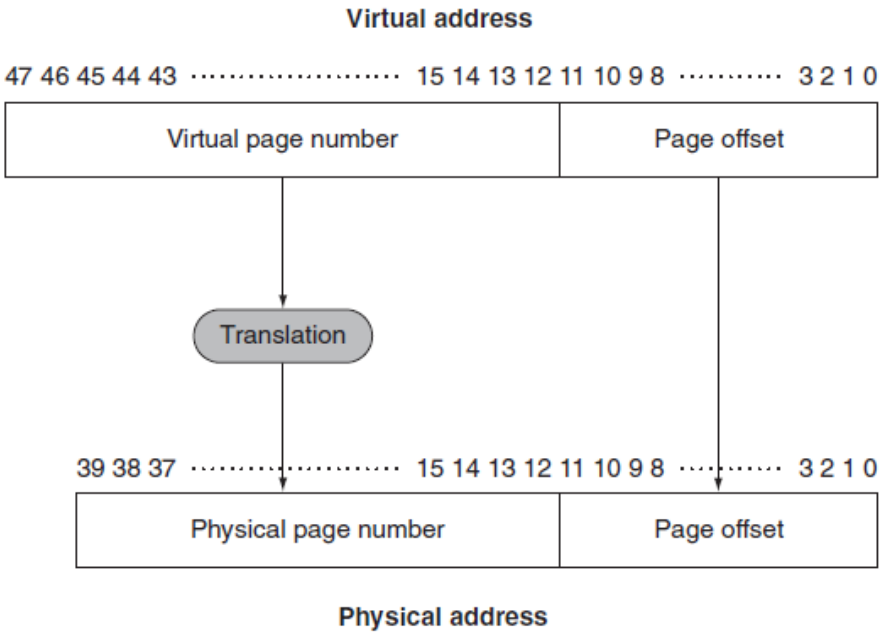

Figura 2. Corespondența dintre o pagină virtuală și o pagină fizică

În caz că are loc page fault, se activează o excepție, iar controlul asupra dispozitivului îl primește sistemul de operare. Acesta trebuie să găsească pagina lipsă în următorul nivel al ierarhiei de memorii (de obicei, memorie flash sau disc magnetic) și să decidă unde să plaseze pagina cerută în memorie.

Pentru că nu se poate cunoaște din timp când o pagină din memorie va fi înlocuită, sistemul de operare creează de obicei desul spațiu pe nivelul secundar de stocare pentru toate paginile unui proces. Acest spațiu poartă denumirea de ”swap space”. Iar pentru a decide căreia dintre paginile deja existente în memorie i se va face ”swap” cu cea căutată, în page table mai apare o intrare, reference/access bit.

Metoda cea mai utilizată pentru înlocuirea paginilor este algoritmul LRU (least recently used), care verifică reference bit și alege pagina cea mai veche și mai puțin accesată dintre toate intrările din memoria principală. LRU se bazează pe principiul localității, care presupune că paginile care au fost folosit intens în instrucțiunile precedente pot fi folosite din nou, curând. Pe de altă parte, paginile care au fost accesate cu mult timp în urmă, nu vor fi necesare decât după multe alte instrucțiuni.

Descrierea unui algoritm precum LRU nu este tocmai simplă: pentru a memora accesele la pagină, se implementează un counter care se incrementează automat la fiecare instrucțiune ce ține de fiecare pagină. La înlocuire, sistemul alege pagina cu cel mai mic număr de referențieri. Dezavantajul este păstrarea valorii în page table, pe lângă celelalte date, ceea ce duce la o dimensiune foarte mare a tabelei.

Există totuși și alte metode folositoare, precum algoritmul Not Recently Used, care verifică biții de referință și dirty bit asociați fiecărei pagini pentru a lua o decizie cât mai optimă. Paginile pot fi împărțite în 4 categorii:

1) Clasa 0 – nu a fost referențiată, nu a fost modificată;
2) Clasa 1 – nu a fost referențiată, a fost modificată;
3) Clasa 2 – a fost referențiată, nu a fost modificată;
4) Clasa 3 – a fost referențiată, a fost modificată;

Algoritmul elimină la înlocuire paginile din clasele cele mai joase, considerând că e mai bine să rămână o pagină nemodificată, dar căutată intensiv, decât o pagină modificată care nu a mai apărut în instrucțiuni de mai multe cicluri de ceas.

FIFO (First-In, First-Out) este o altă metodă cu timp de overhead mic, care permite alegerea unei pagini de șters din memorie. Pentru acest algoritm, sistemul păstrează o listă cu toate paginile din memorie la un moment dat, cu cea mai recentă ca tail, și cea mai veche ca head. Când are loc page fault, pagina din head este eliminată, iar noua pagină necesară în memorie adăugată la final.

Algoritmul Second-Chance reprezintă o modificare simplă a mecanismului FIFO, care verifică și bitul de referință înainte de a elimina o pagină. Dacă acest bit este 0 pentru head – pagina este atât ”veche”, cât și neutilizată, de aceea se înlocuiește instant. În schimb, dacă bitul are valoarea 1, pagina se pune la sfârșitul listei, ca și când abia a fost adăugată în memorie, și se continuă cu verificarea următoarei pagini.

În caz că toate paginile au fost referențiate recent, algoritmul degenerează în FIFO pur. Cu alte cuvinte, se ajunge din nou la pagina de început, care are acum bitul de referință setat pe 0. Astfel, algoritmul se garantează terminarea algoritmului.

Ultimul dintre algoritmii cel mai des folosiți la înlocuire este Clock Page. Diferența față de Second-Chance constă în faptul că lista este acum circulară, iar un pointer indică cea mai veche pagină din memorie. Dacă are loc page fault, e inspectată respectiva pagină, astfel încât bitul de referință setat pe 0 va provoca eliminarea ei, iar setat pe 1 va determina trecerea pointer-ului la următoarea pagină și curățarea bitului de referință. Pasul se repetă până se identifică o pagină nereferențiată recent.

2.3 Gestionarea excepțiilor și întreruperilor

Toate procesoarele moderne (inclusiv majoritatea microprocesoarelor) oferă suport pentru tratarea unor evenimente interne și externe, care modifică execuția normală instrucțiune cu instrucțiune a unui program. Aceste evenimente au primit denumirea de excepții (interne și sincrone) și întreruperi (externe și asincrone), iar cele mai comune exemple sunt: overflow pentru operații aritmetice pe întregi sau în virgulă mobilă, împărțirea la zero, instrucțiune invalidă, întrerupere cauzată de un numărător intern, întrerupere de la periferice (buton, switch etc.) și acces invalid la memorie.

Atunci când se aruncă o excepție sau întrerupere, sistemul de operare execută o secvență de pași înainte de a putea continua execuția normală a unui program. Procesorul trebuie să oprească execuția instrucțiunii curente, să salveze adresa instrucțiunii din program la care a

rămas, să sară la rutina de întrerupere păstrată în memoria ROM și să o execute, iar apoi să revină în programul pe care l-a întrerupt și să continue cu următoarea instrucțiune.

2.3.1 Page fault handling

Sistemul real ce implementează o memorie virtuală trebuie să știe cum să se comporte în caz că nu se regăsește în memoria principală pagina unde se dorește o scriere sau o citire și apare necesitatea de a o aduce din stocarea secundară pentru a o accesa.

Mecanismul de rezolvare al acestei excepții include următorii pași:

1) Hardware-ul acaparează kernel-ul, salvând program counter-ul pe stivă; în același timp, pe mai multe sisteme, în registre CPU speciale se salvează anumite informații despre starea instrucțiunii curente;
2) O rutină în limbaj de asamblare este activată pentru a salva registrele generale de program și alte date volatile, pentru a preveni distrugerea lor de către sistem;
3) Sistemul de operare descoperă că a avut loc un page fault, și încearcă să determine care adresă virtuală l-a provocat. Deseori, unul dintre regiștrii hardware păstrează această informație. În caz contrar, sistemul de operare trebuie să obțină program counter-ul, instrucțiunea curentă și să o parseze;
4) Odată ce se cunoaște adresa virtuală cauză, sistemul verifică validitatea adresei și dacă protecția este consistentă cu aceasta. Dacă nu se îndeplinește condiția precedentă, sistemul verifică dacă există o poziție liberă pentru încărcarea paginii din memorie. Când nu mai sunt poziții libere, este rulat algoritmul de înlocuire al paginii;
5) Dacă pagina selectată pentru a fi înlocuită are dirty bit activat, pagina este programată pentru transfer pe disk, și are loc un context switch care suspendă procesul de eroare până transferul se încheie;
6) De îndată ce pagina e ”curată”, sistemul de operare caută pe disk adresa paginii necesare și programează o operația de aducere în memoria principală. Cât are loc încărcarea paginii, este suspendat procesul de eroare, permițând rularea unui alt proces care este disponibil;
7) Când întreruperea disk-ului indică faptul că pagina a fost încărcată, page table-urile sunt actualizate pentru a-i reflecta noua poziție;
8) Instrucțiunea care a cauzat eroarea este readusă la starea inițială, iar program counter-ul resetat să pointeze la acea instrucțiune;
9) Este programat procesul de eroare, iar sistemul de operare revine la rutina în limbaj de asamblare care l-a apelat;
10) Rutina reîncarcă regiștrii și celelalte informații de stare salvate la începutul executării procesului și se revine în user space, unde execuția programului continuă, ca și când page fault-ul nu a avut loc;

2.3.2 Alte erori

O eroare mai gravă, care nu poate fi rezolvată de către sistemul de operare și mecanismele de memorie este segmentation fault (din engl. ”eroare de segmentare”), care determină că programul pur și simplu a încercat să acceseze o adresă invalidă, care fie nu există, fie are permisiuni restrânse față de utilizatorii care încearcă să o acceseze.

