

FACULTATEA DE INGINERIE ELECTRICĂ ȘI
ȘTIINȚA CALCULATOARELOR

ing. Ionel ZAGAN

TEZĂ de DOCTORAT

-rezumat-

**Contribuții la dezvoltarea sistemelor de operare în timp real cu
funcții implementate în hardware**

Conducător științific:

Prof. univ. dr. ing. **Vasile Gheorghiță GĂITAN**

Suceava, 2017

Acknowledgments

Această lucrare a beneficiat de suport logistic și financiar prin proiectul “*Centru integrat de cercetare, dezvoltare și inovare pentru Materiale Avansate, Nanotehnologii și Sisteme Distribuite de fabricație și control (MANSiD)*”, contract nr. 671/09.04.2015, cofinanțat din Fondul European de Dezvoltare Regională (FEDR) prin Programul Operațional Sectorial “Creșterea Competitivității Economice”.

Această lucrare a beneficiat de suport financiar prin proiectul “*Quality European Doctorate - EURODOC*”, contract nr. POSDRU/187/1.5/S/155450, cofinanțat din Fondul Social European prin Programul Operațional Sectorial Dezvoltarea Resurselor Umane 2007-2013.



Universitatea
Ștefan cel Mare
Suceava

DECIZIA D 65 / 22.05.2017

privind constituirea comisiei pentru susținerea publică a unei teze de doctorat

În conformitate cu prevederile H.G. nr. 26/12.01.2017 privind organizarea și funcționarea Ministerului Educației Naționale, Anexa 3, punctul 39, prin care se instituționalizează Universitatea Ștefan cel Mare din Suceava;

În conformitate cu HG 681 din 29.06.2011 - Codul Studiilor universitare de doctorat;
Având în vedere aprobarea conducerii Universității;

Rectorul Universității "Ștefan cel Mare" din Suceava emite prezenta decizie:

Art. 1 - Se numește comisia de doctorat pentru susținerea publică a tezei de doctorat în data de **22.06.2017**, cu titlul: „**CONTRIBUȚII LA DEZVOLTAREA SISTEMELOR DE OPERARE ÎN TIMP REAL CU FUNCȚII IMPLEMENTATE ÎN HARDWARE**”,

elaborată de **ZAGAN A. Ionel**,
înmatriculat la data de 1.10.2014,
forma de învățământ: cu frecvență, fără taxă,
DOMENIUL: **Calculatoare și tehnologia informației**

PREȘEDINTE:

- Prof. univ. dr. ing. Adrian GRAUR, reprezentant IOSUD, Universitatea "Ștefan cel Mare" din Suceava;

CONDUCĂTOR ȘTIINȚIFIC:

- Prof. univ. dr. ing. Vasile Gheorghiță GĂITAN, Universitatea "Ștefan cel Mare" din Suceava;

REFERENȚI:

1. Prof. univ. dr. ing. Nicolae ȚĂPUȘ, Universitatea Politehnica din București;
2. Prof. univ. dr. Sergiu NEDEVSCI, Universitatea Tehnică din Cluj Napoca;
3. Prof. univ. dr. ing. Corneliu Octavian TURCU, Universitatea "Ștefan cel Mare" din Suceava.

Art. 2. – Direcția Economică respectiv serviciul Secretariat - doctorat vor duce la îndeplinire dispozițiile prezentei decizii.

RECTOR,
Prof. univ. dr. ing. Valentin POPA



C.Sk./C.Sk. / 1 ex.

SECRETAR ȘEF UNIV.,
Jurist Maria MUSCĂ

Cuprins

1. Introducere	1
2. Stadiul actual al SOTR cu funcții implementate în hardware.....	8
3. Analiza algoritmilor de planificare folosiți în STR și a funcțiilor SOTR.....	10
3.1. Algoritmi de planificare	12
3.2. Supracontrolul sistemelor de operare în timp real	13
3.3. Structura generală a nucleului de timp real și funcțiile acestuia	13
4. Descrierea procesorului nMPRA și a resurselor software/hardware	15
5. Contribuții privind implementarea arhitecturii nMPRA și nHSE în FPGA	28
5.1. Descrierea modulelor procesorului nMPRA implementate în Verilog.....	28
5.2. Implementarea și testarea procesorului nMPRA.....	30
5.2.1. Validarea căii de date și a unităților funcționale.....	31
5.2.2. Planificatorul hardware integrat - nHSE.....	36
5.2.3. Validarea instrucțiunilor dedicate planificatorului nHSE.....	39
5.2.4. Mecanismul de sincronizare prin mutex-uri	42
5.2.5. Mecanismul de comunicație inter-task	45
6. Costuri de implementare ale arhitecturii nMPRA și criterii de performanță.....	47
6.1. Analiza puterii consumate.....	47
6.2. Necesarul de resurse pentru implementarea proiectului SoC ce include procesorul nMPRA	48
6.3. Implementarea și analiza comparativă a performanțelor arhitecturii hardware propuse pentru SOTR	51
7. Concluzii, direcții viitoare de cercetare și contribuții	53
7.1. Concluzii finale	53
7.2. Direcții viitoare de cercetare	54
7.3. Contribuții	55
7.3.1. Contribuții teoretice	55
7.3.2. Contribuții practice	56
7.3.3. Diseminarea rezultatelor	57
Bibliografie	59
ANEXA I. Sintetizarea și implementarea proiectului SoC utilizând Vivado Design Suite și Virtex-7	63
ANEXA II. Figura 1. Schimbările de context dintre sCPU0-sCPU1 și sCPU3-sCPU0	68
ANEXA II. Figura 2. Semnalele de ieșire ale etajului pipeline ID/EX	69

ANEXA II. Figura 3. Operanzii furnizați unității aritmetice și logice	70
ANEXA III. Secvență din codul Verilog pentru implementarea planificatorului nHSE.....	71

1. Introducere

Sistemele de operare în timp real (SOTR) sunt prezente în toate aplicațiile înglobate din ariile economice și sociale. Putem observa că sunt din ce în ce mai puține sisteme care să nu utilizeze unul sau mai multe microprocesoare. Din acest considerent cercetările din acest domeniu s-au extins, îmbunătățind SOTR din toate punctele de vedere și garantând astfel performanțe deosebite pentru aplicațiile de timp real.

Dezvoltarea procesoarelor, necesității optimizării și modalității în care este folosit procesorul, algoritmi de planificare și a SOTR au făcut posibilă folosirea timpului la rezoluții de ordinul microsecundelor. Toți acești factori au condus la eficientizarea organizării timpului astfel încât să fie garantată atât predictibilitatea execuției task-urilor cât și satisfacerea condițiilor de timp impuse. Ca urmare a evoluțiilor tehnologice actuale, în cele mai multe domenii s-a obținut un factor de creștere a producției cantitativ și calitativ net superior, cum este cazul și sectorului industrial, unde procesul de automatizare a reușit să degreveze activitățile umane folosite parțial sau total în procesul de fabricație. Aceasta se datorează în principal partajării timpului din sistemele înglobate de timp real și execuției pseudo-paralele a task-urilor într-o implementare de tip uniprocessor.

Una dintre tendințele actuale din domeniul sistemelor de timp real (STR) constă în migrarea către arhitecturi de procesor din ce în ce mai complexe cu sporirea predictibilității execuției și garantarea izolării contextelor task-urilor, obținând astfel aplicații mai sigure și totodată mai performante. **Datorită complexității aplicațiilor și timpilor foarte mici de răspuns din domeniul industrial și automotive a fost necesară proiectarea și dezvoltarea de sisteme hardware cu putere de calcul sporită, permițând astfel o gestiune mai convenabilă a timpului.** Abordarea acestui subiect din punct de vedere științific, tehnologic și economic se datorează evoluțiilor spectaculoase din domeniul sistemelor înglobate și aria de aplicabilitate care poate fi automotive, medical sau robotică. **Elementele de dificultate ale acestei abordări în contextul specificat sunt date de obținerea unei arhitecturi predictibile și performante, prin înglobarea funcțiilor specifice SOTR în hardware și evidențierea îmbunătățirii performanțelor prin programe de test adecvate.** Limitările implementărilor curente sunt date de nivelul proiectării procesorului, memoriei, subsistemului I/O, limbajelor de nivel înalt și a compilatoarelor acestora. **Toate acestea conduc la un timp variabil în comportamentul SOTR implementate în software și un răspuns imprevizibil pentru întreruperi.** Comportamentul nedeterminist la întreruperile externe asincrone se datorează faptului că pentru cele mai multe SOTR comerciale, execuția aceleiași instrucțiuni se finalizează într-un număr variabil de cicli datorită în principal hazardurilor. Arhitecturile obținute trebuie să asigure planificări fezabile chiar dacă factorul de utilizare total al procesorului este aproape de limita maximă.

Cyber-physical systems (CPS) reprezintă următorul pas revoluționar plecând de la sistemele înglobate existente. Împreună cu internetul, serviciile și datele disponibile on line, sistemele înglobate se alătură pentru a forma *Cyber-physical systems*. CPS furnizează baza creării *Internet of Things*. Aceste sisteme au rolul de a valida tehnologii care transformă procesele și aplicațiile inovative într-o realitate, iar granițele dintre lumea virtuală și cea reală vor dispărea. Ca urmare, aceste sisteme promit să revoluționeze interacțiunea noastră cu lumea

fizică la fel cum internetul a transformat comunicarea și interacțiunea personală. Termeni precum, sisteme de timp real, sisteme de operare în timp real și microcontrolere sunt în corelație strânsă cu sistemele înglobate și varianta de viitor a acestora, *Cyber-physical systems*.

Necesitatea obținerii pentru procesele rapide ale unor timpi mici de reacție la stimulii externi a condus la cercetării aprofundate privind arhitectura procesoarelor și a SOTR. În acest caz, majoritatea cercetătorilor din domeniu au ajuns la concluzia că unele componente (sau chiar întreg SOTR) trebuie înglobate în hardware datorită capacității acestuia de a crește prelucrarea paralelă a informației și de a reduce astfel timpii de răspuns al sistemelor înglobate. Izolarea spațială, necesară în sistemele critice înglobate pentru protejarea stării firelor de execuție critice, poate fi garantată prin programarea fiecărei sarcini pe componente distincte de execuție, cum ar fi: fire de execuție (dintr-o arhitectură multi-fir) sau nucleu (într-un procesor multi-nucleu). Astfel, garantând izolarea spațială folosind un procesor pentru fiecare task determină o utilizare inefficientă a resurselor, aceasta fiind o soluție robustă dar inacceptabilă. Pentru a îmbunătăți performanțele sistemelor de operare în timp real trebuie să fie garantată izolarea temporară a firelor de execuție concurente. Predictibilitatea în timp a firelor de execuție facilitează o marjă strânsă pentru WCET (*Worst-Case Execution Time*), evitând utilizarea excesivă sau inefficientă a resurselor.

Tendința actuală în sistemele de operare în timp real este execuția unui număr cât mai mare de task-uri folosind o platformă hardware limitată. Astfel, un singur procesor trebuie să execute mai multe task-uri cu diverse priorități în diferite moduri de lucru ale STR. **Prin mutarea sistemului de operare sau părți ale acestuia în hardware se dorește reducerea surselor de nedeterminism introduse de întreruperile externe asincrone și de timpul variabil de execuție al sistemului de operare în timp real.** Acest timp de execuție variabil este determinat, în principal, de numărul de task-uri, tipul de planificator, dependența de date introdusă de banda de asamblare și întreruperi. Planificatorul de timp real este o unitate de program care controlează lansarea în execuție, întreruperea temporară și terminarea execuției unor module-program pe baza unui algoritm prestabilit, cu scopul de a satisface restricțiile de timp impuse. Planificatoarele hardware au rolul de a degreva procesorul de activitatea de planificare a task-urilor preluând ele această sarcină. Planificatoarele de timp real trebuie proiectate conform unor principii teoretice foarte solide pentru a asigura corectitudinea execuției programelor și implicit funcționarea corectă a aplicațiilor controlate. Realizarea în hardware a procesoarelor specializate, coprocesoarelor sau a planificatoarelor reprezintă o noutate pentru STR, fiind o adevărată provocare în domeniu.

Dispozitivele FPGA la prețuri accesibile [1], cu un număr mare de porți logice, pot reprezenta un suport hardware pentru implementarea și testarea sistemelor de operare în timp real. Tehnologia logicii programabile este o componentă fundamentală din setul de instrumente al oricărui proiectant de circuit și reprezintă suportul pentru dezvoltatorul de dispozitive înglobate [2]. Cu capacitățile lor expansiv favorabile pentru o gamă vastă de aplicații, FPGA-urile sunt ideale pentru rezolvarea multora dintre problemele cu care se confruntă sectorul tehnologiei care evoluează rapid. Principalele avantaje ale matricilor de porți logice programabile includ o flexibilitate sporită [3], o reducere a costurilor și performanțe deosebite.

Obiectivele cercetării privind teza de doctorat

Obiectivul principal al acestei teze de doctorat îl reprezintă cercetările pentru dezvoltarea și implementarea de soluții inovative privind îmbunătățirea performanțelor sistemelor de operare în timp real prin implementarea funcțiilor acestora în hardware. Aceste rezultate vor fi concretizate prin validarea unei microarhitecturi performante, de la care se așteaptă un randament notabil și un câștig de predictibilitate și siguranță. Aceste performanțe pot fi atinse chiar și în cazul apariției unor situații externe de tip hazard sau în cazul unor excepții interne. Arhitectura de procesor nMPRA (*Multi Pipeline Register Architecture*) este una inovativă cu timpi de răspuns foarte mici la stimulii externi [4], [5]. **Îmbunătățirea acestor timpi, precum și minimizarea timpului alocat schimbării contextelor task-urilor, constituie și scopul principal de cercetare al acestei teze de doctorat.** Arhitectura nMPRA utilizează un planificator hardware care este parte constituantă a procesorului, iar controlul acestuia se face prin instrucțiuni dedicate care sunt transmise prin banda de asamblare.

Planificatorul de timp real reprezintă elementul central pentru minimizarea factorului negativ ce îl are supracontrolul sistemului de operare asupra performanțelor STR. Planificarea task-urilor, modul de tratare al salvelor de întreruperi asincrone, cât și timpul necesar schimbării contextelor pot influența negativ limita de planificare pentru sistemele în care numărul întreruperilor este mare iar frecvența de comutare a task-urilor este una superioară datorită încărcării procesorului la limite maxime. Pentru a degreva procesorul de activitatea de planificare și obținerea unui factor de utilizare superior, această teză de doctorat prezintă rezultatele cercetărilor ce reprezintă o noutate în creșterea predictibilității execuției task-urilor de timp real. Prin îmbunătățirea algoritmilor de planificare implementați în hardware împreună cu micșorarea semnificativă a timpului datorat schimbării contextelor s-a realizat o diminuare notabilă a supracontrolului specific sistemului de operare și implicit o îmbunătățire a coeficientului WCET. Una dintre cerințele fundamentale ale unui STR este determinismul execuției task-urilor critice de timp real. Rata de execuție a task-urilor, supracontrolul generat de sistemul de operare și ciclul de ceas necesari schimbărilor de context sunt doar câțiva parametri care pot genera creșterea jitter-ului și ratarea termenelor limită în STR care au la bază planificatoare software.