2.4 Optimizarea mecanismului de translatare -> TLB

TLB (translation-lookaside buffer) este un tip special de cache care păstrează informații despre cele mai recente translatări de adresă. Formatul unei intrări din tabela TLB conține: un tag, date, bit de referință/acces și dirty bit.

În momentul oricărei operații cu memoria, se caută numărul paginii virtuale în TLB. Dacă are loc un hit, numărul paginii fizice e folosit pentru a forma adresa, și bitul de referință corespunzător intrării din tabelă este activat. Însă dacă are loc un miss, trebuie să se determine ce tipul acestei erori.

În cazul implementării memoriilor virtuale cu TLB, eșecul de a găsi o pagină poate fi de 2 tipuri: hard miss (page fault), când pagina într-adevăr nu se regăsește în memoria principală și trebuie adusă din stocarea secundară, și soft miss, care se referă la lipsa translatării în buffer, deși pagina există în memorie.

Când are loc un soft miss, soluția este simplă: se caută datele despre pagină în page table, apoi se înlocuiește cu acestea o altă intrare din TLB și se continuă cu operația specificată în instrucțiunea curentă. Gestionarea unui hard miss, însă, presupune intrarea în mecanismul de page fault. Abia după ce pagina a fost adusă de pe disk în memoria principală și page table-ul, împreună cu buffer-ul TLB au fost actualizate, se poate continua execuția instrucțiunii.

Avantajul utilizării unui TLB constă așadar în rapiditatea regăsirii translatării pentru o pagină virtuală accesată frecvent și totodată în protejarea unei pagini de a fi scrisă, în cazul în care mecanismul de protecție devine necesar. Atunci apare în tabelă un nou câmp, protection bit, care se setează pe 0 pentru a specifica operații de read/write, și pe 1 doar pentru operații de read.

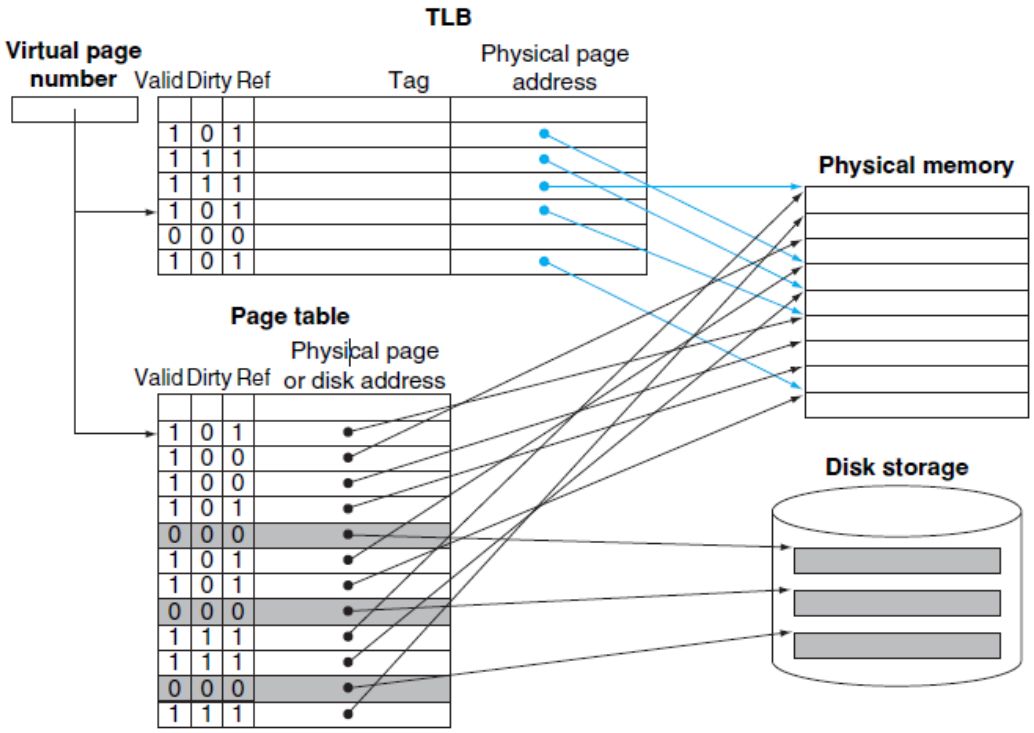

Figura 3. Implementarea unei memorii virtuale cu TLB

2.5 Citirea și scrierea datelor în memoria virtuală

Protecția unei pagini de la a fi accesată doar pentru citire sau pentru citire și scriere se poate implementa dacă se îndeplinesc 3 condiții:

1) Sistemul de operare are atât user space, cât și kernel space;
2) O anumită porțiune a procesorului poate fi doar citită, dar nu și scrisă de către un utilizator;
3) Procesul are mecanisme care să îi permită să facă context switch-uri între user mode și kernel mode în timp ce rulează;

Operațiile de bază, citirea și scrierea datelor acolo unde sunt stocate, se fac în cadrul unei memorii virtuale asemănător cu citirea și scrierea din memoria obișnuită, cu diferența că adresa dată în codul instrucțiunii care se execută pe procesor nu este una reală, ci trebuie translatată conform unei formule matematice pentru a trece din spațiul virtual la spațiul real de adrese.

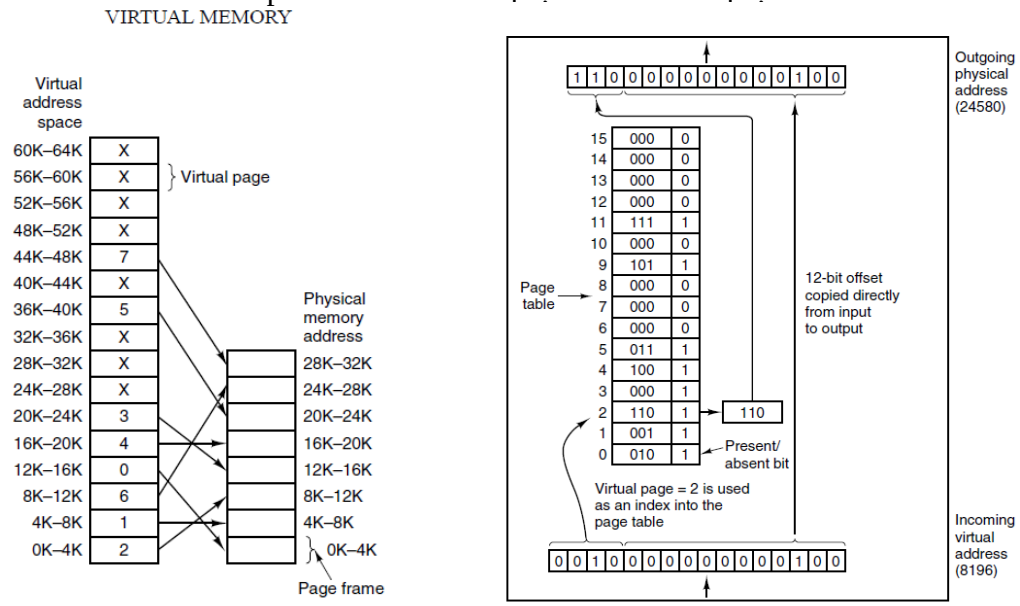

Figura 4. Translatarea unei adrese virtuale folosind un page table

În ce privește operația de scriere, memoria virtuală implementează mecanismul write-back, mult mai ieftin decât write-through, adică în pagina din memoria principală se fac scrieri individuale, apoi la înlocuirea respectivei pagini, datele sunt copiate integral și în stocarea secundară. Condiția pentru efectuarea acestui transfer este dată de dirty bit (din page table sau TLB), doar valoarea ”1” va duce la scrierea paginii virtuale pe discul magnectic.

Write-back are mai multe avantaje:

- Cuvinte individuale (16 sau 32 de biți) pot fi scrise de către procesor mai rapid în memoria principală decât pe disc magnetic;
- Mai multe scriere într-un singur bloc au nevoie de o singură scriere la nivelul mai jos din ierarhia de memorii;
- Când blocurile se scriu pe stocarea secundară, sistemul poate folosi eficient transferul pe bandă largă, deoarece se scrie întreaga pagină, și nu doar o mică parte a acesteia;

2.6 Concluzii

Așadar, memoria virtuală e un mecanism care creează o nouă abstractizare a memoriei, pentru a soluționa eficient problema supraîncărcării memoriei principale folosite de un sistem de operare. În ultimii ani, dat fiind că procesele folosesc din ce în ce mai multă putere computațională, a devenit necesară dezvoltarea unui astfel de spațiu virtual de stocare, care să-l extindă pe cel disponibil în realitate. Pe de altă parte, virtualizarea oferă un mecanism de protecție împotriva virușilor care încearcă să corupă zone de memorie, dar și în cadrul unui sistem, de protecție a unui program de altul, separându-le zonele accesibile pentru procesele lor.

3. Analiză

3.1 Funcționalități

Proiectul implementează funcționalități care permit emularea unui sistem cu memorie virtuală la o scară redusă față de sistemele reale. Astfel, sunt posibile următoarele operații:

- Scriere de date în memorie, conform unei adrese;
- Citire de date din memorie, conform unei adrese;
- Translatarea adresei virtuale folosind o tabelă de translatare (page table) și un buffer TLB care funcționează ca un cache pentru acces mai rapid la informațiile de corespondență dintre o pagină din memoria principală și una din memoria secundară;
- Detectarea unei excepții (page fault) și declanșarea mecanismului de tratare a acesteia;
- Înlocuirea unei pagini care nu a fost accesată de mult timp cu cea care se încearcă a fi accesată și nu are o intrare corespunzătoare validă în page table (tratarea page fault-ului);
- Plasarea unei intrări noi în TLB pe următoarea poziție liberă (memorie asociativă);

3.2 Use-case-uri

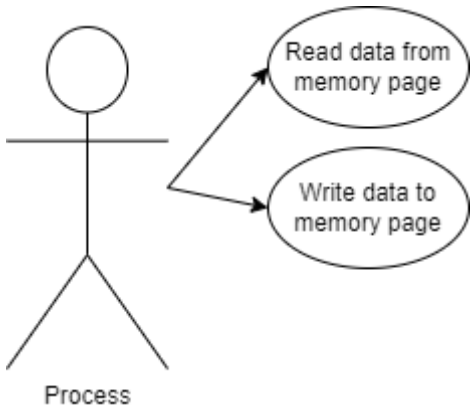

Figura 5. Diagrama de use-case pentru o memorie virtuală

Pentru a fi cât mai aproape de implementarea dintr-un sistem de operare real, un proces ar trebui să poată accesa zona de memorie care îi este alocată pentru scrieri sau pentru citiri de date.

Se consideră că procesul trimite către unitatea de management a memoriei o adresă care trebuie translatată și blocul de date pentru scriere sau locația unde s-ar plasa datele citite (e.g: o variabilă locală de pe stiva procesului).

3.3 Mecanisme și algoritmi

Cel mai utilizat (dar nu și cel mai eficient) algoritm pentru tratarea situațiilor de page fault este LRU (Least Recently Used) page replacement. Ideea acestui algoritm este să se memoreze un istoric al utilizării paginilor și în momentul în care are loc un miss în page table, să se înlocuiască cea mai veche pagină accesată cu o alta, nouă.

În limbajul VHDL, LRU s-ar putea implementa cu un counter pentru fiecare pagină. Dacă nu mai există niciun loc liber în memoria principală, se verifică bitul de ‘dirty’ al paginii cu cea mai mică valoare numărată pentru a încărca posibilele modificări în memoria secundară, apoi se înlocuiește intrarea cu cea pentru pagina nouă. Dacă spațiu liber mai există, bitul de validitate devine ‘1’ pe acea poziție din page table, se adaugă datele noi despre pagina reală și se incrementează counter-ul.

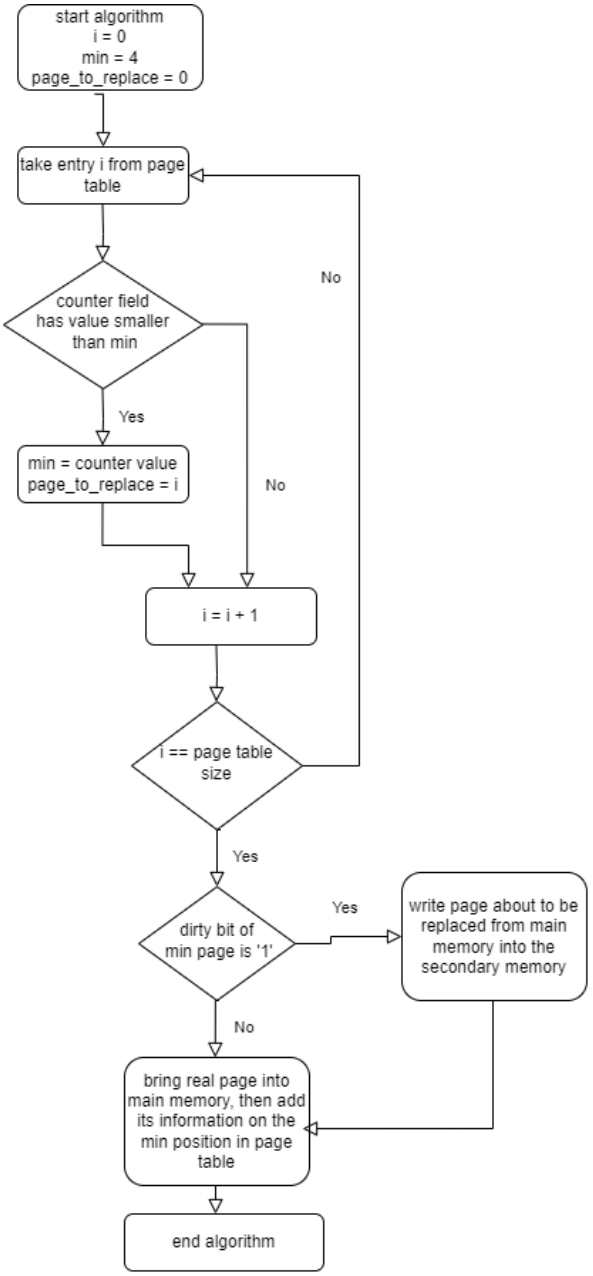

Figura 6. Diagrama de stări pentru algoritmul de înlocuire LRU

Mecanismul principal utilizat în proiect este cel de depistare și de tratare a excepțiilor. Un astfel de eveniment are loc atunci când o pagină virtuală nu are o intrare validă în page table, ceea ce înseamnă că datele nu se regăsesc în memoria principală și trebuie aduse de pe disk.

Excepția se declanșează odată cu obținerea semnalului de miss cu valoarea ’1’ de la unitatea de management a memoriei virtuale și presupune:

1) Salvarea PC-ului (numărător de program) care a declanșat excepția într-un registru EPC și a codului de excepție într-un registru de cauză;
2) Localizarea paginii care a cauzat întreruperea în memoria secundară;
3) Aducerea paginii din memoria secundară în memoria principală;
4) Actualizarea page table-ului și a TLB-ului cu translația noii pagini (algoritmul LRU);
5) Reîncărcarea numărătorului de program cu valoarea din EPC, curățarea registrului de cauză;
6) Reluarea instrucțiunii, care nu mai declanșează page fault;

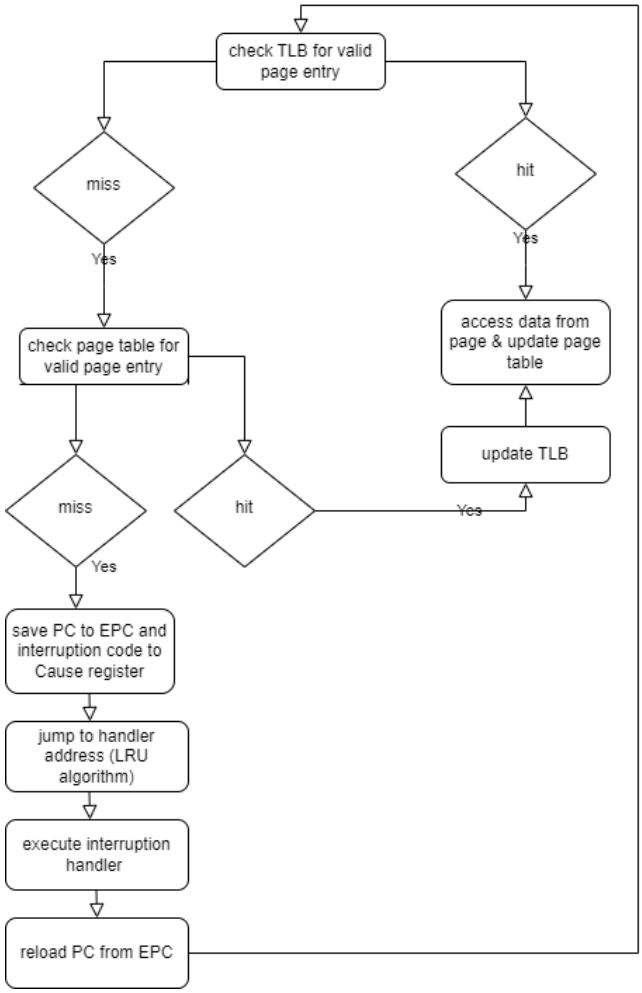

Figura 7. Diagrama de stări pentru page fault handling

3.4 Modul de codificare a datelor

3.4.1 Adresele folosite pentru accesarea datelor

au următorul format:

a) Pentru memoria principală (pentru implementare VHDL):

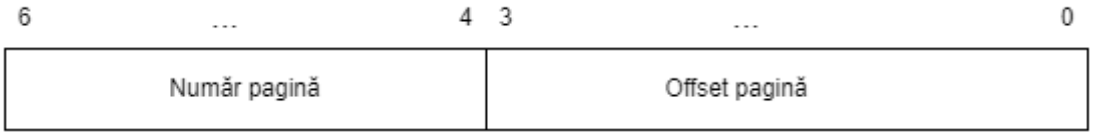

Figura 8. Codificarea adreselor din memoria principală

Numărul paginii se codifică pe 3 biți, ceea ce permite încărcarea a până la 8 pagini, iar offset-ul din cadrul paginii pe 4 biți, dimensiunea aleasă pentru o pagină scaled-down fiind de 16 de octeți (= 128 de biți, fiecare byte e adresabil individual).

b) Pentru memoria secundară (pentru implementare VHDL):

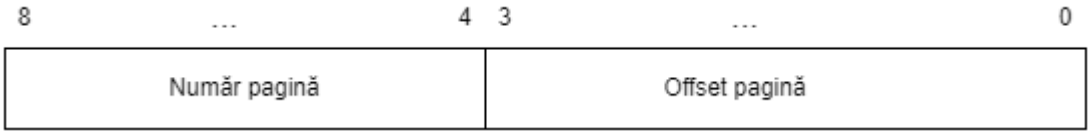

Figura 9. Codificarea adreselor din memoria virtuală

Numărul paginii virtuale are nevoie de mai mulți biți de codificare, deoarece în memoria virtuală există mai multe pagini, în cazul dat 32. Numărul de biți pentru offset rămâne însă același, paginile având aceeași dimensiune cu cele din memoria principală.