În această teză de doctorat este descris și validat un sistem de întreruperi pentru un planificator de timp real implementat în hardware care elimină total sau parțial supracontrolul generat de funcțiile specifice sistemului de operare. Noutatea acestui sistem de întreruperi este reprezentată de faptul că pentru selecția task-urilor și gestiunea întreruperilor nu este necesară folosirea unui controler de întreruperi dedicat. Această teză prezintă atât o descriere teoretică detaliată, contribuțiile cât și rezultatele experimentale obținute în urma cercetărilor privind îmbunătățirea performanțelor sistemelor de operare în timp real prin implementarea în hardware a funcțiilor specifice acestora. Pentru a răspunde la întrebările și problemele din cadrul STR actuale, capitolele rezervate descrierii și validării acestui procesor demonstrează funcționalitatea și performanțele de timp real ale planificatorului. Deoarece eficientizează utilizarea timpului și minimizarea jitter-ului, dezvoltarea sistemelor de operare în timp real cu funcții implementate în hardware reprezintă o alternativă realistă și fezabilă la soluțiile existente și descrise în partea introductivă a acestei teze de doctorat.

Structura lucrării de cercetare

Teza de doctorat este structurată pe șapte capitole la care se mai adaugă lista de referințe bibliografice și trei anexe. Astfel, în *capitolul 1*, denumit *Introducere* este prezentată o scurtă descriere a tezei de doctorat, fiind prezentate obiectivele cercetării și structurarea lucrării de cercetare.

Capitolul 2 prezintă și analizează stadiul actual al sistemelor de operare în timp real cu funcții implementate în hardware. Astfel, se subliniază faptul că problema cea mai importantă a unui STR o constituie planificarea resurselor: procesor, memorie, porturile I/O și rețelele de comunicație atunci când sistemele sunt distribuite; s-a sesizat aspectul că tot mai multe sisteme complexe se bazează, în principal, pe controlul procesorului; s-a evidențiat că evoluția într-un ritm alert a dispozitivelor FPGA a influențat puternic atât metodologia de proiectare cât și cerințele impuse uneltelor de dezvoltare; se menționează că un sistem încorporat de timp real, ca orice sistem tehnic, trebuie să fie fiabil și sigur; și nu în ultimul rând s-a sesizat că domeniile de utilizare pot implica însăși siguranța umană, impunând condiții de timp stricte. Din caracteristicile generale ale arhitecturilor de procesor analizate în acest capitol reiese faptul că dezvoltarea mecanismelor în timp real și planificarea proceselor cu condiții stricte de timp este o provocare de actualitate în domeniul STR și implicit în sistemele înglobate și varianta de viitor a acestora, *Cyber-physical systems*. În acest context se evidențiază rolul important al planificatoarelor hardware, care au scopul de a degreva procesorul de activitatea de planificare a task-urilor preluând ele această sarcină. Astfel, este analizată arhitectura nMPRA care utilizează un planificator hardware care este parte constituantă a procesorului, iar controlul acestuia se face prin instrucțiuni integrate care sunt transmise prin banda de asamblare. Concluzia finală, care încheie acest capitol, scoate în evidență avantajele planificării hardware a SOTR și utilizarea arhitecturii nMPRA ca suport pentru cercetările prezentate în această teză de doctorat.

În finalul acestui capitol este prezentată și o comparație între cele mai semnificative arhitecturi de procesor descrise în literatura de specialitate conturând avantajele fiecărei implementări. Astfel, sunt prezentate diferențele dintre cele mai semnificative proiecte analizate, cum ar fi: hthtread, FASTCHART, SPEAR, PRET, FlexPRET, MERASA, JOP, JOP-Plus, XMOS și KOMODO. Această analiză a fost realizată cu scopul de a întreprinde direcții noi de cercetare pentru îmbunătățirea performanțelor STR.

În *capitolul 3* sunt prezentate modelele și algoritmii de planificare care stau la baza micșorării segmentării programelor, cât și la optimizarea execuției task-urilor de timp real. Sunt analizate și descrise mai multe modele astfel încât planificatorul de timp real, validat în această teză, să poată implementa un algoritm de planificare optim atât pentru arhitectura nMPRA cât și pentru setul de task-uri definit de aplicația de timp real.

Puterea de calcul nu este o caracteristică fundamentală a STR, aceasta fiind un termen abstract care depinde de coeficienții procesului pentru care este folosit sistemul de timp real. Așadar, STR sunt acele sisteme care furnizează un răspuns valid într-un interval de timp impus de termenele limită ale task-urilor asociate procesului controlat. Algoritmul de planificare și implicit minimizarea timpului de răspuns la evenimente dintr-un STR reprezintă factori primordiali în creșterea vitezei de răspuns. Aceste aspecte constituie subiecte fundamentale pentru a obține echipamente suficient de rapide folosite în aplicațiile cele mai

exigente. În acest sens, pentru garantarea predictibilității unui STR nu este nevoie doar de o viteză de execuție a task-urilor sporită ci și de o metodă de planificare optimă, pentru a satisface toate sarcinile impuse sistemului înainte de termenele limită. Astfel, STR folosite în industria constructoare de mașini, automatizări generale, industria atomo-electrică, în domeniul aero-spațial și sistemele medicale reprezintă o categorie foarte importantă și indispensabilă din societatea zilelor noastre, unde elementul cu o deosebită importanță îl constituie microprocesorul. Deoarece în majoritatea sistemelor implementate în ariile enumerate anterior, microprocesoarele nu numai că ușurează munca omului în procesul de producție sau dezvoltare, ci reprezintă elementul central ce favorizează augmentarea calității produselor. Acest lucru permite obținerea de performanțe deosebite, importanța STR crescând cu atât mai mult cu cât ele sunt folosite și pentru a elimina producerea de pagube materiale. Eficiența este o altă caracteristică pe care trebuie să o însușească un STR, astfel încât planificatorul din componența acestuia să satisfacă cerințele stringente ale execuției task-urilor pentru un sistem scalabil ce are la dispoziție resurse hardware limitate. În acest context, se poate afirma că un STR trebuie să fie robust și sigur chiar și în acele situații în care cerințele ating puncte de maxim, astfel încât să garanteze o funcționare corectă în procesul din care face parte. Cu toate că toleranța la erori reprezintă o altă caracteristică importantă a acestor sisteme, planificatorul nu trebuie să permită existența situațiilor imprevizibile ce pot afecta siguranța operatorului uman sau chiar al calității produselor și serviciilor rezultate.

Contribuțiile din acest capitol sunt rezultatul cercetărilor teoretice și aplicative privind planificarea de timp real, deoarece teza de doctorat acordă o atenție sporită minimizării supracontrolului datorat sistemului de operare, diminuând efectul de jitter, timpul alocat planificatorului și comutării contextelor. Prin implementarea și validarea procesorului descris în această teză s-a obținut minimizarea efectelor negative produse de acești parametri, îmbunătățind astfel predictibilitatea și determinismul de timp real al procesorului nMPRA.

Capitolul 4 prezintă arhitectura nMPRA și modelul experimental al planificatorului de timp real implementat în hardware. De asemenea este descris și proiectul SoC (*System on Chip*) care integrează procesorul nMPRA, memoria dual-port, driver-ele pentru comunicația UART și modulele care gestionează conexiunile I/O (LCD, selectoare DIP și LED-uri). Această arhitectură de procesor a fost dezvoltată și implementată folosind kit-ul de dezvoltare cu FPGA Virtex-7, fiind în special proiectată pentru minimizarea supracontrolului datorat planificatoarelor clasice, reducerea efectului de jitter și totodată pentru eliminarea imprevizibilității datorate tratării întreruperilor asincrone. În secțiunile aferente capitolului 4 sunt descrise amănunțit blocurile procesorului nMPRA, structura planificatorului nHSE (*Hardware Scheduler Engine*) cât și organizarea internă a proiectului SoC.

Acest capitol analizează arhitectura MIPS32 ca suport pentru implementarea nMPRA. MIPS (*Microprocessor without Interlocked Pipeline Stages*) este o arhitectură de microprocesor de tip RISC (*Reduced Instruction Set Computer*) dezvoltată de *MIPS Technologies*. Consumul scăzut de energie și caracteristicile privind consumul de putere ale sistemelor înglobate ce reprezintă implementări MIPS32, disponibilitatea utilităților pentru dezvoltare în domeniul sistemelor înglobate, precum și faptul că MIPS32 este o arhitectură cunoscută, toate îi asigură procesorului de tip MIPS un rol important în industria sistemelor înglobate.

Această teză prezintă o soluție inovatoare pentru prioritizarea hardware a întreruperilor și evenimentelor atașate la același task. În acest context, întreruperea împrumută prioritatea și comportamentul task-ului. Astfel, comportamentul întreruperilor este mult mai predictibil în contextul unei aplicații de timp real (un task nu poate fi întrerupt decât de întreruperile atașate unui task mai prioritar). Spre deosebire de soluția software, prin hardware orice întrerupere are același timp de răspuns. Mai mult decât atât, soluția propusă poate oferi și priorități statice sau dinamice pentru întreruperi funcție de prioritatea task-ului la care sunt atașate. Putem spune că soluția aleasă conține un management unitar al întreruperilor și evenimentelor. S-a implementat soluția hardware pentru gestionarea întreruperilor sub forma unui bloc hardware adițional denumit *Blocul codor de prioritate* care generează identificatorul întreruperii având cea mai mare prioritate. Schema de prioritizare a întreruperilor a fost extinsă și asupra evenimentelor devenind astfel o soluție generalizată pentru oricare nou tip de eveniment care poate fi atașat la arhitectura nMPRA pentru tratarea situației în care devin active evenimente multiple. Schema este simplă și se poate aplica tuturor tipurilor de evenimente. Contribuțiile din acest capitol constau în prezentarea unor metode originale pentru creșterea predictibilității procesorului nMPRA, micșorarea efectului de jitter prin implementarea de către nHSE a unui spațiu unificat de priorități pentru task-uri și evenimente și augmentarea factorului de execuție al procesorului.

Capitolul 5 prezintă în detaliu contribuțiile privind implementarea procesorului nMPRA utilizând kit-ul cu FPGA Virtex-7 produs de Xilinx, limbajul de descriere hardware Verilog, platforma de dezvoltare Vivado, simulatorul Vivado și analizorul ChipScope. În cadrul acestui capitol sunt ilustrate blocurile funcționale ale procesorului nMPRA, instrucțiunile procesorului și ale planificatorului implementat în hardware și regiștrii necesari implementării planificatorului nHSE. Utilizând structura organizațională a arhitecturii MIPS32, este propusă arhitectura cu regiștrii pipeline multiplicați, nMPRA. Cu toate că structura hardware cu cinci etaje pipeline a procesorului nMPRA este una foarte complexă, acest capitol prezintă și analizează testele dedicate implementării procesorului și a memoriei de instrucțiuni și de date, testele efectuate pentru verificarea modulului Bootloader și a driver-elor pentru periferice. Pentru aceasta au fost efectuate mai multe teste practice în vederea implementării și testării practice a întregului proiect SoC.

Arhitectura propusă în această teză de doctorat, înlocuiește metodele de salvare pe stivă cu un algoritm de remapare ce permite execuția unui nou task începând cu următorul ciclu-procesor, conține o implementare originală bazată pe o structură hardware utilizată pentru planificarea statică și dinamică a task-urilor, permite un management unitar al evenimentelor și întreruperilor, definește o metodă de atașare a întreruperilor la task-uri, asigurând cerințele sistemelor de timp real. Performanța arhitecturii nMPRA nu constă în puterea de procesare, ci în viteza de comutare de context și în viteza de execuție a algoritmului de planificare. Arhitectura îndeplinește cerințele sistemelor de timp real, consumul de memorie și de putere fiind redus. nMPRA oferă un grad înalt de siguranță în utilizare datorită izolării totale a contextelor task-urilor, eliminând astfel posibilitatea de corupere a datelor. Cu toate acestea, realizarea procesorului nMPRA cu funcții implementate în hardware asigură determinismul execuției task-urilor critice datorită mecanismului de priorități care garantează acest lucru. În acest capitol se reliefează o soluție inovatoare pentru prioritizarea întreruperilor atașate la aceeași task, cu un înalt grad de flexibilitate și care produce același timp de răspuns pentru

toate întreruperile. Schema hardware de prioritizare globală a întreruperilor este simplă și poate fi aplicată tuturor categoriilor de evenimente, permițându-se chiar introducerea de noi categorii de evenimente prin actualizarea arhitecturii și introducerea unui nou registru capcană pentru fiecare tip nou de eveniment.

Formele de undă ilustrate în acest capitol au fost obținute astfel încât să se prezinte starea semnalelor la cele mai semnificative momente de timp, semnalele fiind preluate atât în etapa de simulare cât și în cea de depanare hardware utilizând circuitul FPGA Virtex-7. Capturile de ecran au fost obținute cu ajutorul simulatorului Vivado și a instrumentului de depanare hardware ChipScope, având drept scop validarea arhitecturii nMPRA și a proiectului SoC.

În **capitolul 6** sunt descrise criteriile de performanță ale procesorului nMPRA, prezentând totodată și o comparație realistă cu alte implementări propuse în literatura de specialitate. Deoarece operația de multiplicare a resurselor multiplexate s-a realizat la nivelul fiecărui element de memorare, s-a obținut un raport cost/performanță mai mult decât convenient. Astfel, s-a realizat doar multiplicarea regiștrilor conținuți în etajele pipeline și nu multiplicarea întregului modul Verilog ce implementează etajul pipeline respectiv, obținând un avantaj semnificativ față de alte arhitecturi de procesor cu planificator hardware. Acest studiu susține faptul că arhitectura nMPRA este una scalabilă și flexibilă, ce poate fi folosită cu succes în STR de mici dimensiuni. Se poate afirma că necesarul de memorie alocat implementării procesorului nMPRA este acceptabil în condițiile în care alte implementări de procesor propuse în literatura de specialitate utilizează sute de KB de memorie RAM. Ultimele secțiuni din acest capitol prezintă atât un exemplu de model utilitar cât și domeniile de aplicație în care modelul experimental de procesor validat în această teză de doctorat poate fi folosit cu succes.

Capitolul 7 prezintă concluziile finale ale acestei lucrări de cercetare, contribuțiile aduse în acest domeniu, direcțiile viitoare de cercetare cât și lista lucrărilor publicate și susținute în reviste și conferințe internaționale, diseminând astfel rezultatele obținute în această perioadă de cercetare relativ scurtă. Teza de doctorat se încheie cu o listă de referințe bibliografice și un număr de trei anexe unde sunt grupate o serie de informații cu privire la implementarea practică a arhitecturii nMPRA și a planificatorului nHSE.

Anexa I prezintă metodologia de elaborare, sintetizare și implementare a proiectului SoC utilizând mediul de proiectare *Vivado Design Suite*. Această anexă mai conține și descrierea metodologiei de configurare și testare a kit-ului de evaluare VC707 prin intermediul proiectului *Built-in self test* (BIST).

Anexa II prezintă formele de undă obținute în procesul de validare a procesorului nMPRA cât și a proiectului SoC. Sunt ilustrate capturi de ecran care evidențiază cele mai reprezentative semnale ale planificatorului, unității de control și calea de date a procesorului.

Anexa III prezintă o parte din codul Verilog scris pentru implementarea planificatorului nHSE. Codul sursă Verilog a fost simulat, sintetizat și implementat în FPGA folosind mediul de proiectare Vivado 2016.2, acesta având un rol esențial în facilitarea activității de cercetare.