3.4.2 Modul de codificare a datelor despre date

a) Formatul unei intrări în page table:

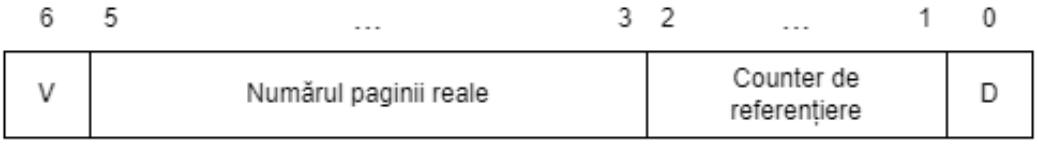

Figura 10. Codificarea intrărilor în page table

Intrarea pentru fiecare pagină virtuală din page table conține un bit de validitate (bitul 6), 3 biți pentru codificarea numărului paginii reale din memoria principală, un counter intern pe 2 biți (cu saturare) pentru a reține un istoric al accesului la fiecare și un bit ‘dirty’, care va comanda la înlocuire scrierea tuturor modificărilor făcute pe parcurs în memoria secundară (mecanism de scriere write-back).

b) Formatul unei intrări în Translation-Lookaside Buffer:

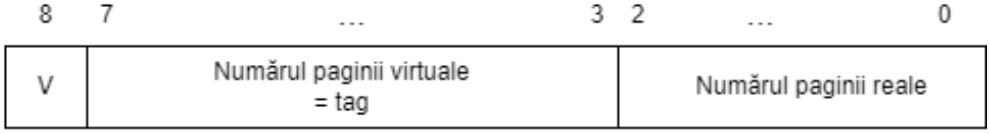

Figura 11. Codificarea intrărilor în TLB

TLB este un cache pentru page table, astfel acesta are o dimensiune mai mică și reține doar cele mai recente translatări efectuate. O intrare din TLB conține un bit de validitate pentru respectiva translatare, numărul paginii virtuale care funcționează ca și un câmp de tag pentru o memorie cache obișnuită, și numărul paginii reale corespondente numărului paginii virtuale primite în codul instrucțiunii curente.

4. Proiectare/Design

4.1 Arhitectura proiectului

Proiectul este definit și implementat în limbajul de descriere hardware VHDL, care presupune:

1) Entități pentru declararea componentelor și porturilor de intrare/ieșire pentru acestea;
2) Arhitecturi pentru fiecare componentă care să implementeze comportamentul așteptat;
3) Semnale care să asigure comunicarea între componente (folosind semnale, valorile de pe port-urile out ale unei componente pot deveni valorile pentru port-urile in ale altei componente);
4) Procese concurente, care se execută în același timp și prin care se descrie comportamentul componentelor în arhitecturi;
5) Instrucțiuni secvențiale, care se execută în interiorul proceselor și permit folosirea variabilelor și sintaxei clasice din alte limbaje de programare, precum bucle for, if-uri și else-uri, case-uri;

Mediul de dezvoltare utilizat este Vivado 2020.1, ceea ce aduce anumite beneficii, precum verificarea dinamică a sintaxei, importarea unor biblioteci de funcții predefinite (în proiect apar IEEE.STD_LOGIC_1164, IEEE.STD_LOGIC_UNSIGNED și IEEE.STD_LOGIC_ARITH, care permit inclusiv adunarea între doi vectori de biți), sinteza, implementarea și depanarea proiectului, generarea unui bitstream pentru rularea pe o placă FPGA și generarea unor scheme RTL sau netlist.

Componentele necesare (redate în următorul subcapitol) pentru funcționarea și testarea unității de management pentru memoria virtuală se descriu separat, fiecare într-un fișier propriu, apoi sunt instanțiate într-o componentă globală (top module), unde se declară și semnalele intermediare, pasate de la o entitate la alta folosind ‘port map’.

4.2 Componentele utilizate

- **Memoria Principală (RAM):** reprezintă o memorie cu semnal de enable, citire asincronă și scriere sincronă; dimensiunile alese (pentru a nu supraîncărca resursele plăcuței) sunt – 8 pagini x 128 de biți = 1KB;

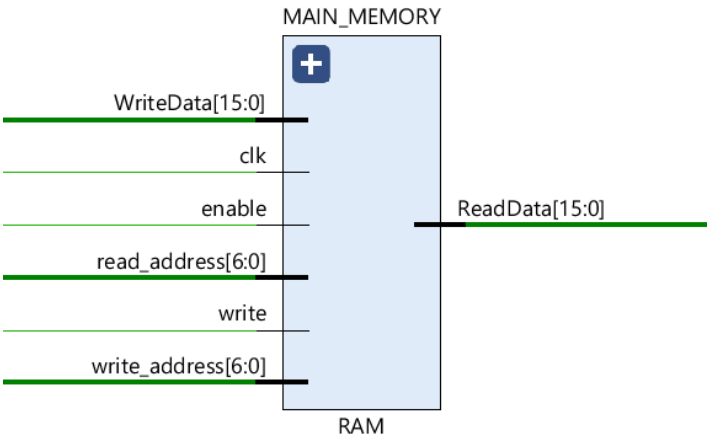

Figura 12. Componenta pentru memoria principală

- **Memoria Virtuală:** la fel o memorie cu semnal de enable, citire asincronă și scriere sincronă; dimensiunile sunt însă mai mari decât ale memoriei principale – 32 de pagini x 128 de biți = 4KB;

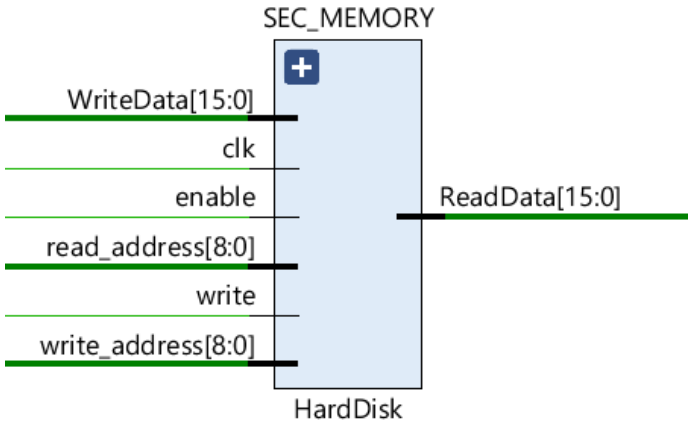

Figura 13. Componenta pentru memoria secundară

- **Unitatea de management a memoriei virtuale:** o componentă care primește ca input instrucțiunea de executat și care conține un buffer TLB și un page table, pentru memorarea translatărilor de la pagină virtuală la pagină din memoria principală; unitatea de management este responsabilă de efectuarea operațiilor de scriere și citire, de încărcarea unei pagini lipsă în memoria principală, de generarea diverselor semnale de comandă, și de activarea unei întreruperi, dacă nu se poate găsi o anumită pagină în memoria reală;

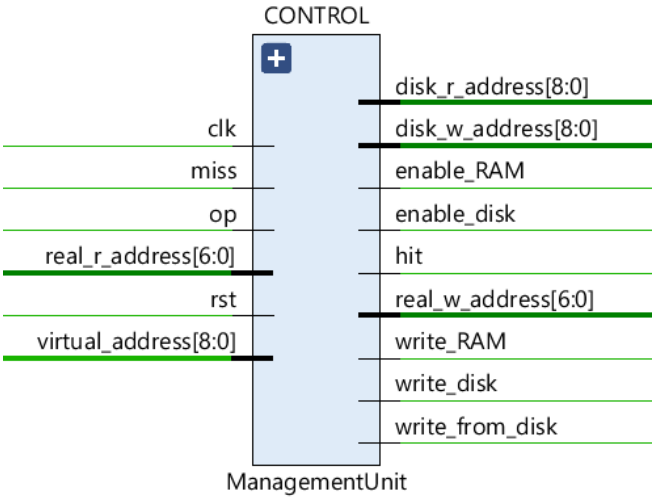

Figura 14. Unitatea de management a memoriei virtuale

- **Afișor cu 7 segmente:** pentru a putea vizualiza datele din memorie; conține în interior un decodificator care mapează valorile hezadecimale la un vector de 7 elemente, care reprezintă segmentele afișorului;

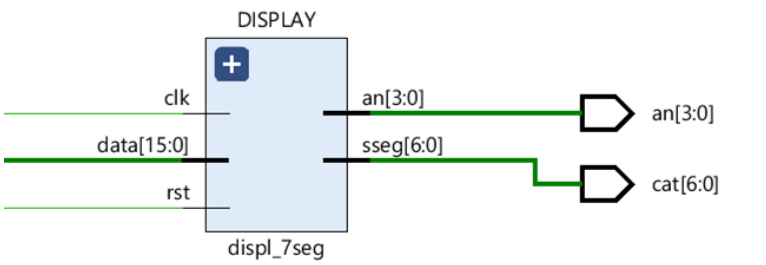

Figura 15. Componenta de afișare a datelor pe 7 segmente

4.3 Model de date

Am utilizat pentru descrierea datelor tipuri deja existente în limbajul VHDL, în biblioteca STD_LOGIC_1164, și anume STD_LOGIC pentru semnalele de 1 bit și STD_LOGIC_VECTOR pentru semnale pe mai mulți biți. De asemenea, am folosit tipul integer (cu interval de valori limitat cu 'range') pentru simplitate în momentul indexării page table-ului sau TLB-ului.

Memoriile au fost declarate ca tipuri de date array, indexate de la 0, cu valori de tip STD_LOGIC_VECTOR.

De asemenea, am definit 2 tipuri noi de date, pentru stările care apar la tratarea excepțiilor și în cadrul algoritmului de înlocuire al paginilor.

Pentru anumite operații de conversie (de exemplu de la integer la std_logic_vector sau invers) și aritmetice (adunare între vector de biți și un întreg) a fost nevoie de includerea altor 2 biblioteci: STD_LOGIC_ARITH și STD_LOGIC_UNSIGNED.

Exemple de declarații:

a) Memoria principală (idem pentru cea virtuală, doar alte dimensiuni):

```
type page is array (0 to 2**4-1) of STD_LOGIC_VECTOR(7 downto 0);
type mem is array (0 to 7) of page;
```

Figura 16. Declarația tipurilor de date pentru memorii

b) Page table și TLB:

```
type page_table_type is array(0 to 31) of STD_LOGIC_VECTOR(6 downto 0);
signal page_table: page_table_type := ("1000000", others => (others => '0'));
type lookaside_buffer is array(0 to 3) of STD_LOGIC_VECTOR(8 downto 0);
signal TLB: lookaside_buffer := ("100000000", others => (others => '0'));
```

Figura 17. Declarația tipurilor de date pentru page table și TLB

c) Câteva semnale auxiliare din unitatea de management:s

```
-- iterators
variable i : integer range 0 to 127 := 0;
variable j : integer range 0 to 15 := 0;
-- variables for finding page to replace
variable min_count: STD_LOGIC_VECTOR(2 downto 0) := "111";
variable min_entry: integer;
variable min_real_page: STD_LOGIC_VECTOR(3 downto 0) := "0000";
variable entry_count: STD_LOGIC_VECTOR(2 downto 0);

variable hit_var: STD_LOGIC;
```

Figura 18. Declarații de semnale auxiliare din unitatea de management

d) Tipuri de date definite pentru stările algoritmului

```
type LRU_states is (START_ALG, END_ALG, TAKE_ENTRY, CHECK_MIN, INC_I, IS_DIRTY, BRING_PAGE);
signal LRU_s : LRU_states := START_ALG;

type PGH_states is (RELOAD, CHECK_TLB, CHECK_PT, SAVE_CURRENT_EXEC_HANDLER, EXEC_LRU, UPDATE_TLB, ACCESS_DATA);
signal PGH_s : PGH_states := CHECK_TLB;
```

Figura 19. Declarații de tipuri noi

4.4 Interacțiunea dintre componente

Interacțiunea dintre componente se efectuează prin intermediul semnalelor, declarate în arhitectura entității top module. Legături există între toate modulele, pentru a permite funcționarea proiectului conform cu specificațiile.

Cele mai importante legături se creează însă între cele 2 memorii, principală și secundară, și unitatea de gestionare a acestora.

Semnalele de control enable_RAM, enable_disk, write_RAM și write_disk sunt necesare pentru efectuarea operațiilor de scriere a datelor din paginile de memorie, cât și pentru aplicarea algoritmului de înlocuire a unei pagini.

Virtual_address este un semnal care se obține din codul instrucțiunii și care reprezintă adresa care se dorește accesată și care va fi translatată în real_r_address sau real_w_address în dependență de operație, folosind page table-ul intern din ManagementUnit. Semnalul real_address ajunge apoi la memoria principală, de unde se citesc sau scriu datele necesare procesului care a generat instrucțiunea.

Semnalele hit și miss sunt determinate de succesul sau eșecul de a fi găsit pagina în memoria principală. Miss va declanșa o întrerupere, care va fi tratată prin aducerea paginii necesare din memoria secundară.

Save_PC_to_EPC determină încărcarea în registrul EPC a counter-ului de program curent, iar rollback_PC este un semnal auxiliar care provoacă revenirea la PC-ul anterior, care a generat întreruperea, din EPC.

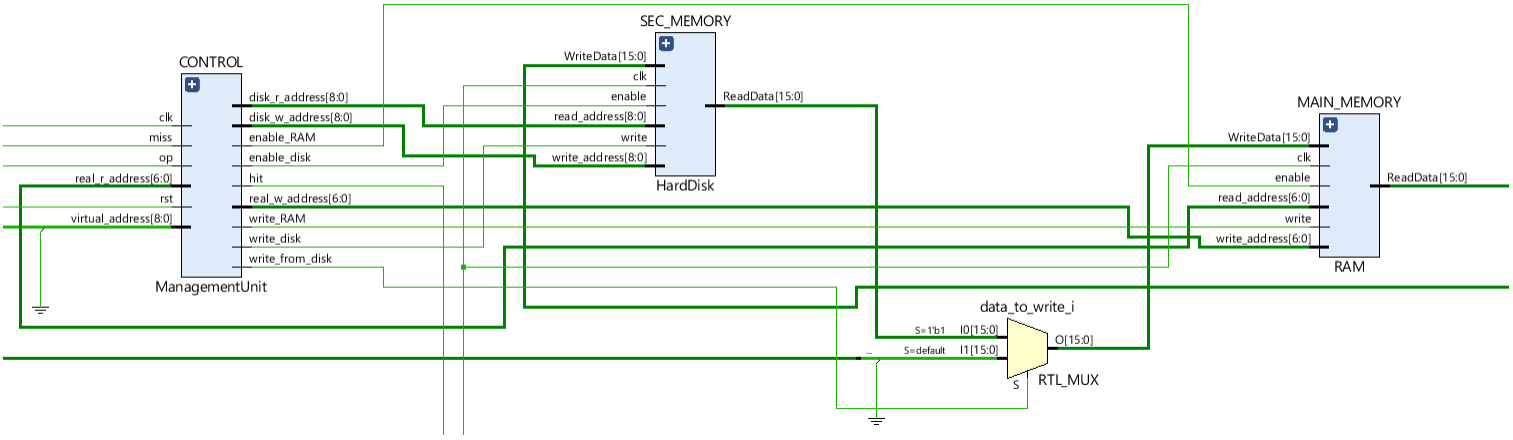

Figura 20. Comunicarea între cele 2 memorii și unitatea de management

Write_from_disk devine necesar ca selecție pentru un multiplexor în momentul scrierii valorilor în memoria principală, deoarece există 2 surse de astfel de date: hard disk-ul, la încărcarea unei pagini, și date externe, la rularea unei instrucțiuni de scriere. Pentru ’1’, se vor scrie date din memoria secundară, iar pentru ‘0’, valorile din codul instrucțiunii.

Memoriile comunică între ele prin datele pe care le transferă una de la alta, portul WriteData de la HardDisk este legat cu portul ReadData de la RAM și invers. Intrarea WriteData a memoriei principale e determinată de multiplexor și semnalul write_from_disk, iar adresele pentru ambele sunt calculate în unitatea de management (semnalele disk_r_address, disk_w_address și real_r_address, real_w_address).

5. Implementare

Memoriile primesc o adresă de citire ca intrare, după care pe ieșirea de date apar datele păstrate la acea poziție din pagină. Această operație poate fi descrisă în afara unui proces, unde pageNumRead sunt partea de adresă care codifică numărul paginii, iar pageOffsetRead sunt biții care reprezintă octetul de la care începe citirea:

```
-- citire asincrona din memorie
ReadData(7 downto 0) <= memory(conv_integer(pageNumRead))(conv_integer(pageOffsetRead));
ReadData(15 downto 8) <= memory(conv_integer(pageNumRead))(conv_integer(pageOffsetRead)+1);
```

Figura 21. Citirea datelor din memorie

Procesul de scriere în memorie necesită o verificare în plus, pentru semnalul write care determină dacă datele de pe portul de intrare sunt valide sau nu pentru a fi adăugate în memorie, și are loc pe front crescător de ceas.

```
WRITE_PROC: process(clk, write, enable, WriteData, pageNum, pageOffset)
begin
    if RISING_EDGE(clk) then
        if (write = '1' and enable = '1') then
            memory(conv_integer(pageNum))(conv_integer(pageOffset)) <= WriteData(7 downto 0);
            memory(conv_integer(pageNum))(conv_integer(pageOffset)+1) <= WriteData(15 downto 8);
        end if;
    end if;
end process;
```

Figura 22. Procesul de scriere în memorie

Partea de gestionare a memoriei virtuale a fost implementată ca o secvență de stări, conform cu diagramele și tipurile de date definite în capitolul 3. Mașina de stări reprezintă în limbajul VHDL o instrucțiune de case, unde fiecare ramură efectuează un pas diferit.

```
if rising_edge(clk) then
case PGH_s is
    when CHECK_TLB =>...

     when CHECK_PT =>...

     when ACCESS_DATA =>...

     when SAVE_CURRENT_EXEC_HANDLER =>...

     when EXEC_LRU =>...

     when UPDATE_TLB => ...

     when RELOAD =>          ...
end case;
```

Figura 23. Implementarea diagramei de stări pentru mecanismul de tratare a excepțiilor

Algoritmul LRU care se aplică în cazul în care nu mai sunt poziții libere în memoria principală se execută ca o rutină de întrerupere, de aceea e integrat în starea EXEC_LRU, care asigură rezolvarea situației de page fault.

```
when EXEC_LRU =>
   -- begin the LRU algorithm in order to find the page to replace
   case LRU_s is
       when START_ALG =>...

       when TAKE_ENTRY =>...

       when CHECK_MIN => ...

       when INC_I =>...

       when IS_DIRTY =>...

       when BRING_PAGE =>...

       when END_ALG =>...
   end case;
```

Figura 24. Implementarea diagramei de stări pentru algoritmul de înlocuire a paginii

Operația dorită, scriere sau citire în memoria principală, are loc în starea de ACCESS_DATA, unde unitatea de management generează semnalele necesare pentru activarea memoriei RAM și obținerea datelor, translatând totodată adresa virtuală primită într-o adresă reală:

```
-- !! only if the address or operation has changed => otherwise, the system stays in the same state and
-- the counter shouldn't be incremented
if virtual_change /= virtual_address then
    page_table(virt_page_number)(2 downto 1) <= page_table(virt_page_number)(2 downto 1) + 1;
    -- decrement counter for all other pages in main memory
    for i in 0 to 31 loop
        if i /= virt_page_number then
            if page_table(i)(2 downto 1) /= "00" then
                page_table(i)(2 downto 1) <= page_table(i)(2 downto 1) - 1;
            end if;
        end if;
    end loop;
    virtual_change <= virtual_address;
elsif op_change /= op then
    page_table(virt_page_number)(2 downto 1) <= page_table(virt_page_number)(2 downto 1) + 1;
    -- decrement counter for all other pages in main memory
     for i in 0 to 31 loop...
    op_change <= op;
end if;

ram_r := page_table(virt_page_number)(5 downto 3) & virtual_address(3 downto 0);
if op = '0' then
    enable_RAM <= '0';
else
    enable_RAM <= '1';
    write_RAM <= '1';
    ram_w := page_table(virt_page_number)(5 downto 3) & virtual_address(3 downto 0);
    -- modify dirty bit in case of write
    page_table(virt_page_number)(0) <= '1';
end if;
```