2. Stadiul actual al SOTR cu funcții implementate în hardware

În cadrul acestui capitol sunt prezentate și descrise cele mai semnificative arhitecturi de procesor cu funcții implementate în hardware. Sunt discutate aspectele teoretice și practice în ceea ce privește planificatoarele arhitecturilor analizate, implementate fie prin software fie prin hardware. Aceste cercetări se focalizează în principal pe obținerea unor arhitecturi de procesor predictibile, accelerarea planificatoarelor și implementarea unor algoritmi de planificare din cadrul nucleelor de timp real realizate în hardware. Arhitecturile de procesor luate în considerare sunt în general scalabile, în funcție de caracteristicile FPGA-ului utilizat [6], [7] și tipul de procesor implementat.

În cadrul procesoarelor folosite în aria aplicațiilor mobile, cum ar fi sistemele medicale sau multimedia, creșterea frecvenței de lucru nu reprezintă o soluție eficientă în mod direct datorită creșterii consumului energetic. Astfel, prin integrarea pe aceeași pastilă de siliciu a unui număr de nuclee de calcul similare sau proiectarea unor arhitecturi multi-fir și cu linii pipeline din ce în ce mai adânci, s-a permis o gestiune mai optimă a timpului procesor. Pentru cazul procesoarelor din aria aplicațiilor mobile, prin proiectarea unor arhitecturi multithreading sau hiperthreading s-a permis o gestiune mai optimă a timpului procesor fără a crește frecvența de operare a procesorului. Pentru a exemplifica, se poate aminti de familia de procesoare ARM Cortex A9, A15 sau A53 cu toate că acestea sunt destinate și pentru aplicații multimedia nu doar pentru STR. Pe aceste procesoare multi-core rulează aplicații de timp real care utilizează două, patru, sau opt nuclee, în funcție de performanța și consumul de energie din aplicațiile mobile. Un alt exemplu este procesorul *i.MX 6SoloX* propus de Freescale. Această implementare reprezintă o soluție fiabilă ce ajută la securitatea aplicațiilor din domeniul *Internet of Things* (IoT). Astfel, *i.MX 6SoloX* este primul procesor de aplicații din industrie care integrează într-un singur chip un nucleu ARM Cortex-A9 și un nucleu ARM Cortex-M4 [2]. Acest procesor a fost proiectat pentru a permite dispozitivelor de timp real, performanță de excepție și eficiență energetică. Procesorul oferă capacitatea de a rula pe nucleul Cortex-A9 un sistem de operare cu o interfață utilizator și în același timp beneficiază de determinismul în timp real oferit de nucleul Cortex-M4. Acest lucru este fundamental pentru o gamă largă de aplicații din domeniul industrial, automotive sau medical, deoarece acestea necesită o interfață cu utilizatorul modernă dar, mai cu seamă trebuie să fie fiabile, sigure și deterministe în comunicația cu alte dispozitive din rețea.

În cadrul acestui capitol sunt analizate diferite arhitecturi de procesor și planificator hardware ce au fost propuse în literatura de specialitate. Pentru a păstra amprenta conceptuală lăsată de autori în proiectarea și prezentarea acestor proiecte s-a ales ca schemele bloc ale implementărilor de procesor să fie cât mai asemănătoare cu cele originale, acestea fiind preluate din articolele de specialitate. Principalele caracteristici a celor mai reprezentative implementări descrise în acest capitol, precum și diferențele dintre acestea, sunt concretizate prin datele prezentate în Tabelul 2-1. Aspectele urmărite se referă la: implementarea planificatorului, replicarea resurselor (PC, fișierul de regiștri sau regiștrii pipeline), banda de asamblare (numărul de etaje corespunzătoare fiecărei benzi de asamblare), mecanisme de sincronizare și comunicație, tipul de planificare și tipul implementării. În ceea ce privește planificatorul sistemului de operare în timp real, majoritatea implementărilor au în vedere obținerea unei arhitecturi deterministe în timp.

Tabelul 2-1: Caracteristicile principalelor SOTR prezentate în acest capitol

	Copro- cesor	Tipul planificatorului	Replicarea resurselor	Pipeline	Mecanisme de sincronizare și comunicație inter-task	Tipul de planificare	Tipul implementării
hthread [8], [9]	Da	Un planificator HW reprezentând un nucleu din SOTR	Nu	Nu	Da (implementate în hardware)	Static (FIFO, Round Robin, bazat pe priorități)	Single-core
FASTCHART [10], [11]	Nu	HW	Nu	Nu	Da (sunt introduse în următoarea versiune [11])	Static (rate monotonic)	Single-core
SPEAR [12]	Nu	SW	Nu	O bandă de asamblare cu 3 etaje	Da (implementate în hardware)	Static	Single-core
nMPRA [4], [13], [14]	Nu	HW	Da	O bandă de asamblare cu 5 etaje	Da (implementate în hardware)	Static și dinamic	Single-core
PRET [15], [15]	Nu	HW	Nu	O bandă de asamblare cu 5 etaje	Da	Static (Round Robin)	Single-core
FlexPRET [17]	Nu	HW	Nu	O bandă de asamblare cu 5 etaje	Da (PRET-C)	Static și dinamic (EDF, rate- monotonic)	Single-core
JOP și JOP-Plus [18], [19]	Nu	SW	Nu	O bandă de asamblare cu 3 etaje	Da	Dinamic	Single-core
Merasa [20], [21]	Nu	Un planificator HW și unul SW pentru optimizare	Da (Fișierul de regiștri)	Două benzi de asamblare cu 5 etaje/nucleu	Da (pentru Single-core și multi-core)	Dinamic	Multi-core
XMOS [22], [23]	Nu	HW	Da (Fișierul de regiștri)	O bandă de asamblare cu 4 etaje/nucleu	Da (pentru Single-core și multi-core)	Dinamic	Multi-core
Komodo [24]	Nu	SW	Da	O bandă de asamblare cu 4 etaje	Nu	Dinamic	Single-core

3. Analiza algoritmilor de planificare folosiți în STR și a funcțiilor SOTR

Pentru a descoperi și aprofunda direcțiile de cercetare din domeniul arhitecturilor de procesoare cu un singur nucleu și multi-nucleu, trebuie să știm dacă în acest sens mai putem optimiza arhitecturile de procesor cu un singur nucleu astfel încât să obținem o eficiență maximă în aplicațiile de timp real cât și în cele cu consum redus de energie. Prin folosirea timpului procesor cu un factor de utilizare superior, se poate asigura controlul predictibil și determinist al unui proces critic specific sistemelor de timp real.

În acest capitol sunt prezentate câteva metode și algoritmi de planificare folosiți în planificarea task-urilor de timp real din STR. Sistemele înglobate de timp real sunt acele sisteme care oferă un răspuns corect într-un interval de timp prestabilit [3]. Acest interval de timp, denumit termen limită sau deadline, conduce la împărțirea sistemelor de timp real în:

- *STR SOFT* - ratarea unui termen limită nu determină un efect critic;
- *STR HARD* - neîndeplinirea unui termen limită determină o situație de hazard.

Unele sisteme de timp real critice folosite în aeronautică, automotive, robotică sau în industrie sunt critice în adevăratul sens al cuvântului. Pentru aceste sisteme izolarea spațială a task-urilor și predictibilitatea execuției acestora reprezintă trăsături foarte importante.

Principala caracteristică a STR este aceea de a asigura controlul determinist și predictibil al unui proces [25]. În aplicațiile de timp real critice, obținerea unui răspuns corect după termenul limită prestabilit este insuficient și nu mai poate fi luat în considerare. În funcție de consecințele datorate ratării unui termen limită, task-urile de timp real se împart în trei categorii:

- Dacă rezultatele produse de un task după termenul său limită conduc la efecte catastrofice, acesta se poate numi task de timp real *hard*;
- Un task se poate considera *firm* dacă rezultatele produse de acesta, după termenul limită, nu mai sunt folosite în sistem și nu implică daune;
- Task-urile de timp real se pot numi *soft*, dacă rezultatele produse după termenul limită pot fi folosite în sistem chiar dacă degradează performanțele acestuia.

În acest capitol vom prezenta câteva probleme de bază care sunt luate în considerare în timpul proiectării sistemelor de operare în timp real, utilizate pentru controlul aplicațiilor critice [26]. Vom mai discuta atât despre predictibilitatea în timp, sincronizarea inter-task cât și de mecanismele de comunicație pentru interschimbarea informațiilor de stare ale task-urilor periodice. În abordările tradiționale, un sistem de calcul poate fi văzut ca o mașină secvențială. Cele mai multe limbaje de programare necesită atenția programatorului pentru specificarea algoritmilor ca secvențe de instrucțiuni. Procesoarele execută programe prin extragerea instrucțiunilor mașină, câte una pe rând într-o anumită secvență. Fiecare instrucțiune este executată într-o secvență de operații precum: extragerea instrucțiunii, extragerea operanzilor, realizarea operației aritmetice, logice sau de lucru cu memoria și stocarea rezultatului [27]. Deoarece costul implementării hardware a unităților de calcul a scăzut iar tehnologiile de proiectare și implementare au evoluat semnificativ, arhitecții au la

îndemână mult mai multe oportunități pentru a garanta predictibilitatea sistemelor și paralelismul execuției programelor. Cel mai important criteriu de performanță pentru un procesor este rata cu care execută instrucțiunile, $MIPS\ rate = f * IPC$, f fiind frecvența ceasului procesorului, iar IPC (instrucțiuni per ciclu) este o medie a numărului de instrucțiuni executate pentru fiecare ciclu de ceas.

Pentru îmbunătățirea performanțelor, arhitecții au abordat problema prin creșterea frecvenței de ceas și augmentarea coeficientului IPC, crescând numărul instrucțiunilor care sunt completate într-un ciclu de ceas. Pentru a realiza acest lucru a fost nevoie de implementarea benzilor de asamblare cu mai multe etaje iar mai apoi executarea în paralel a instrucțiunilor multiple, obținând astfel arhitecturi superscalare. Prin folosirea pipeline-ului și a pipeline-ului multiplu, principala problemă este să se maximizeze utilizarea fiecărui etaj al benzii de asamblare. Pentru aceasta, arhitecții au propus și creat mecanisme complexe pentru execuția instrucțiunilor într-o ordine diferită, crescând astfel complexitatea întregului procesor și menținând totodată puterea consumată în limite acceptabile. O alternativă pentru această problemă, care permite un înalt grad de paralelizare la nivel de instrucțiune, este tehnica denumită multithreading. În esență, fluxul instrucțiunilor este divizat în mai multe fluxuri denumite și thread-uri, astfel încât aceste thread-uri să fie executate în paralel. Pentru aceasta au fost realizate o varietate de proiecte multithreading, implementate în proiecte experimentale și chiar comerciale.

Conceptul de *thread* folosit în descrierea procesoarelor *multithreaded* poate sau nu să coincidă cu conceptul de *thread software* din cadrul sistemelor de operare multiprogram. Așadar, un proces este o instanță a unui program ce rulează pe computer. Acest proces încorporează două caracteristici și anume, resursele și planificarea. Resursele sunt reprezentate de spațiul necesar pentru stocarea imaginii procesului (program, date, stivă și diferite attribute care definesc procesul), iar planificarea procesului presupune execuția sa, ce se poate realiza simultan cu alte procese, fiecare având definită o prioritate și o stare, precum *RUN*, *READY* sau *IDLE*. Schimbarea contextelor este o operație de trecere a procesorului de la un proces la altul salvând toate datele de control al procesului, regiștrii interni și alte informații necesare restaurării acestuia. Conceptul de *thread* constituie unitatea de lucru ce reprezintă procesul. Un thread este executat secvențial și întreruptibil, planificatorul având potențialitatea să selecteze oricând un alt thread. O schimbare a thread-ului este actul prin care controlul procesorului este transferat de la un thread la altul din cadrul aceluiasi proces. În mod tipic, această schimbare este mai puțin costisitoare decât schimbarea proceselor. Așadar, un thread este caracterizat de o execuție și planificare în schimb ce procesele sunt caracterizate atât de execuție și planificare cât și de resursele acestuia. Deoarece multiple thread-uri din același proces pot partaja aceleași resurse, comutarea thread-urilor este mai mică consumatoare de timp decât comutarea proceselor. Cele mai multe sisteme de operare tradiționale acceptă thread-urile, iar toate procesoarele comerciale până în prezent au folosit tehnica de explicit multithreading. Aceste sisteme execută concurent instrucțiuni din diferite thread-uri explicite, fie prin intercalarea instrucțiunilor aparținând diferitelor thread-uri utilizând o bandă de asamblare comună (partajată de aceste thread-uri), fie prin execuția paralelă utilizând benzi de asamblare separate.

3.1. Algoritmi de planificare

Această secțiune descrie și compară abordările actuale care au contribuit la reducerea segmentării execuției programelor, prezentând metode eficiente pentru minimizarea costurilor de planificare prin eliminarea întreruperilor inutile. Întrebarea dacă sistemele preemptive sunt mai bune decât sistemele non-preventive a fost abordată pentru o lungă perioadă de timp. În literatura de specialitate au fost furnizate multe soluții parțiale, unele probleme fiind încă obiectul discuțiilor. Fiecare dintre aceste soluții are avantaje și dezavantaje, în funcție de predictibilitatea și eficiența sistemului pentru care au fost implementate. Schimbarea contextelor este un factor cheie în algoritmi de planificare de timp real deoarece permite sistemului de operare să aloce imediat procesorul task-urilor cu prioritate mai mare. În sistemele full-preemptive, execuția task-ului curent poate fi întreruptă în orice moment de un alt task cu prioritate mai mare. Execuția task-ului întrerupt se va relua doar atunci când nu mai există task-uri cu prioritate mai mare pregătite pentru execuție. În unele implementări schimbarea contextelor poate fi complet interzisă pentru a evita interferențele imprevizibile dintre task-uri dar și pentru îmbunătățirea predictibilității sistemului. Pentru unele sisteme de timp real, planificatorul preemptiv poate fi dezactivat doar pentru anumite intervale de timp în timpul execuției secțiunilor critice, cum ar fi ISR. Un dezavantaj principal al implementărilor non-preventive este acela că introduce un factor suplimentar de blocare pentru task-urile cu prioritate mare, dar cu toate acestea, există mai multe avantaje importante atunci când se adoptă acest tip de planificare. Atunci când realizăm o analiză a sistemelor de operare, inclusiv a planificatorului acestuia, trebuie luate în considerare următoarele aspecte [3]:

- În multe situații practice, precum planificarea I/O sau comunicația utilizând mediile partajate, orice întrerupere este imposibil sau foarte greu de acceptat. Aceasta deoarece, suspendarea task-ului curent va determina o creștere a efectului de *cache miss* și o influență negativă asupra mecanismului pre-fetch, implicând un WCET impredictibil.
- În planificarea non-preemptivă, problemele introduse de excluderea mutuală sunt neînsemnate deoarece, prin natura algoritmului de planificare se garantează accesul exclusiv la resursele partajate.
- În STR hard cu planificare non-preemptivă efectul de jitter este minim pentru toate task-urile din sistem, simplificându-se astfel tehnicile de control pentru compensarea și diminuarea efectelor negative datorate întârzierilor.
- Execuția non-preemptivă permite folosirea tehnicilor de partajare a stivei pentru a salva spațiu de memorie din sistemele înglobate de mici dimensiuni.

Planificatoarele preemptive introduc fluctuații pentru timpii de execuție a task-urilor, degradând astfel predictibilitatea sistemului. În procesul de proiectare al acestor tipuri de planificatoare trebuie avut în vedere existența unor costuri introduse de:

- *Planificare* - reprezintă timpul consumat de algoritmul de planificare;
- *Pipeline* - însumează timpul datorat ciclilor de ceas pierduți de instrucțiunile care au fost deja extrase și decodificate, deoarece pe banda de asamblare trebuie introduse instrucțiunile noului task, timpul necesar introducerii noului task pe banda de

asamblare și timpul datorat refacerii benzii de asamblare pentru task-ul întrerupt în momentul în care acesta își reia execuția;

- *Cache-related* - reprezintă timpul necesar pentru a încărca liniile cache pierdute în momentul schimbării contextului;
- *Bus-related* - reprezintă timpul introdus de operațiile de acces la memoria RAM, datorate efectului de *cache miss*.

Însumarea tuturor acestor timpi, sau doar o parte din aceștia, reprezintă *Architecture related cost*, acest cost fiind caracterizat de o variație semnificativă în funcție de punctele în care are loc schimbarea de context. În analiza algoritmilor de planificare trebuie să se țină cont de aspecte precum: complexitatea implementării, eficacitatea schemei de planificare și predictibilitatea în estimarea coeficientului *Architecture related cost*.

3.2. Supracontrolul sistemelor de operare în timp real

Supracontrolul sistemelor de operare în timp real reprezintă timpul folosit de procesor pentru execuția primitivelor și mecanismelor nucleului, cum ar fi comutarea de context, introducerea task-urilor în cozile de așteptare, lansarea rutinei de tratare a întreruperilor [28], actualizarea structurilor de date precum TCB, SCB sau implementarea mecanismelor de sincronizare și comunicație inter-task. Timpul necesar efectuării acestor operații este de obicei mult mai mic decât timpul de execuție al task-urilor, fiind adesea neglijat în analiza de planificare și în testele finale pentru validarea nucleelor de timp real.

În unele cazuri, atunci când task-urile aplicației au timpi mici de execuție și constrângeri severe de timp, supracontrolul introdus de execuția funcțiilor de bază ale SOTR nu poate fi neglijat, putând genera interferențe deloc neglijabile în execuția task-urilor. În aceste situații, predictibilitatea sistemului poate fi garantată numai dacă efectele supracontrolului sistemului de operare sunt luate în considerare în analiza de fezabilitate pentru stabilirea schemei de planificare. Timpul necesar schimbării contextelor reprezintă cel mai semnificativ factor din orice SOTR [29]. Acesta este o limită intrinsecă a nucleului, care nu depinde de algoritmul de planificare și nici de structura setului de task-uri. În cazul sistemelor de timp real, un alt factor de supracontrol este timpul necesar procesorului pentru a executa rutina de tratare a întreruperilor.

3.3. Structura generală a nucleului de timp real și funcțiile acestuia

Sistemul de operare în timp real este un program care controlează execuția programelor aplicație, funcționând ca o interfață între aplicații și partea hardware. Sistemul de operare maschează programatorului detaliile părții hardware, furnizând totodată o interfață cât mai abordabilă pentru o utilizare mai eficientă a întregului sistem.

Un sistem de operare tipic garantează servicii în următoarele arii: crearea programelor, execuția programelor, accesul la dispozitivele I/O, controlul acceselor la fișiere, accesele sistem, detecția și corecția erorilor și monitorizarea performanței. Sistemul de operare decide când un dispozitiv I/O poate fi folosit de un program în execuție, controlând totodată accesul la fișierele utilizator. Procesorul însuși este o resursă, iar sistemul de operare trebuie să determine cât de mult timp procesor poate fi dedicat execuției unui anumit program utilizator. În cazul sistemelor multi-procesor, această decizie trebuie să acopere execuția tuturor procesoarelor. Arhitectura setului de instrucțiuni definește repertoriul instrucțiunilor în

limbajul mașină pe care un sistem îl poate înțelege și respecta. Această interfață este granița dintre hardware și software. Pentru programele utilizator este disponibil un subset de instrucțiuni denumite *user ISA*, iar sistemul de operare are acces la instrucțiuni în limbaj mașină adiționale (*system ISA*) care se ocupă cu gestionarea resurselor.

Nucleul reprezintă partea cea mai profundă a oricărui sistem de operare, fiind în contact direct cu partea hardware reprezentată de nivelul mașină. De obicei un nucleu execută următoarele activități: gestiunea proceselor, gestiunea întreruperilor și sincronizarea proceselor. Gestiunea proceselor este serviciul principal pe care sistemul de operare trebuie să-l furnizeze. Aceasta implică implementarea unor funcții precum: crearea și terminarea proceselor, planificarea job-urilor, operațiile de planificare, schimbarea contextelor și alte activități relative. Mecanismul de tratare a întreruperilor reprezintă partea sistemului de operare ce garantează gestiunea întreruperilor care pot fi generate de către dispozitivele periferice precum convertoare analogic digitale sau senzori. În sistemele de operare clasice acest mecanism implică execuția unei rutine dedicate fiecărei întreruperi (*driver*), pentru a transfera date de la dispozitivul periferic către memoria principală, sau invers. Așadar, task-urile aplicației pot fi întrerupte în orice moment de către rutinele dedicate întreruperilor. În SOTR, această abordare poate introduce întârzieri impredictibile pentru task-urile critice, cauzând nerespectarea termenelor limită de execuție a acestora. Din acest motiv în STR, mecanismul de tratare a întreruperilor este integrat cu mecanismul de planificare. Astfel, întreruperile pot fi programate în același mod ca task-urile, garantând astfel fezabilitatea sistemului de operare chiar și atunci când întreruperile sunt tratate în timp real.

După cum se poate vedea în Figura 3-1 structura nucleului de timp real poate fi împărțită în patru straturi și anume: *Machine layer*, *List management*, *Processor management* și *Service layer*. Nivelul *Machine layer* este scris în limbaj de asamblare, reprezentând acel nivel care interacționează direct cu partea hardware, nefiind vizibil nivelului utilizator. Pe acest nivel se implementează primitive precum tratarea întreruperilor și a timer-elor sau schimbarea contextelor task-urilor. Nivelul *List management* realizează gestiunea și memorarea stării task-urilor, furnizând primitive ce permit introducerea sau eliminarea task-urilor în liste sau cozi de așteptare. *Processor management layer* reprezintă nivelul în care se realizează operația de planificare. În ultimul nivel *Service layer*, este implementat setul de funcții sistem vizibile utilizatorului. Putem enumera funcții precum crearea, suspendarea și terminarea task-urilor.

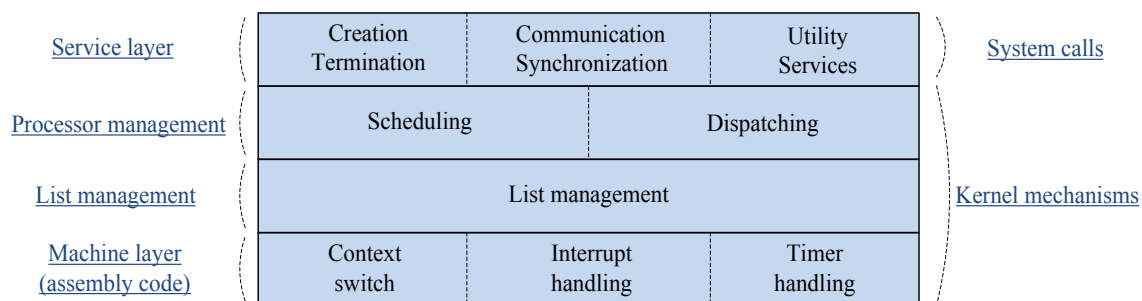


Figura 3-1 Structura unui sistem de operare în timp real [3]

4. Descrierea procesorului nMPRA și a resurselor software/hardware

nMPRA (Multi Pipeline Register Architecture) este acronimul pentru o arhitectură MIPS cu regiștrii pipeline multiplicată de n ori. Arhitectura nMPRA a fost propusă și descrisă inițial în [4] și [5]. Astfel, s-a luat în considerare o linie pipeline MIPS simplă, prezentată în [30] și s-a multiplicat de n ori PC-ul, regiștrii pipeline și fișierul de regiștri (Figura 4-1). Ca urmare, structura reprezentând resursele multiplexate de n ori împreună cu blocurile comune dintre regiștrii pipeline (memoria de instrucțiuni, ALU, memoria de date, unitatea de detectare a hazardului, unitățile de avansare și unitatea de control) formează o arhitectură MIPS tipică [31] pe care o vom numi semiprosesor (sCPU). O instanță i a acestui semiprosesor va fi denumită sCPU $_i$.

Unitatea nHSE are rolul de a activa la un moment dat doar un singur sCPU $_i$ din cele n . Dacă un sCPU $_i$ este oprit la un moment dat de nHSE și este activat alt sCPU $_i$, toată informația specifică programului care se rula pe sCPU $_i$ -ul oprit este conservată datorită multiplicării regiștrilor PC, regiștrii pipeline și fișierul de regiștri. Programul care rulează pe un sCPU $_i$ poate fi un task dintr-o aplicație multitasking. Trecerea de la un sCPU $_i$ la altul nu necesită salvarea regiștrilor de uz general și nici ștergerea conținutului regiștrilor pipeline, acest fapt determinând o comutare foarte rapidă a contextelor. Dacă fiecare sCPU $_i$ rulează un task i , rezultă că și comutarea de la un task la altul se face foarte rapid [32].

Arhitectura nMPRA implementează un planificator hardware care face parte din procesor, comunicația cu acesta realizându-se cu ajutorul unui set de instrucțiuni dedicate. În arhitecturile de procesor clasice de tip preemptive, procedura de schimbare a contextelor presupune salvarea stării firului de execuție în memoria internă de lucru sau pe stivă. Această operație poate determina un efect de jitter și implicit poate afecta predictibilitatea STR critice de tip hard. Pentru a elimina acest dezavantaj, nMPRA comută contextele pe principii remapării resurselor multiplexate precum registrul PC, fișierul de regiștri și regiștrii pipeline.

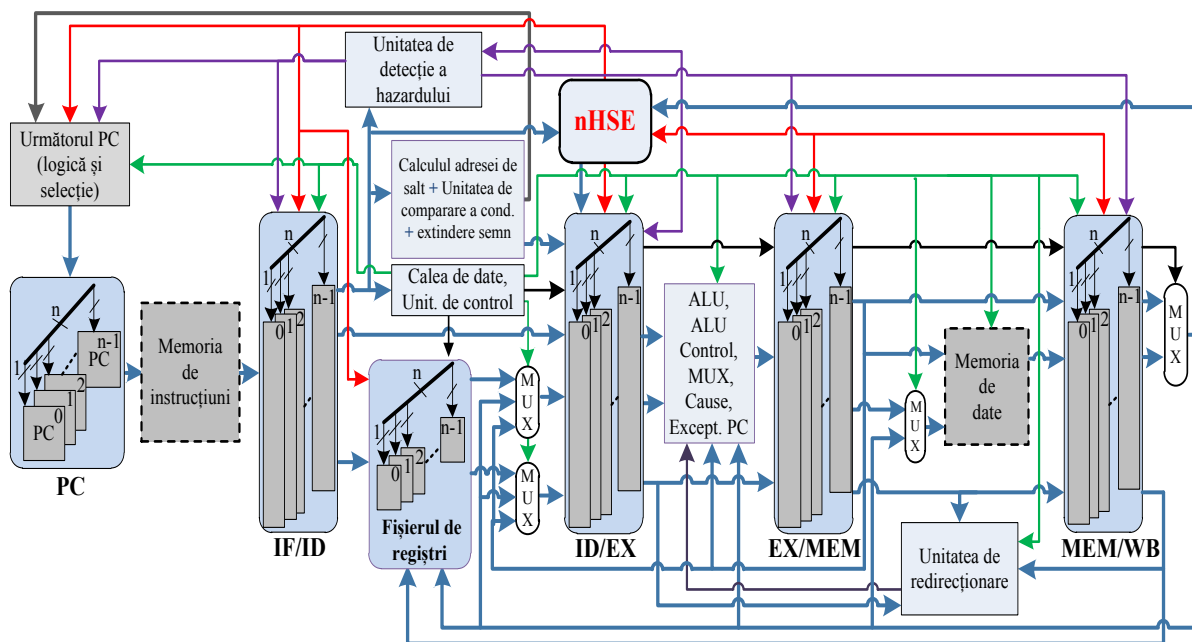


Figura 4-1 Arhitectura nMPRA

Implementarea nMPRA validată în această teză de doctorat are la bază proiectul XUM descris în [33], acesta fiind un procesor MIPS32 cu cinci etaje pipeline. În Figura 4-2.a se poate observa multiplicarea resurselor la nivelul fiecărui element de memorare, iar în Figura 4-2.b sunt ilustrați regiștrii pipeline multiplicați de n ori. Memoria necesară pentru implementarea acestor regiștri este direct proporțională cu numărul de task-uri. Pentru a garanta o execuție predictibilă, autorii aleg o schemă de planificare fixă având la bază codificatorul de adrese prioritar din componența planificatorului hardware. Implementarea algoritmului de planificare dorit rămâne la alegerea utilizatorului, în funcție de frecvența de execuție a task-urilor sau de tipul aplicației controlate de SOTR. De aceea, operația de remapare a resurselor și blocul de timer-e din componența nHSE reprezintă mecanisme valoroase care alocă spațiu proiectantului să implementeze o schemă de planificare predictibilă de tip Round Robin cu ignorarea priorităților. Pentru izolarea spațială a task-urilor s-a ales o implementare specială a fișierului de regiștri care, pe lângă faptul că realizează remaparea contextelor task-urilor fără a mai fi necesară salvarea acestora, are avantajul că introduce un jitter minim prestabilit.

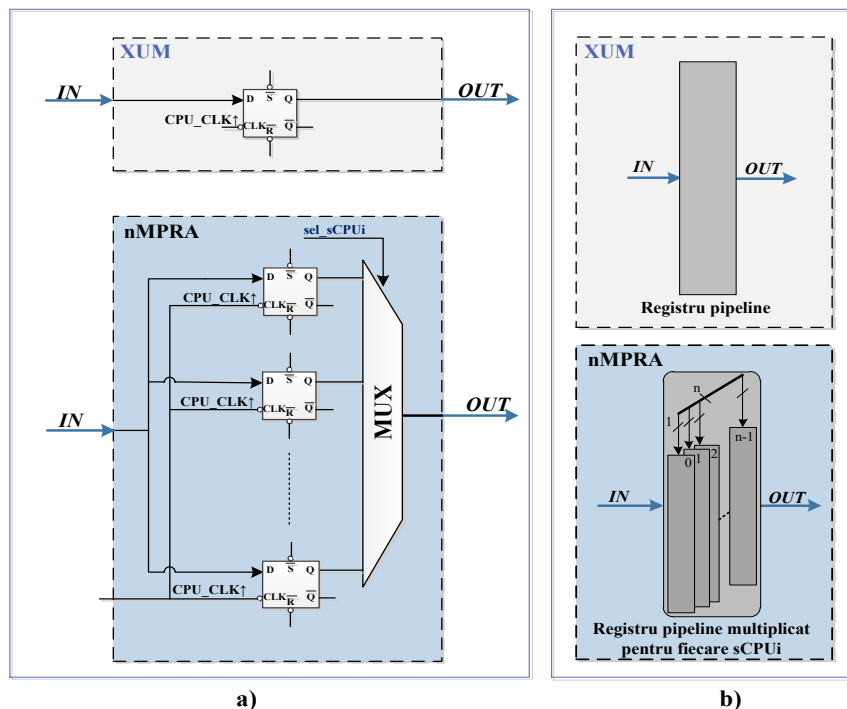


Figura 4-2 Multiplicarea resurselor din cadrul arhitecturii nMPRA raportată la procesorul XUM [33]

Implementarea prezentată în această teză de doctorat folosește structura organizatorică a arhitecturii MIPS, introducând câteva instrucțiuni speciale pentru controlul planificatorului hardware. Se înglobează astfel utilitatea de planificator într-un bloc funcțional separat dar care face parte din procesor, eliminând dezavantajele specifice planificatoarelor software actuale. În comportamentul SOTR actuale, există situații în care apare fenomenul de inversiune a priorităților când două task-uri partajează o resursă exclusivă. De asemenea există posibilitatea de inversare a priorităților când task-urile de prioritate ridicată sunt suspendate de întreruperile asiguate unor task-uri de prioritate mai scăzută, afectând WCET-ul întregului sistem. Ordonarea task-urilor și întreruperilor într-un spațiu unificat are scopul de a reduce din acest dezavantaj, făcând din arhitectura nMPRA o implementare predictibilă.