Figura 25. Generarea semnalelor de control pentru scriere/citire din memoria principală

6. Testare/Experimente

Am testat fiecare componentă în parte, iar apoi proiectul în totalitate folosind testbench-uri, asignând diverse valori pentru semnale și pentru intrările de date. Cu ajutorul formelor de undă generate de simularea în mediul Vivado, am putut corecta comportamentul fiecărei părți componente a proiectului.

```
clk <= not clk after 5ns;
enable <= '1';
write <= '0', '1' after 15ns, '0' after 30ns;
WriteData <= x"0050";
address <= B"000_0000",
        B"001_00000" after 5ns,
        B"001_0001" after 10ns,
        B"000_0101" after 15ns;


TEST: RAM port map(clk => clk,
                   enable => enable,
                   write => write,
                   read_address => address,
                   write_address => address,
                   WriteData => WriteData,
                   ReadData => ReadData);
```

Figura 26. Valori de testare pentru memoria principală

```
clk <= not clk after 5ns;
enable <= '1';
write <= '0', '1' after 15ns, '0' after 30ns;
WriteData <= x"000F";
address <= B"00000_0000",
        B"00011_0000" after 5ns,
        B"00001_0100" after 10ns,
        B"00001_0110" after 15ns;


TEST: HardDisk port map(clk => clk,
                        enable => enable,
                        write => write,
                        read_address => address,
                        write_address => address,
                        WriteData => WriteData,
                        ReadData => ReadData);
```

Figura 27. Valori de testare pentru memoria secundară

Testele pentru fiecare memorie au inclus 4 adrese transmise la momente diferite de timp, și atât efectuarea unor operații de citire, cât și de scriere în memorie.

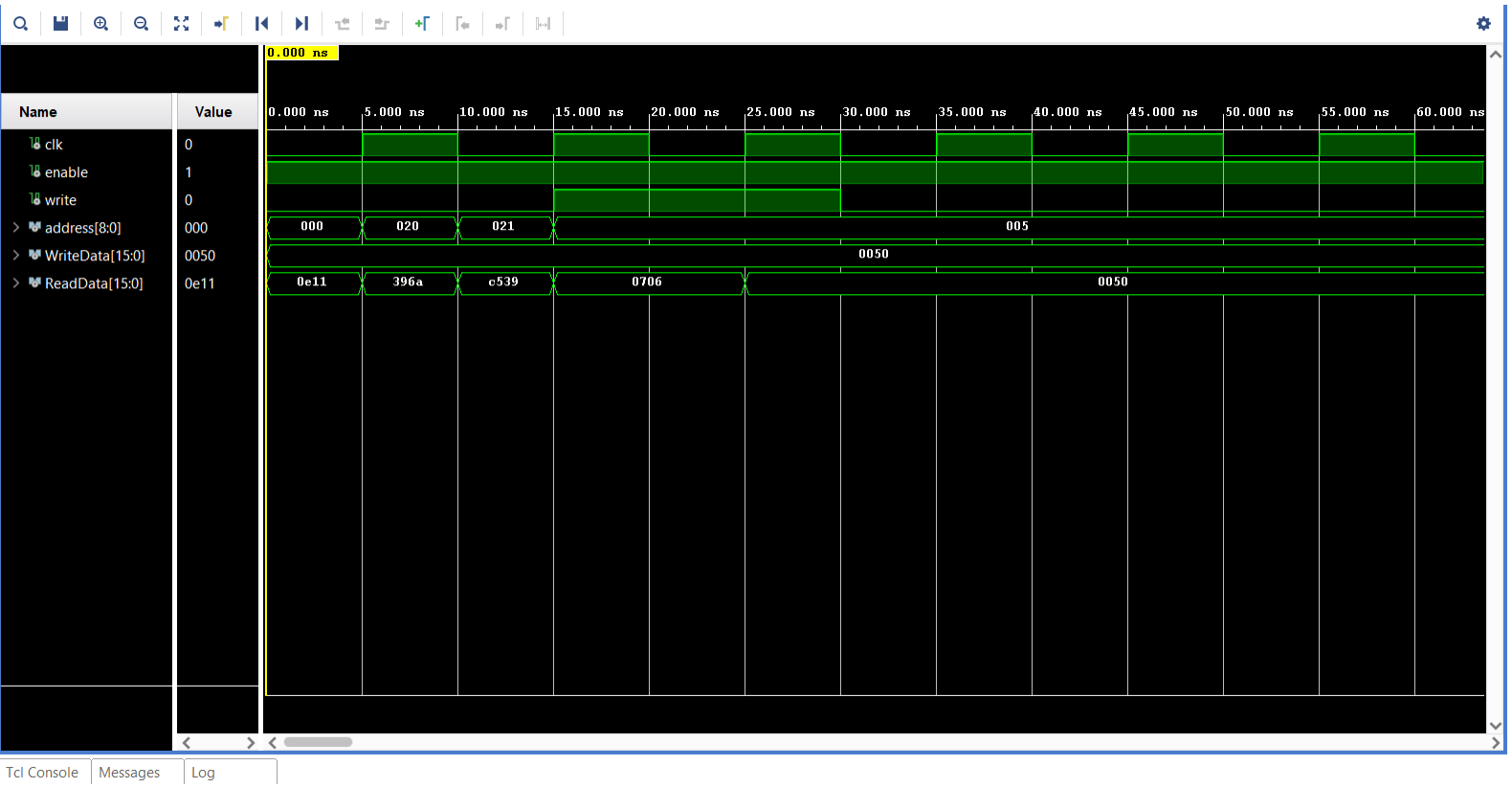

Figura 28. Detaliu din simularea unei memorii

Unitatea de management a fost testată pe 3 cazuri de adrese virtuale, dintre care 2 pentru efectuarea unei operații de citire și una pentru efectuarea operației de scriere.

```
clk <= not clk after 5ns;
virtual_address <= B"00000_0000", B"00011_0010" after 1000ns, B"00010_0011" after 2000ns;
op <= '0', '1' after 3000ns;
r <= '1', '0' after 5ns;

TEST: ManagementUnit port map (clk => clk,
                               rst => r,
                               virtual_address => virtual_address,
                               op => op,
                               real_r_address => real_r_address,
                               real_w_address => real_w_address,
                               disk_r_address => disk_r_address,
                               disk_w_address => disk_w_address,
                               enable_RAM => enable_RAM,
                               enable_disk => enable_disk,
                               write_RAM => write_RAM,
                               write_disk => write_disk,
                               hit => hit,
                               miss => miss,
                               save_PC_to_EPC => save_PC,
                               rollback_PC => rollback_PC,
                               write_from_disk => write_from_disk);
```

Figura 29. Valori de testare pentru unitatea de management

Rularea întregului mecanism de tratare al excepției, inclusiv cu aplicarea algoritmului LRU și revenirea la instrucțiunea anterioară durează 83 de cicluri de ceas!!!

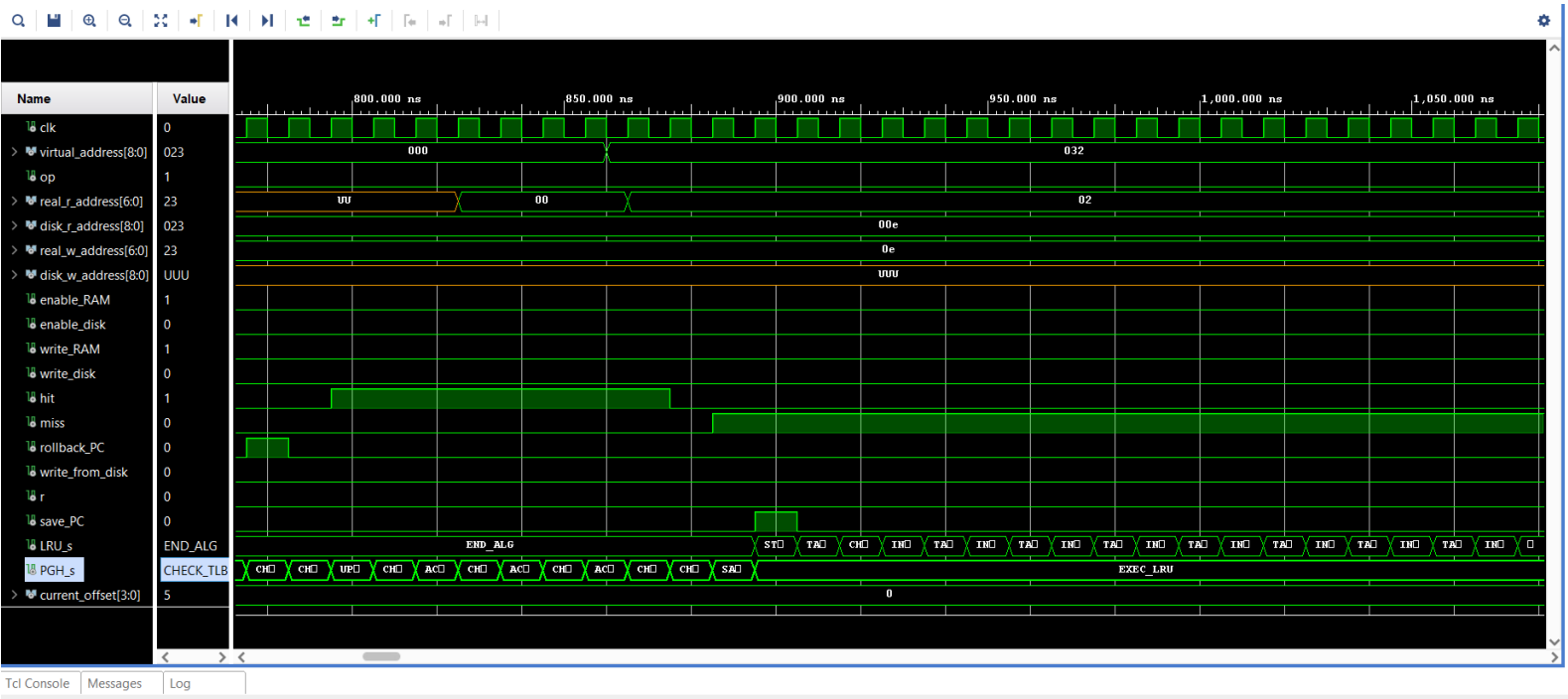

Figura 30. Detaliu din simularea unității de management

Se poate observa modul în care unitatea de management trece dintr-o stare în alta conform diagramelor definite mai sus, rezolvând astfel eroarea care apare la încercarea de a se accesa o pagină care încă nu a fost adusă în memoria principală.