În ceea ce privește performanțele arhitecturii nMPRA, diferențele sunt superioare față de implementările clasice care salvează regiștrii pe stivă. În cazul salvării contextelor task-urilor prin intermediul stivei, timpul dedicat acestei operații este direct proporțional cu mărimea și numărul de regiștri care trebuie salvați. Aceste acțiuni de salvare și restaurare a contextului implică execuția unui număr de cicli de acces la memorie, determinând astfel un supracontrol adițional care reduce timpul util al procesorului. Timpul necesar schimbării contextelor este dependent de numărul și dimensiunea regiștrilor salvați pe stivă și lățimea magistralei de date care interconectează procesorul cu memoria RAM. Arhitectura nMPRA folosește doar o parte din acest mecanism de salvare pe stivă, păstrând doar funcționalitatea necesară stocării informațiilor corespunzătoare apelurile funcțiilor imbricate.

În cadrul arhitecturii nMPRA fiecare task are un registru PC, un fișier de regiștri și un set de regiștri pipeline propriu. Comutarea contextelor task-urilor este controlată de către planificator prin simpla operație de remapare a resurselor multiplexate aferente task-ului care urmează a fi executat [34]. Datorită folosirii operației de remapare a resurselor multiplicate, operația de schimbare a contextului durează un ciclu de ceas indiferent de numărul și dimensiunea regiștrilor de lucru care necesită salvați. În modelul propus, nHSE este un automat de stări finite, fiind parte integrată a procesorului. În alte arhitecturi studiate planificatorul hardware este proiectat ca și componentă externă a procesorului, comunicația cu acesta realizându-se prin magistrale de date și adrese, porturi de intrare/ieșire sau memorie mapată.

nMPRA implementează o linie de asamblare cu 5 etaje pentru păstrarea performanțelor de calcul caracteristice strategiei RISC. Etajele benzii de asamblare sunt următoarele [35]:

- *IF* - Citirea instrucțiunii din memorie;
- *ID* - Decodificarea instrucțiunii;
- *EX* - Etapa de calcul aritmetic sau logic;
- *MEM* - Ciclul de citire/scriere în memoria de date;
- *WB* - Scrierea rezultatelor în fișierul de regiștri sau direct în etajele anterioare.

Fișierul de regiștri este compus din 32 de regiștri a câte 32 biți pentru fiecare context al unui task, fiind special proiectat pentru a garanta comutarea rapidă a task-urilor. Fiecare task suportă o imbricare a apelurilor funcțiilor similară cu cea din arhitecturile de procesor MIPS clasice. nHSE este unitatea funcțională care stă la baza arhitecturii nMPRA. Spre deosebire de alte arhitecturi propuse, planificatorul hardware propus în [4] este parte constituantă a procesorului, fiind controlat prin intermediul instrucțiunilor dedicate acestuia. Așadar, pe frontul crescător al unui ciclu de ceas operează atât blocul de control al căii de date cât și modulul nHSE.

Performanța arhitecturii nMPRA este garantată de operația de comutare a contextelor care este mult mai rapidă decât cea de salvare a regiștrilor pe stivă. În ceea ce privește arhitectura memoriei, varianta folosită pentru validare folosește o memorie dual-port pentru date și instrucțiuni, proiectată on-chip. Așadar, în schema generală a arhitecturii nMPRA memoria de instrucțiuni și date este ilustrată în două blocuri distincte pentru o reprezentare mai clară. Constrângerile datorate necesității izolării spațiale a task-urilor, pot fi satisfăcute prin utilizarea de memorii distincte pentru stocarea codului și a datelor.

În logica de planificare se utilizează un spațiu unificat de priorități pentru întreruperi și task-uri [36]. Această regulă asigură termenele limită de finalizare a execuției task-urilor, oferind un răspuns de timp real la întreruperile externe. Aceasta deoarece task-urile de prioritate mare nu pot fi afectate de întreruperile asignate task-urilor mai puțin prioritare. Întreruperile respectă același regim de execuție ca și task-urile [37] iar activarea sau dezactivarea execuției lor este permisă de nHSE [38]. Sistemul de întreruperi poate fi gestionat prin intermediul instrucțiunilor dedicate planificatorului nHSE. Acesta implementează o regulă de planificare preemptivă bazată pe priorități, ordonând task-urile în ordine crescătoare a priorităților. Planificatorul nu realizează nici un fel de calcul cu privire la timpul de execuție necesar fiecărui task, în afară de componenta *watchdog* care asigură funcția de siguranță a întregului sistem.

Spre deosebire de alte arhitecturi cu planificator hardware, arhitectura nMPRA este o implementare multi-pipeline cu multiplexare de resurse, ceea ce înseamnă că fiecare task are propriul set de regiștri pipeline. De aceea, performanțele arhitecturii nMPRA nu se referă la puterea de calcul ci la timpul de comutare a contextelor task-urilor și la viteza de execuție a operației de planificare. În cazul cel mai defavorabil, aceste operații pot dura trei cicli de ceas, iar când întreruperea este atașată unui task cu prioritate mai mare decât celui aflat în execuție, operația de schimbare a contextelor poate fi îndeplinită într-un singur ciclu de ceas (Figura 4-3).

nMPRA elimină posibilitatea apariției fenomenului de blocare la infinit (înfometare) a proceselor, asigurând totodată izolarea spațială a task-urilor și predictibilitatea sistemului. Acest lucru este datorat unificării într-un singur spațiu de priorități a tuturor întreruperilor și task-urilor din sistem [39]. Un task poate fi planificat și introdus pe banda de asamblare de către nHSE, doar dacă nu există task-uri sau întreruperi mai prioritare [40].

Arhitectura nMPRA este una foarte puternică datorită proprietăților sale:

- Banda de asamblare nu este resetată, ca urmare nu este necesară salvarea respectiv restaurarea contextelor datorită multiplicării resurselor (PC, regiștrii pipeline și fișierul de regiștri);
- Comutarea între task-uri este realizată uzual într-un singur ciclu mașină sau în trei cicli mașină atunci când se lucrează cu instrucțiuni atomice și memoria globală;
- Reacția sistemului la un eveniment extern nu va depăși 1.5 cicli de ceas dacă evenimentul este atașat la un task mai prioritar decât task-ul curent;
- Implementează o instrucțiune puternică prin care un task poate să aștepte diverse tipuri de evenimente [41-43] (eveniment de timp, mutex, întrerupere, două termene limită, etc.);
- Este prevăzută cu un controler distribuit pentru întreruperi prin care întreruperea moștenește prioritatea task-ului [44];
- Îmbunătățește timpul de răspuns la evenimente multiple simultane și întreruperi multiple simultane prin utilizarea codificatoarelor de prioritate și transferul direct către handler-ele de evenimente [45];
- Suportă un planificator static și oferă suport pentru planificarea dinamică a task-urilor.

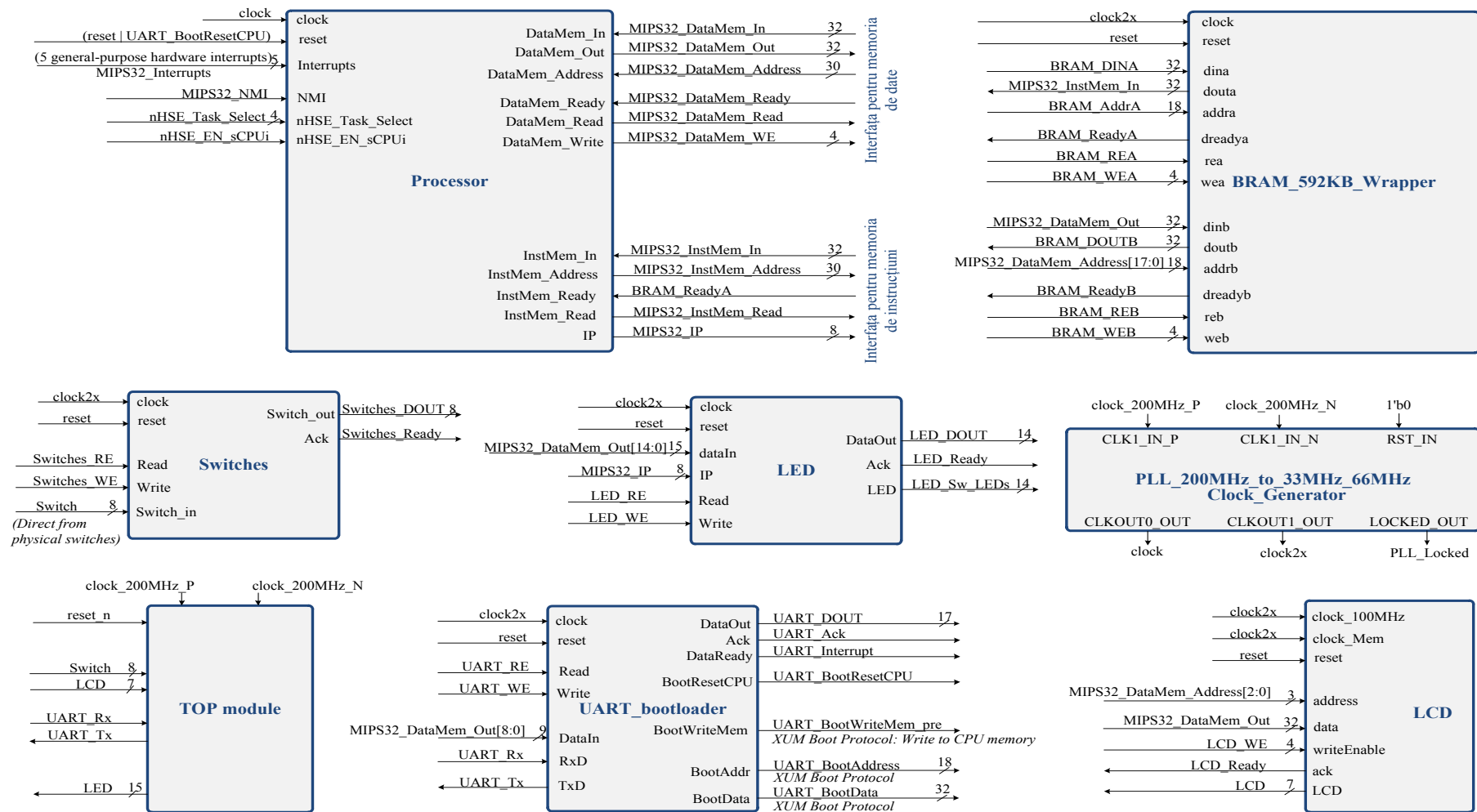


Figura 4-3 Structura proiectului SoC ce conține procesorul nMPRA

În cazul în care programul efectuează un salt la o altă adresă, eficiența benzii de asamblare este micșorată, determinând astfel deteriorarea performanțelor benzii de asamblare prin ignorarea datelor existente în regiștrii din calea de date și încărcarea acestora începând cu prima instrucțiune din noul segment de cod program executat. În Figura 4-4 s-a reprezentat etajul IF (*Instruction Fetch*) împreună cu registrul pipeline IF/ID (*Instruction Fetch/Instruction Decode*). În acest etaj al benzii de asamblare, registrul PC este încărcat cu adresa corespunzătoare instrucțiunii din memoria program ce trebuie extrasă iar mai apoi executată în etajele următoare ale benzii de asamblare. Actualizarea registrului PC este realizată cu una din următoarele adrese, furnizată din etajul curent sau etajul ID:

- *IF_PCAdd4* - ieșirea sumatorului *PC_Add4*;
- *ID_JumpAddress* - 32 biți reprezentând $\{ID_PCAdd4[31:28], Instruction[25:0], 2'b00\}$ pentru instrucțiunile de tip *J* [31];
- *ID_BranchAddress* - adresa de salt condițional $\{14\{immediate[15]\}, immediate, 2'b0\}$;
- *ID_ReadData1_End* – adresa furnizată de ieșirea multiplexorului *IDRsFwd_Mux* din etajul ID.

În funcție de unitatea de control și de detecție a hazardului, se va actualiza adresa următoarei instrucțiuni stocată în registrul PC și se va extrage din memoria program instrucțiunea curentă corespunzătoare semiprocessorului selectat de planificatorul nHSE. Setarea semnalelor de control pentru multiplexoarele *PCSrcStd_Mux* (pentru selecția sursei PC) și *PCSrcExc_Mux* (pentru selecția unei excepții pentru PC) este realizată de unitatea de control a procesorului nMPRA și modulul *CPZero* ce implementează coprocessorul 0. În acest etaj există și sumatorul *PC_Add4* necesar pentru adunarea cu 4 a PC-ului curent, reducând această sarcină unității aritmetice și logice. Astfel, în registrul pipeline IF/ID se va memora instrucțiunea extrasă din memoria program, valoarea PC actuală necesară pentru repornire în cazul unei excepții apărute în etajele următoare și valoarea PC+4 necesară pentru extragerea următoarei instrucțiuni. Semnalele *IF_Stall*, *ID_Stall*, *IF_Exception_Flush* și *IF_Flush* sunt impuse de unitatea de control și modulul *CPZero*, permițând stagnarea și golirea benzii de asamblare în cazul situațiilor de hazard și a excepțiilor.

Registrul pipeline ID/EX stochează datele obținute în urma decodificării instrucțiunii și extragerii operanzilor din fișierul de regiștri și starea liniilor de control necesare în etajele următoare. Așadar, în etajul ID din banda de asamblare se realizează decodificarea instrucțiunii citită din memorie și citirea datelor din fișierul de regiștri. Operanzii citați din fișierul de regiștri vor fi stocați în etajul pipeline următor dacă instrucțiunea este de tipul *R* sau *I*, sau vor fi ignorați cum este cazul instrucțiunilor de salt. În Figura 4-4 sunt ilustrate multiplexoarele *PCSrcStd_Mux* și *PCSrcExc_Mux*, cât și ieșirile pe 32 de biți furnizate de aceste circuite combinaționale.

În etajul pipeline ID sunt proiectați regiștrii de deplasare pentru alinierea în memorie la nivel de cuvânt pe 32 de biți, iar pentru asigurarea lățimii cuvântului de date este proiectată și unitatea de extindere a semnului. După cum putem observa în Figura 4-4, acest etaj conține atât sumatorul necesar pentru calculul adreselor de salt, cât și unitatea de comparare a condiției. Această unitate are drept intrări cei doi operanzi citați din fișierul de regiștri iar la ieșire furnizează condițiile logice destinate unității de control.



În Figura 4-5 este reprezentată schema etajului de execuție a benzii de asamblare împreună cu semnalele de control și date necesare pentru selecția operanzilor unității aritmetice și logice. În etajul de execuție a instrucțiunilor are loc efectuarea operațiilor aritmetice și logice. Rezultatul furnizat de ALU este transmis și memorat în câmpul *M_ALU_Result* din registrul pipeline EX/MEM prin intermediul liniilor *EX_ALU_Result[31:0]*, fiind utilizat ulterior în etajul MEM. Se pot observa de asemenea semnalele de control furnizate de unitatea de control atât pentru ALU cât și pentru multiplexoarele de selecție a celor doi operanzi. În cele ce urmează sunt prezentate numele acestor semnale, structura cât și descrierea detaliată a acestora. Multiplexorul *EXRsFwd_Mux* împreună cu semnalele *EX_RsFwdSel[1:0]* selectează valoarea operandului citit din fișierul de regiștri corespunzător semiprocessorului sCPUi, rezultatul ALU redirectionat din etajul MEM, câmpul *WB_WriteData* redirectionat din etajul WB sau valoarea *EX_RestartPC*. După cum se poate observa și în Figura 4-5, multiplexoarele *EXRtFwdLnk_Mux*, *EXALUImm_Mux* și *EXRtRdLnk_Mux* pot selecta atât valoarea imediată extinsă pe 32 de biți, al doilea operand citit din fișierul de regiștri, cât și câmpurile *EX_Rt* și *EX_Rd*.

Pentru a degreva etajul EX de operațiile decizionale, unitatea de comparate a condiției a fost implementată în etajul pipeline ID. Aceste aspecte trebuie luate în considerare, având în vedere că arhitectura nMPRA se bazează pe principiul remapării contextelor, realizând multiplexarea tuturor resurselor multiplicate. Astfel, se reduce din dimensiunea registrului pipeline ID/EX cât și din logica combinațională necesară implementării instrucțiunilor de salt. Semnalele de control destinate etajelor MEM și WB sunt transmise prin etajul EX și memorate în registrul pipeline EX/MEM, păstrând același format.

După cum putem vedea în Figura 4-6, în etajul pipeline MEM s-a realizat un modul ce implementează controlerul de memorie pentru accesul la memoria de date în scriere sau citire. Unitatea de control dictează operațiile de scriere și citire în memoria de date prin intermediul semnalelor de control *M_MemRead* și *M_MemWrite*, transmise și memorate la fiecare ciclu de ceas o dată cu contextul instrucțiunii pe tot parcursul etajelor benzii de asamblare. Atât validarea datelor cât și adreselor este realizată de unitatea de control, selecția datelor de intrare (*M_ReadData2_PR*, *WB_WriteData*) fiind efectuată de multiplexorul *MWriteData_Mux*. Câmpul *M_MemReadData* ce conține datele citite din memorie va fi stocat în următorul registru pipeline MEM/WB pentru utilizarea acestor date în următorul ciclu de ceas. Calea de date astfel proiectată este ilustrată în figurile 4-4, 4-5 și 4-6.

Conținutul câmpului adresă specificat controlerului de memorie este constituit din rezultatul unității ALU. Acesta reprezintă valoarea citită din fișierul de regiștri și deplasamentul aliniat pe 32 de biți.

În Figura 4-6 se poate vedea organizarea la nivel logic a etajului MEM cât și a registrului pipeline EX/MEM, punând un accent deosebit pe resursele multiplicate. Se pot observa atât semnalele provenite de la unitatea nHSE cât și semnalele *EX_Stall* și *EX_Exception_Flush*. Modificarea ieșirilor de tip registered se realizează la frontul crescător al ceasului *clock*, similar cu toți regiștrii pipeline din procesor, având la bază semnalele de comandă *nHSE_Task_Select[3:0]*.



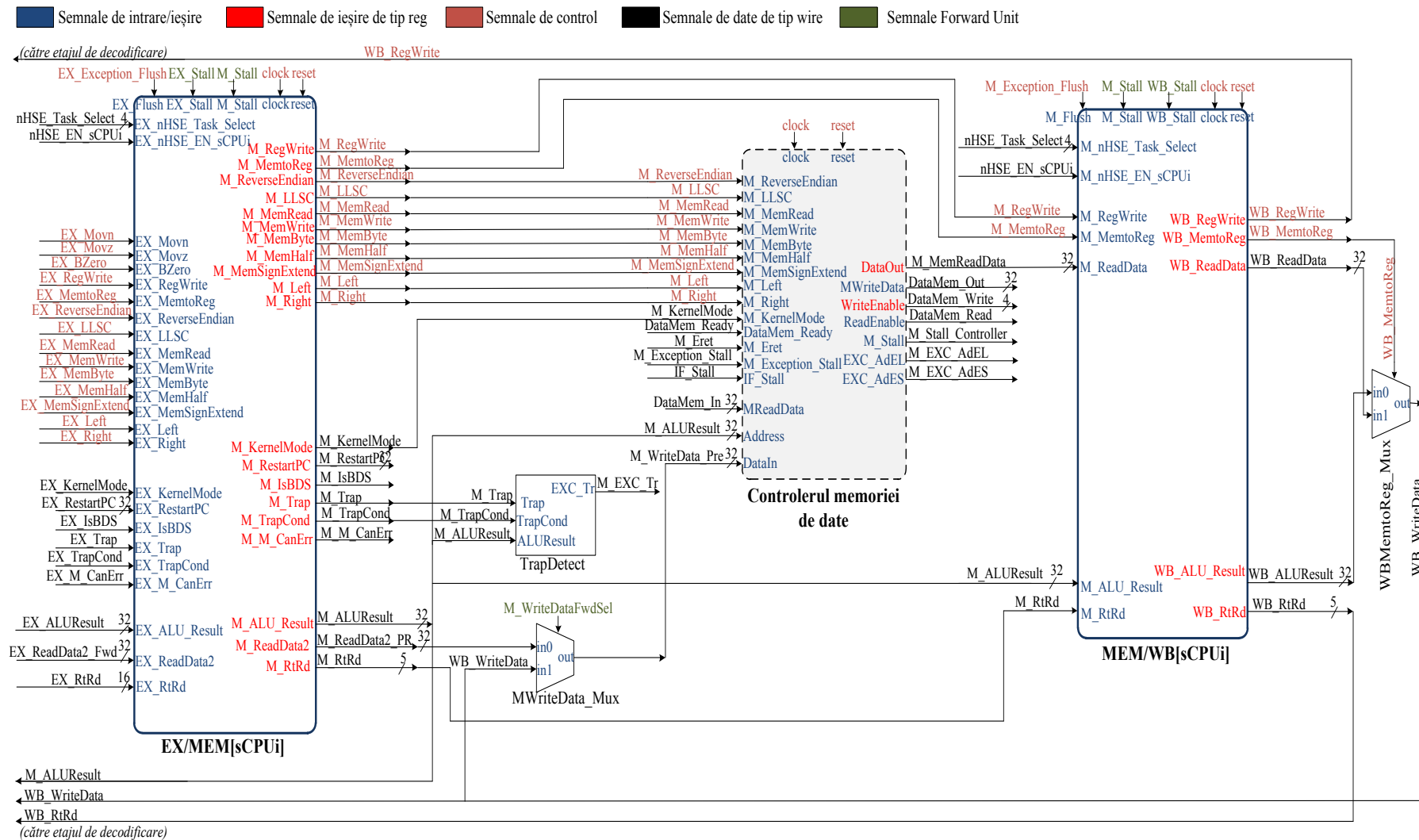


Figura 4-6 Implementarea etajelor pipeline MEM și WB

Procesorul nMPRA folosește o arhitectură de memorie Harward, accesele la date și instrucțiuni fiind realizate într-un spațiu separat de adrese. Pentru modulul *MemControl* atât interfața de date cât și de adrese sunt pe 32 de biți, folosind formatul *Big Endian* sau *Little Endian* în funcție de valoarea parametrului *Big_Endian* setat în fișierul *MIPS_Parameters.v*. nMPRA suportă accese la memorie de tip word, halfword și byte, magistrala memoriei de date fiind una sincronă folosită pentru accesele la memoria RAM on-chip. Ea folosește un număr minim de semnale de control și un protocol simplu pentru a se asigura că memoria de date și instrucțiuni este accesată în scriere într-un singur ciclu de ceas.

Etajul WB realizează scrierea rezultatului în fișierul de regiștri sau în etajele anterioare atunci când unitatea de detecție a hazardului semnalizează apariția unei situații de hazard. Multiplexorul *WBMemtoReg_Mux* din etajul WB este comandat de către unitatea de control prin intermediul semnalului *WB_MemtoReg*. Acest multiplexor realizează selecția regiștrilor *WB_ALUResult* și *WB_ReadData* memorați în registrul pipeline MEM/WB. În funcție de operația aritmetică sau logică, sau de acces la memorie efectuată de instrucțiunea executată, prin intermediul ieșirii *WB_WriteData* acest multiplexor va furniza la ieșire datele necesare.

Unitatea aritmetică și logică implementată în cadrul procesorului nMPRA realizează următoarele operații:

- Operații aritmetice cu semn;
- Operații aritmetice fără semn;
- Operații logice;
- Operații pentru calculul deplasamentului din cazul instrucțiunilor *lw* și *sw*.

Modulul ALU este degrevat de operațiile decizionale pentru instrucțiunile de salt condiționat deoarece acestea au fost implementate în etajul pipeline ID de către unitatea de testare a condiției. Astfel, ALU nu face calculul adreselor instrucțiunilor de salt condiționat sau necondiționat deoarece această sarcină le revine sumatoarelor din etajele pipeline IF și ID.

Datele reprezentând rezultatul operației efectuate în etajul precedent (*M_ALUResult*), cât și semnalele de control *M_RegWrite* și *M_MemtoReg* utilizate în etajul pipeline WB sunt ilustrate în Figura 4-6.

În implementarea nMPRA unitatea de control reprezintă suportul pentru o arhitectură de procesor flexibilă și performantă. În Figura 4-7 sunt prezentate intrările și ieșirile modulului *Control*. Semnalele pe care le furnizează la ieșire reprezintă linii de control și excepții de tip *reg* pentru calea de date cât și operația transmisă unității aritmetice și logice. Furnizarea semnalelor de control pentru calea de date are loc în etajul pipeline ID. Modulul *Control_Unit* realizează următoarele operații:

- Setează principalele semnale de control necesare căii de date, având la bază registrul *OpCode*;
- Setează semnalele de control pentru situațiile de hazard;
- Efectuează asignările necesare unității aritmetice și logice din etajul pipeline EX;
- Detecția ramificațiilor (opțiunile sunt bazate pe excluderea reciprocă);
- Setează un indicator pentru a semnaliza faptul că următoarea instrucțiune este un *Branch Delay Slot* ($assign\ NextIsDelay = Datapath[15] \mid Datapath[14]$);
- Setează semnalele de control pentru *Move Conditional*;

- Validează semnalele corespunzătoare operațiilor Mfc0 și Mtc0;
- Setează semnalele necesare pentru accesele coprocesoarelor 1, 2 și 3;
- Semnalizează excepțiile găsite în etajul ID;
- Setează accesele nealiniat la memorie (*lwl*, *lwr*, *swl* și *swr*).

Modulul *Control_Unit* reprezintă unitatea de control a procesorului nMPRA. Această unitate setează biții de control din calea de date pentru fiecare instrucțiune citită din memorie. Aceste semnale sunt reprezentate de câmpurile instrucțiunii executate, rezultatele unității de testare a condiției și semnalul *ID_Stall* furnizat de unitatea de detecție a hazardului. Astfel, semnalele de control însoțesc instrucțiunea prin fiecare etaj pipeline, determinând toate stările și operațiile necesare pe care procesorul trebuie să le execute secvențial pentru fiecare etaj pipeline. În funcție de semnalele de stare, această unitate setează biții de control necesari execuției fiecărei instrucțiuni.

Procesoarele MIPS execută instrucțiunea de salt sau ramificație și cea din *Delay Slot* ca unitate indivizibilă. Dacă există o excepție ca rezultat al execuției instrucțiunii din *Delay Slot*, instrucțiunea jump sau branch nu este executată iar excepția apare ca fiind cauzată de către instrucțiunea de salt sau ramificație. În cadrul procesorului nMPRA, toate operațiile de salt și ramificație determină execuția instrucțiunii din *Branch Delay Slot*, indiferent dacă ramificația se efectuează sau nu. Excepțiile referitoare la instrucțiunile de salt fac parte din grupul instrucțiunilor *Branch Likely*, și nu sunt implementate în modulul *Control*. Pe lângă acest lucru, există un grup de instrucțiuni de salt condiționat, denumite *Branch Likely*, pentru care instrucțiunea următoare ce se află în așa-zisul *Delay Slot* este executată numai dacă ramificația are loc. Chiar dacă instrucțiunile *Branch Likely* sunt incluse în specificațiile MIPS, software-ul este încurajat să evite aceste instrucțiuni deoarece ele vor fi scoase din viitoarele revizii ale arhitecturii MIPS. Așadar, instrucțiunile de salt condiționat *Branch Likely* (BEQL, BGEZALL, BGEZL, BGTZL, BLEZL, BLTZALL, BLTZL, BNEL) nu au fost implementate în cadrul procesorului nMPRA. Pentru calea de date corespunzătoare procesorului nMPRA toate semnalele sunt active pe 1 logic.

MIPS pune la dispoziția utilizatorului un sistem de coprocesoare pentru extensia de funcționalități ale procesorului de bază. La dispoziția utilizatorului poate fi coprocesorul 2 (COP2). Extensiile specifice aplicațiilor MIPS (ASEs) și instrucțiunile definite de utilizator (UDIs) reprezintă alte două aspecte importante. Astfel, arhitecturile MIPS32 și MIPS64 asigură un suport solid pentru extensii specifice aplicațiilor utilizator. Ca și extensii opționale la arhitectura de bază, acestea nu încarcă fiecare implementare a arhitecturii cu instrucțiuni sau capacități care nu sunt necesare decât pentru o implementare specifică. Arhitecturile MIPS32 și MIPS64 permit instrucțiuni definite de utilizator specifice pentru realizarea fiecărei implementări, acest lucru reprezentând un suport adițional pentru ASEs. Astfel, câmpurile *Special 2* și *COP2* sunt rezervate pentru capacitatea definită de fiecare implementare. Utilizând MIPS32 ISA s-au implementat noile instrucțiuni specifice planificatorului nHSE, prezentarea mai pe larg a acestora regăsindu-se în specificațiile procesorului nMPRA, disponibil în format electronic [46]. Utilizând instrucțiunile destinate planificatorului nHSE se poate administra sistemul general de întreruperi, inclusiv gestiunea individuală a acestora. Aceste instrucțiuni dedicate sunt decodificate independent față de instrucțiunile program, având la bază informațiile din registrul pipeline IF/ID.