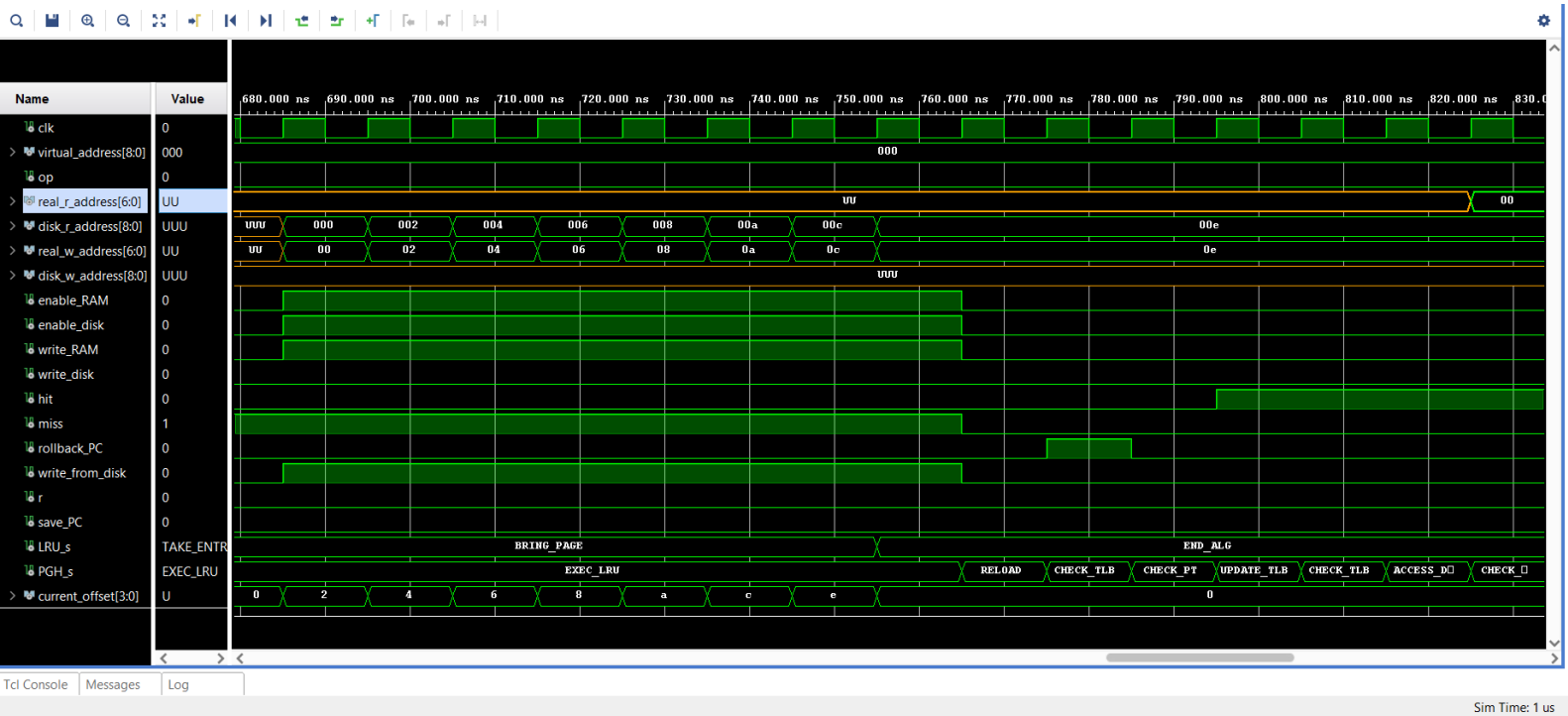

Figura 31. Detaliu din simulare – aducerea paginii în memoria principală

Modulul în cadrul căruia se face interconectarea memoriilor și a unității de management, a fost de asemenea testat pe 3 cazuri de adrese virtuale:

```
clk <= not clk after 5ns;
rst <= '0';
-- input(15) = op, input(14 downto 7) = data,
-- input(6 downto 2) = virtual page number, input(1 downto 0) = virtual page offset
input <= B"0_00000000_00011_00",
         B"1_01101100_00001_10" after 3000ns,
         B"0_00001111_00001_00" after 6000ns;


TEST: Full port map (clk => clk,
                     rst => rst,
                     input => input,
                     an => an,
                     cat => cat,
                     hit => hit,
                     miss => miss);
```

Figura 32. Valori de testare pentru modulul complet

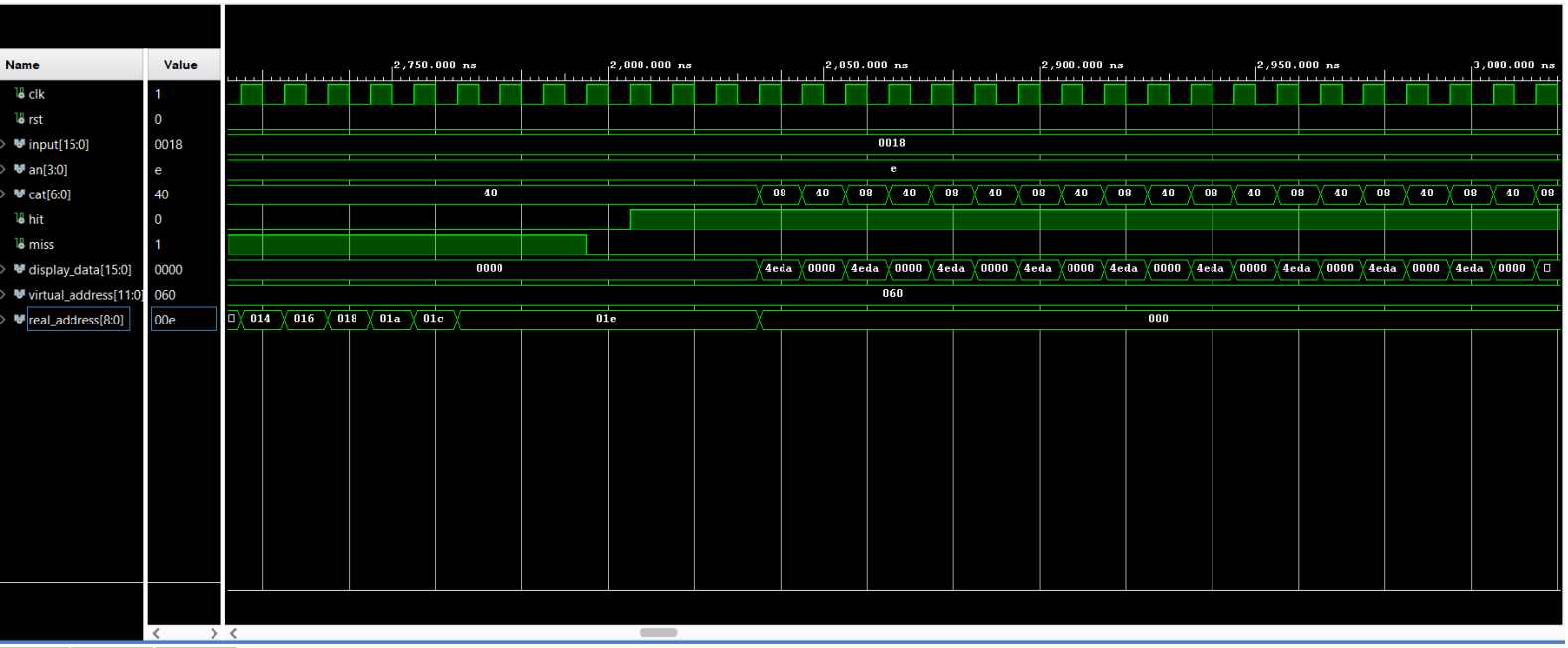

Figura 33. Detaliu din simularea modulului complet – citirea datelor după aducerea paginii în memoria principală

În figura 34, se observă apariția semnalului de miss în momentul în care pagina specificată de adresa virtuală nu se află în memoria RAM, iar în figura 35, are loc o scriere în pagina adusă din stocarea secundară (în starea de dinaintea citirii se citește valoarea ”3599”, apoi în următorul ciclu de ceas, valoarea de la acea adresă devine ”006c”).