5. Contribuții privind implementarea arhitecturii nMPRA și nHSE în FPGA

Procesorul nMPRA pe 32 de biți este în conformitate cu MIPS32 Release 1 ISA. Această implementare de procesor a fost proiectată și realizată utilizând circuitul FPGA *xc7vx485tffg1761-2* produs de Xilinx. Implementarea curentă include un procesor de sine stătător, precum și un proiect complet *System on Chip* dezvoltat și testat pe kit-ul de dezvoltare cu FPGA Virtex-7 prezentat în Figura 5-1.

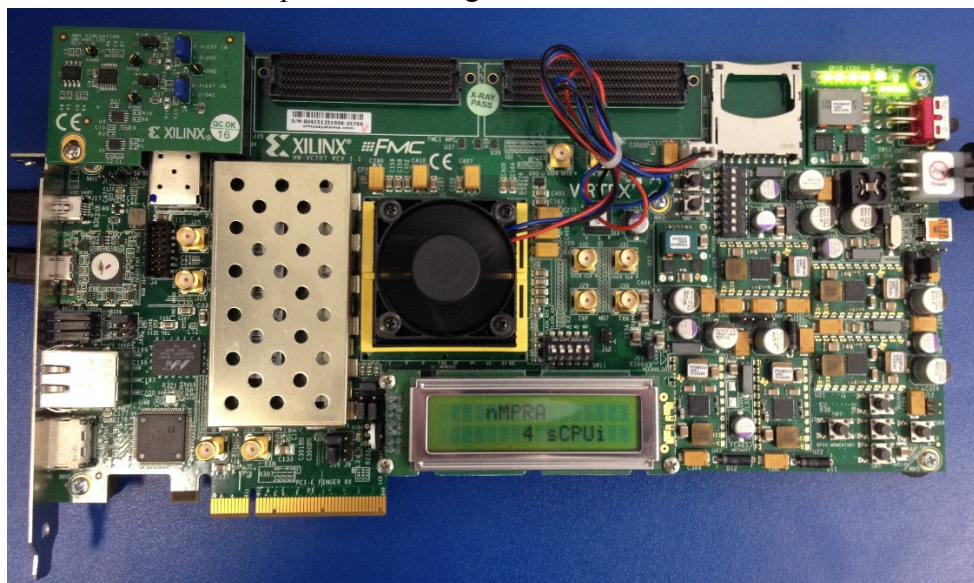


Figura 5-1 Kit-ul de dezvoltare VC707 produs de Xilinx

Avantajele majore aduse de această platformă de dezvoltare bazată pe tehnologia logicii programabile sunt garantate de o înaltă performanță raportată la consumul de energie, integrarea folosind tehnologia pe 28 nm, performanțe DSP (*Digital Signal Processing*) și lățime de bandă I/O. Acest circuit FPGA dispune de 485760 *Logic Cells*, maxim 8175 *Distributed RAM* (Kb), 1030 *Block RAM/FIFO w/ ECC* (36 Kb fiecare), 2800 *DSP Slices*, un modul *Analog Mixed Signal* (AMS) / XADC precum și alte resurse importante [47-51].

5.1.Descrierea modulelor procesorului nMPRA implementate în Verilog

Un sistem digital este în primul rând o multitudine de circuite combinaționale și secvențiale conectate împreună. Verilog este un limbaj de descriere hardware dezvoltat inițial de către *Design Automation* în 1984, devenind un standard industrial datorită simplității sale, având o structură asemănătoare cu cea a limbajului de programare C și timpi de proiectare foarte rapizi [52], [53]. Structurile de intrare/ieșire sunt de asemenea mai mult sau mai puțin similare, cu toate că trebuie să se aibă în vedere faptul că Verilog este un instrument de proiectare hardware și nu un instrument de proiectare software .

Deși este o arhitectură cu multiplexare de resurse, cu schimbări minore asupra proiectului SoC, acesta se poate porta pe diverse platforme hardware. Pentru aceasta este necesară modificarea modului de ceas, BRAM, UART, I/O și a fișierului de constrângeri .xdc.

În varianta originală a proiectului, fără multiplicarea resurselor și planificatorul hardware integrat nHSE, procesorul MIPS32 utilizează aproximativ *1800 Slice Registers* și *4000 LUTs*, utilizând kit-ul de dezvoltare Virtex-7 cu circuitul FPGA xc7vx485tffg1761-2. Întregul SoC utilizează aproximativ *2700 Slice Registers* și *5100 LUTs* pe aceeași placă de dezvoltare.

Implementarea procesorului nMPRA de sine stătător cu 16 semiprocresoare, folosind același circuit FPGA xc7vx485tffg1761-2, utilizează aproximativ *15988 Slice Registers (FFs + Latch)* și *29495 LUTs (LUTs as Logic + LUTs as Memory)*. Trebuie de precizat faptul că planificatorul hardware de timp real conține și suportul necesar pentru planificarea dinamică, unitatea nHSE fiind parte integrată a procesorului.

În cele ce urmează sunt enumerate proprietățile procesorului nMPRA:

- Bandă de asamblare cu cinci etaje pipeline cu detecția hazardului și redirecționarea datelor;
- Execuția în ordine și *single-issue* a instrucțiunilor implementate respectând arhitectura setului de instrucțiuni MIPS32 Release 1 ISA;
- Arhitectură Harward cu porturi separate pentru instrucțiuni și date;
- Implementarea tuturor instrucțiunilor MIPS32, inclusiv înmulțirea hardware și împărțirea, operații atomice pentru load linked/store condițional și instrucțiuni loads/stores nealiniat;
- Implementarea completă a Coprocesorului 0 permițând întreruperi ISA-compliant, excepții și modulele User/Kernel;
- Împărțirea hardware este simplă, multiciclu și rulează asincron față de banda de asamblare permițând de cele mai multe ori mascarea acestei latențe;
- Interfața memoriei este separată de procesor din motive de flexibilitate cu posibilitatea de conectare a diferitelor module RAM;
- În mod implicit, implementarea hardware a procesorului este *Big Endian*. Astfel, adresa celui mai semnificativ octet este adresa cuvântului, suportând și *Little Endian* pentru modul User;
- Adrese parametrizate pentru vectorii de întreruperi și excepții implementând adrese de delimitare între regiunile user/kernel;
- Comentarea vastă a codului sursă și a întregului proiect SoC compus din module separate;
- Cod independent de furnizor;
- Proiectul nu dispune de unități precum *Memory Management Unit* (MMU), *Memory Protection Unit* (MPU) sau *Floating Point Unit* (FPU).

Top.v reprezintă fișierul situat la nivelul cel mai superior din proiectul curent. Este de asemenea cunoscut ca și un *motherboard* care conectează module precum procesorul, memoria, semnalele de ceas și dispozitivele I/O. Toate intrările și ieșirile, cum ar fi semnalele de ceas sau pinii de transmisie și recepție UART, trebuie să corespundă cu pinii circuitului FPGA folosit.

Fișierul Verilog *Processor.v* împreună cu instanțierea modulelor din interiorul acestui fișier creează un procesor MIPS32 complet. Modulul top-level este *Processor*, iar interfața acestuia constă din cinci întreruperi hardware de uz general, o întrerupere hardware nemascabilă, 8 întreruperi pentru diagnoză ce pot fi eliminate și o interfață cu memoria dual-

port implementată on-chip folosind *IP Block Memory Generator 8.3 (Rev. 1)*, atât pentru instrucțiuni cât și pentru date. Modulul *Processor* este cel mai important modul instanțiat din proiect. Acest fișier conține în cea mai mare parte a sa, instanțierea și realizarea legăturilor dintre blocurile de bază ale procesorului conform schemelor de proiectare. Acest modul include foarte puțină logică, deși conține cele mai multe module instanțiate.

Fișierul de regiștri pentru procesorul nMPRA conține $NR_TASKS * 32$ regiștri de uz general a câte 32 de biți fiecare și două porturi de citire a acestora în funcție de task-ul selectat. În Figura 5-2 este reprezentat fișierul de regiștri pentru procesorul nMPRA într-o configurație cu 4 sCPUi. Registrul 0 este setat întotdeauna ca având valoarea $32'h00000000$. Scrierea în fișierul de regiștri se efectuează în funcție de semiprocessorul selectat de către planificator. La frontul pozitiv al ceasului datele furnizate de intrarea *WriteData* (32 de biți) sunt scrise în registrul index *WriteReg* (5 biți) la comanda semnalului *RegWrite*. Citirea combinațională din fișierul de regiștri are la bază task-ul planificat *ID_nHSE_Task_Select*.

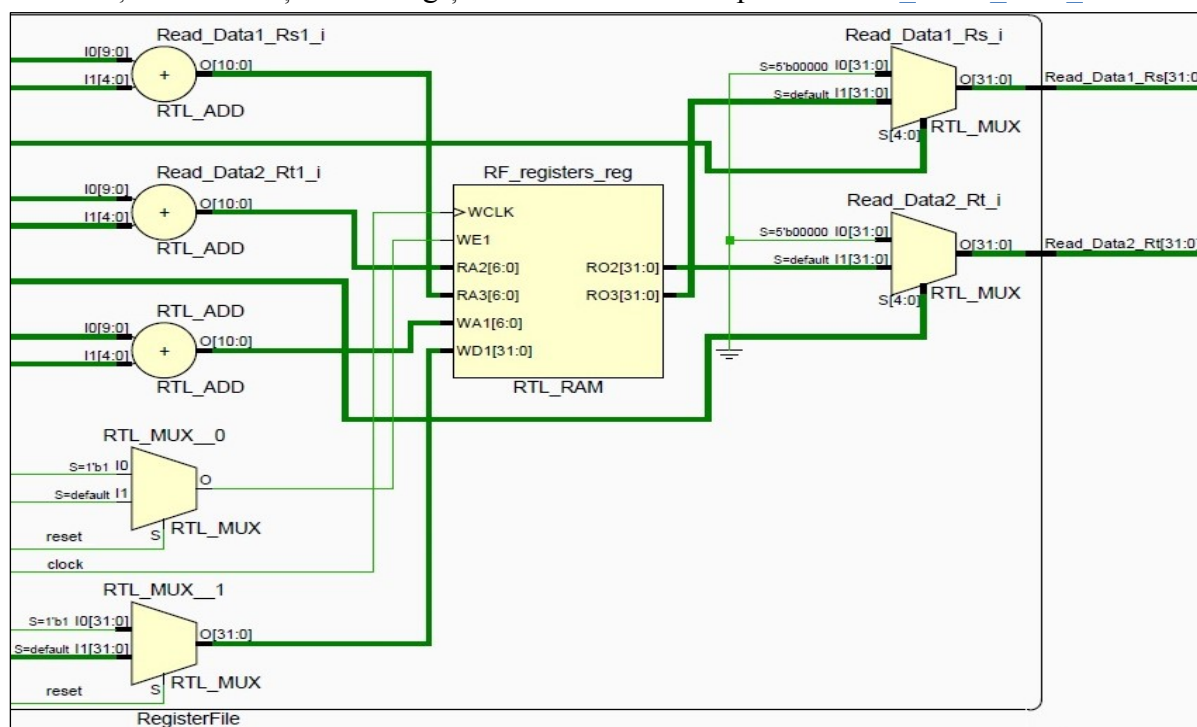


Figura 5-2 Fișierul de regiștri al procesorului nMPRA

Arhitectura descrisă în aceasta teză de doctorat nu folosește conceptul de stivă folosit în procesoarele existente, doar funcționalitatea pentru apelul funcțiilor imbricate și a ordinii acestora. Pentru a asigura o comutare rapidă a contextelor arhitectura nMPRA se bazează pe multiplicarea setului de regiștri de uz general pentru fiecare sCPUi. Astfel, fiecare semiprocessor implementează 32 de regiștri pe 32 de biți reprezentând un banc de regiștri, iar totalitatea bank-urilor compun fișierul de regiștri al arhitecturii nMPRA. Fiecărui semiprocessor îi corespunde un banc de regiștri iar salvarea contextelor funcțiilor și transmiterea parametrilor se realizează similar ca în cazul procesoarelor MIPS clasice.

5.2.Implementarea și testarea procesorului nMPRA

Datorită complexității arhitecturii nMPRA, validarea acesteia a fost împărțită în trei părți și anume:

- Validarea căii de date și a unităților funcționale:
 1. Verificarea capacității de funcționare a unității de control;
 2. Testarea implementării regiștrilor PC, bancurilor din fișierul de regiștri și regiștrilor pipeline pentru fiecare sCPUi;
 3. Testarea sumatoarelor simple, unităților de deplasare, de extindere a semnului și a multiplexoarelor;
 4. Validarea operațiilor realizate de unitatea aritmetică și logică, inclusiv operația de împărțire multi-ciclu;
 5. Unitatea de detecție a hazardului și cea de redirectionare a datelor;
- Validarea planificatorului de timp real nHSE;
 1. Implementarea unității nHSE și a instrucțiunilor dedicate planificatorului hardware integrat ce au fost realizate în COP2;
 2. Verificarea corectitudinii operației de comutare a regiștrilor pipeline și a bancurilor din fișierul de regiștri pentru un sCPUi generat de planificatorul nHSE;
 3. Testarea datelor reprezentând contextele task-urilor salvate în regiștrii pipeline, fișierul de regiștri și PC, inclusiv verificarea transferului de date între COP0 și COP2 chiar și în prezența hazardurilor;
 4. Implementarea și testarea mecanismului de sincronizare și comunicație inter-task;
- Testarea procesorului nMPRA în cadrul proiectului SoC:
 1. Testarea proiectului SoC implementat pe placa de evaluare Virtex-7 prezentată în ANEXA I;
 2. Testarea driverelor și a protocolului de boot;
 3. Îmbunătățirea comunicației UART.

5.2.1. Validarea căii de date și a unităților funcționale

În această teză de doctorat a fost descrisă și validată arhitectura de procesor nMPRA utilizând placa de dezvoltare Virtex7, fără a insista asupra descrierii întregului proiect SoC. A fost acordată o deosebită atenție planificatorului de timp real nHSE, codificatorului de priorități pentru evenimente și întreruperi, îmbunătățirii predictibilității de execuție prin eliminarea parțială sau totală a hazardurilor din banda de asamblare și minimizarea jitter-ului pentru schimbarea contextelor task-urilor [54-56]. În Figura 5-3 sunt prezentate semnalele de ceas *clock* (33MHz) și *clock_mem* (165MHz). Putem urmări semnalul *reset_n* și semnalul de ceas diferențial de 200MHz (*clock_200MHzP*, *clock_200MHzN*) al kit-ului de dezvoltare Virtex-7 produs de Xilinx. Cu ajutorul acestui semnal și al IP-ului *Clocking Wizard 5.2 (Rev. 1)* obținem semnalul de ceas *clock* al procesorului nMPRA, acesta fiind de 33MHz.



Figura 5-3 Semnalele de ceas necesare procesorului nMPRA și memoriei implementate on-chip

Pentru sincronizarea cu memoria program implementată on-chip, semnalele utilizate în cazul operațiilor de citire și scriere sunt modificate pe frontul pozitiv al semnalului *clock_mem*, pe parcursul implementării și testării procesorului nMPRA folosindu-se diverse frecvențe pentru acest semnal. În cazul validării instrucțiunilor MIPS implementate de arhitectura nMPRA, s-au urmărit formele de undă obținute în urma simulării și cele obținute în urma depanării on-chip cu analizorul ChipScope. Pentru testarea procesorului în cazul hazardului de date s-a folosit un program scris în cod mașină, verificând astfel calea de date, unitatea de detecție a hazardului și cea de redirecționare a datelor. O atenție deosebită a fost acordată acestei etape, deoarece trebuie garantată integritatea acestor semnale chiar și atunci când în prezența unui hazard are loc o schimbare de context, este executată o instrucțiune dedicată planificatorului nHSE sau este tratată o întrerupere.