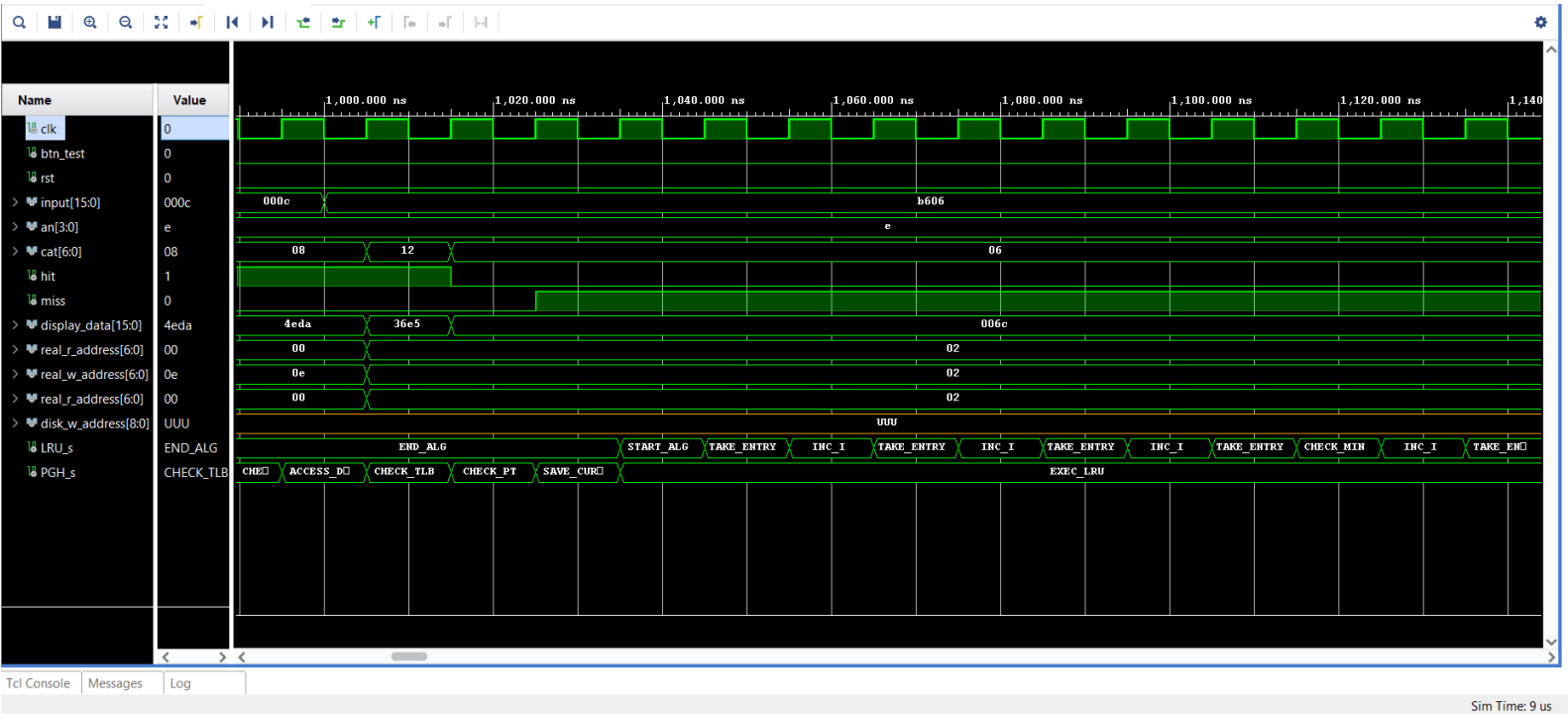

Figura 34. Detaliu din simularea modulului complet – declanșarea unui page fault

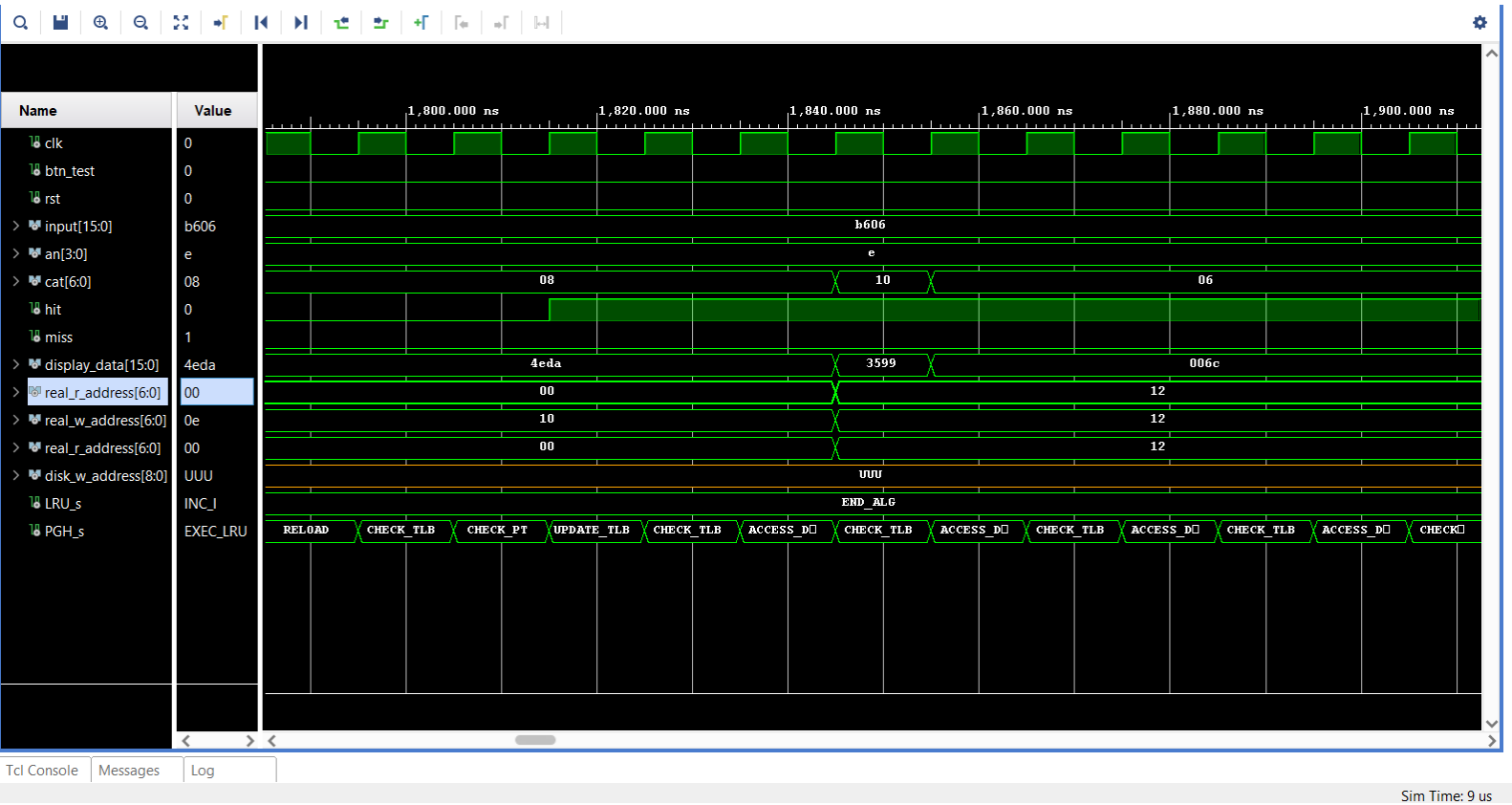

Figura 35. Detaliu din simularea modulului complet – scrierea într-o pagină din memoria principală

La final, pentru a verifica funcționarea algoritmului de înlocuire, proiectul a fost testat și pe placa de dezvoltarea FPGA Basys 3.

7. Concluzii

Așadar, implementarea unei memorii virtuale și a unei unități de gestionare a acesteia este un pas important în asigurarea unei utilizări mai eficiente a memoriei, a izolării și protejării datelor unui proces de alte procese care accesează aceeași zonă de memorie, a scalabilității, proprietate necesară în contextul aplicațiilor complexe care consumă multe resurse și a fiabilității sistemului, în stare să detecteze și să rezolve situațiile de eroare care pot apărea (în cazul dat, page fault). Proiectul dat reușește să îndeplinească această sarcină.

Am întâlnit și anumite dificultăți în implementarea proiectului, mai ales la partea de implementare, multe resurse oferind explicații doar la nivel pur teoretic despre mecanismele de tratare a întreruperilor și despre algoritmii de înlocuire a paginilor din memoria principală.

O altă problemă a apărut la partea de testare pe placă, din cauza resurselor limitate (butoane, switch-uri, celule FPGA etc.) de pe aceasta. Un compromis a fost folosirea celor 16 switch-uri de pe Basys 3 în felul următor: ultimii 2 biții -> offset-ul adresei, biții 14-7 -> octetul cel mai puțin semnificativ care se introduce la scriere în memorie, biții 6-2 -> numărul paginii din adresa virtuală, iar bitul 15 -> operația de efectuat.

În concluzie, proiectul simulează cu succes modul în care memoriile virtuale îmbunătățesc funcționalitatea, siguranța și performanța generală a unui sistem de operare. Acest mecanism reprezintă baza pentru un mediu computațional mai robust și mai adaptabil, capabil să facă față obstacolelor cauzate de aplicațiile software moderne.

8. Bibliografie

[1] David. A. Patterson & John L. Hennessy, “Computer Organization and Design: RISC-V Edition”, publicată de Morgan Kaufmann Publishers, anul 2018

[2] Andrew S. Tanenbaum & Herbert Bos, “Modern Operating Systems”, ediția IV, publicată de Pearson Education Limited, anul 2015

[3] Florin I. Oniga, cursuri de Arhitectura Calculatoarelor, anul 2022

[4] Erika Dennis, MIPS I/O and Interrupt course, Florida State University

[5] LRU Page Replacement Algorithm – Codingninjas, https://www.codingninjas.com/studio/library/lru-page-replacement-algorithm

[6] Dominic Sweetman, “See MIPS Run”, publicată de Algorithmics Ltd., anul 1999

9. Index de imagini

Anexa 1. Schema completă a proiectului