Tabelul 5-1 prezintă instrucțiunile executate de procesor în Figura 5-4, putându-se observa extragerea instrucțiunii *0x25081011* din memoria program. La momentul de timp marcat de cursor se pot vedea semnalele corespunzătoare adresei *addra[17:0]*, semnalul de activare a citirii *rea*, semnalul *dreadya* ce indică validitatea datelor cât și instrucțiunea citită din memorie *douta[31:0]*.

Tabelul 5-1: Codul executat de procesor în diagrama din Figura 5-4

Descriere	Cod aplicație
Secțiune de cod pentru validarea extragerii instrucțiunilor din memoria program.	3c080000, // 0011 11_00 000_0 1000_0000 0000 0000 0000
	// <i>lui, R[rt=8]={imm, 16'b0}, Load Upper Imm.</i>
	25081011, // 0010 01_01 000_0 1000_0001 0000 0001 0001
	// <i>addiu, R[rt=8]=R[rs=8]+SignExtImm</i>
	3c090000, // 0011 11_00 000_0 1001_0000 0000 0000 0000
	// <i>lui, R[rt=9]={imm, 16'b0}, Load Upper Imm.</i>

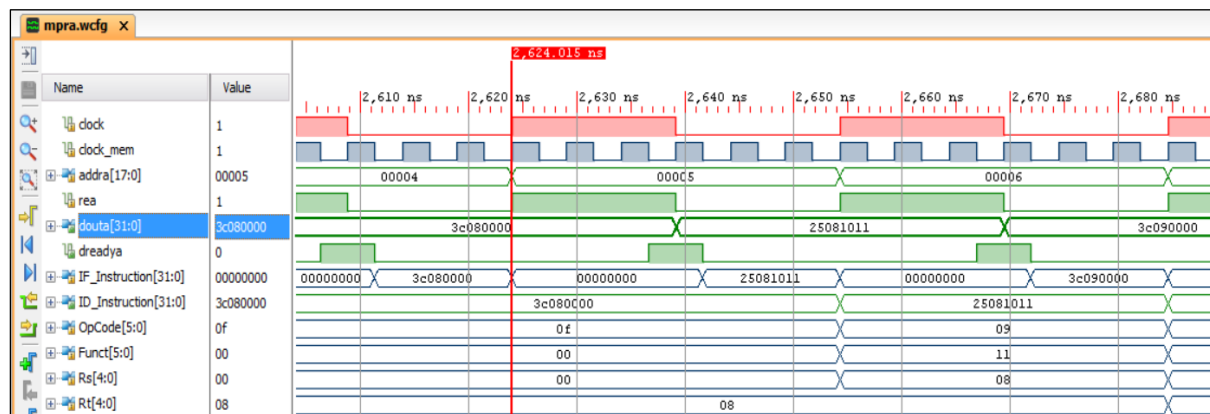


Figura 5-4 Semnalele necesare pentru extragerea instrucțiunilor MIPS din memoria program

În ANEXA II [Figura 1] este reprezentată situația în care în banda de asamblare există un hazard iar prin momentul de timp 18.563ns marcat de cursor sunt indicate datele conținute în regiștrii procesorului, acestea fiind salvate pe parcursul schimbării de context. Este prezentată starea semiprocessorului sCPU0 înainte de schimbarea de context și restaurarea acestuia când execuția sa este reluată. De remarcat faptul că, chiar și în prezența hazardului apărut în banda de asamblare, multiplexarea resurselor multiplicare nu a afectat execuția instrucțiunilor, prin adăugarea unor timpi suplimentari în astfel de cazuri. În aceeași figură se poate observa implementarea regiștrilor PC cât și a semnalelor de selecție provenite de la modulul nHSE. O dată ce a fost calculat noul registru PC cu ajutorul sumatorului simplu *PC_Add4*, avem disponibilă și adresa următoarei instrucțiuni [57], [58]. În procesorul nMPRA avansarea

datelor se realizează prin intermediul modului de redirectionare, indisponibilitatea datelor necesare determinând introducerea de Nop-uri și reducând implicit performanțele benzii de asamblare.

În Figura 5-5 sunt prezentate intrările și ieșirile unității de testare a condiției pentru două intrări a câte 32 de biți și cinci biți de ieșire relativi condițiilor necesare unității de control. Tot în acest etaj este proiectat și un sumator pentru calculul adresei de salt, valorile furnizate de aceste module (*ID_JumpAddress*, *ID_BranchAddress*) fiind disponibile pentru selecție la intrările multiplexorului *PCSrcStd_Mux*.

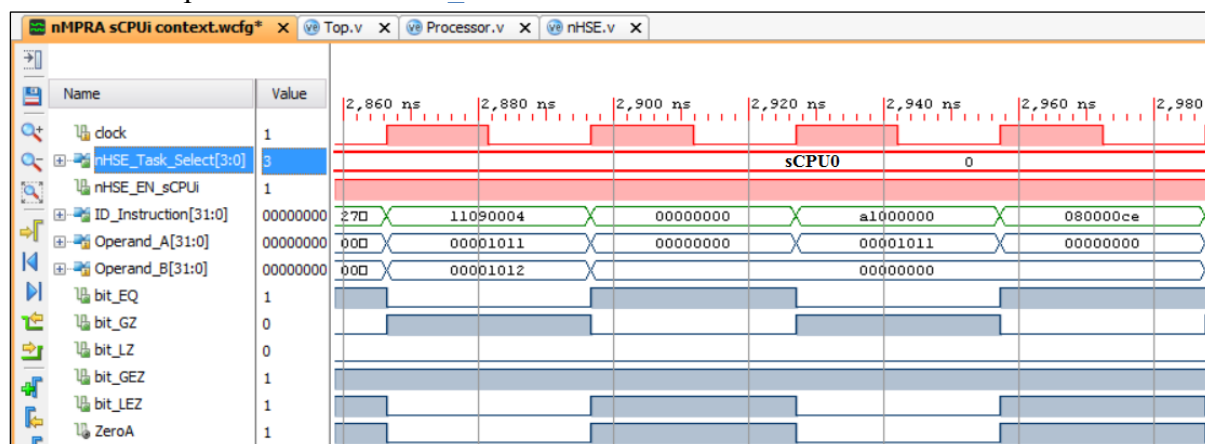


Figura 5-5 Semnalele corespunzătoare unității de testare a condiției executând codul din Tabelul 5-2

Tabelul 5-2: Codul executat de procesor în diagrama din Figura 5-5

Descriere	Cod aplicație
Secțiune de cod pentru testarea unității de testare a condiției.	11090004, // 0001 00_01 000_0 1001_ 0000 0000 0000 0100 // <i>beq, if(R[rs=8]==R[rt=9]) PC = PC+ BranchAddr</i> 00000000, // <i>nop</i> a1000000, // 1010 00_01 000_0 0000_ 0000 0000 0000 0000 // <i>sb, M[R[rs=8]+SignExtImm](7:0)=R[rt=0](7:0), Store Byte</i> 080000ce, // 0000 10_00 0000 0000 0000 0000 1100 1110 // <i>j, PC=JumpAddr, Jump</i>

În ANEXA II [Figura 2] sunt prezentate o parte din semnalele conținute în registrul pipeline ID/EX, acesta fiind cel mai mare consumator de resurse dintre regiștrii pipeline. În ANEXA II [Figura 3] se pot observa operanzii furnizați unității aritmetice și logice, registrul *ID_AluOp* cât și rezultatul operației efectuate necesar în etajele pipeline MEM și WB. Transmita și memorarea semnalelor de control prin calea de date se efectuează concomitent cu datele necesare execuției operației dictate de opcod-ul instrucțiunii, garantând astfel consistența contextelor pentru o eventuală schimbare a sCPUi-ului selectat. Executând codul încărcat prin intermediul fișierului *Boot.coe*, se va testa calea de date observând formele de undă corespunzătoare. În Figura 5-6 este reprezentată schema bloc pentru execuția instrucțiunii *0x24420001 (addiu r2, r2, SignExtImm)*, rezultând două cazuri de hazard corespunzătoare momentelor de timp marcate de cursoarele C3 și C4 din Figura 5-7. După cum se poate observa, caracteristicile formelor de undă obținute cu simulatorul Vivado corespund cu logica implementată la nivelul programului de descriere hardware Verilog. În funcție de situația apărută, redirectionarea datelor în etajul de decodificare sau execuție se realizează utilizând căile *Fwd_ex1* și *Fwd_ex2*. Pentru aceasta a fost necesară modificarea unității de control deoarece registrul care este redirectionat este unul care provine de la coprocesorul 2.

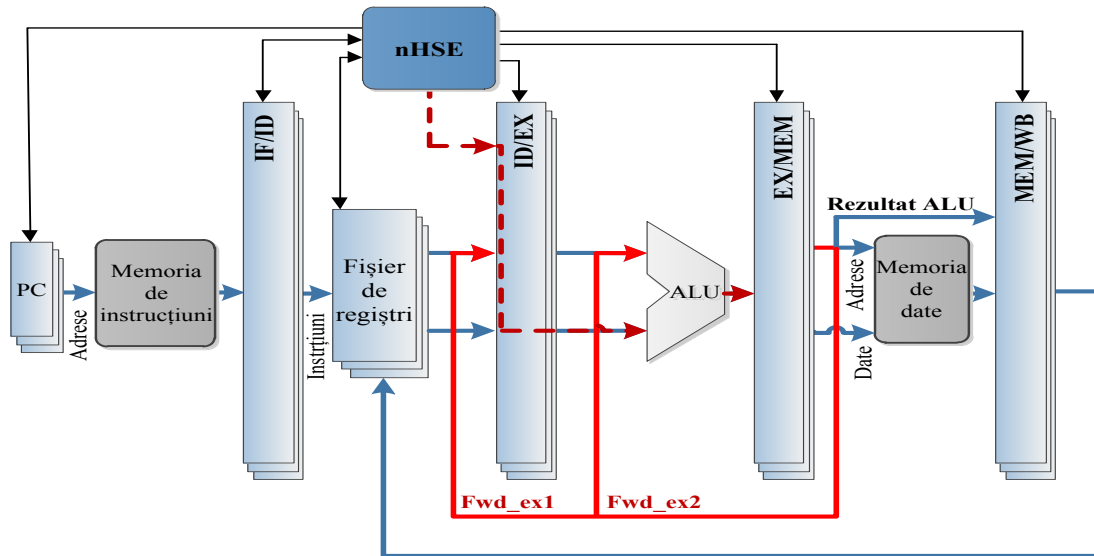


Figura 5-6 Redirecționarea datelor în situațiile de hazard prezentate în Figura 5-7

Figura 5-7 prezintă propagarea semnalelor de control și redirecționarea datelor din etajul MEM, codul executat fiind descris în prima parte din Tabelul 5-3. Se poate observa operația de salvare a datelor în cazul în care este copiat un registru din nHSE (COP2) într-un registru de uz general al coprocesorului 0 la nivelul semiprocessorului sCPU0 activat prin intermediul semnalelor *nHSE_Task_select[3:0]* și *nHSE_EN_sCPUi*.

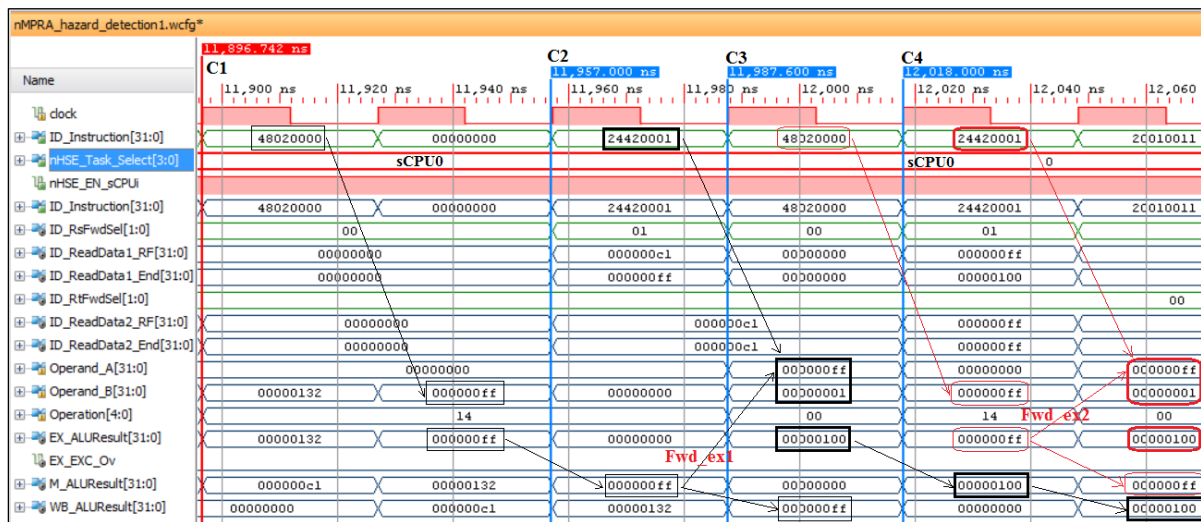


Figura 5-7 Redirecționarea datelor în cazul unei situații de hazard

În cazul execuției instrucțiunilor furnizate de *ID_Instruction[31:0]*, se va testa funcționalitatea unității de detecție a hazardului. Cursorul C1 indică instrucțiunea *0x48020000* ce transferă valoarea *0x000000FF* din registrul de monitorizare *mrTEVi[sCPU0]* (COP2) în registrul r2 din fișierul de regiștri (COP0), în timp ce rezultatul adunării corespunzătoare instrucțiunii MIPS *0x24420001* este vizibil accentuat, acesta fiind transmis regiștrilor *M_ALUResult_reg[0][31:0]* și *WB_ALUResult_reg[0][31:0]* prin intermediul semnalelor *M_ALUResult* și *WB_ALUResult*. În cazul execuției instrucțiunii *0x24420001*, la momentul de timp marcat de cursorul C2 semnalele de control

$ID_RsFwd_Sel[1:0]$ și $ID_RtFwd_Sel[1:0]$ selectează operandii necesari pentru operația de adunare.

Tabelul 5-3: Secțiuni de program executate pentru testarea situațiilor de hazard (Figura 5-7) și a operației de schimbare a contextelor (Figura 5-8)

Descriere	Cod aplicație
Instrucțiuni redată în diagrama din Figura 5-7. Secvența de cod este folosită pentru testarea situațiilor de hazard apărute în banda de asamblare.	<pre> //--- sCPU0 --- 48020000, //0100 10_00 000_0 0010_0000000000000000 //instrucțiunea <i>stmr</i> copie din registrul <i>mrTEVi[sCPU0]</i> în registrul r2 din GPR COP0 00000000, //nop 24420001, //0010 01_00 010_0 0010_0000 0000 1100 0001 //addiu, R[rt=2]=R[rs=2]+SignExtImm 48020000, //0100 10_00 000_0 0010_0000000000000000 //instrucțiunea <i>stmr</i> copie din registrul <i>mrTEVi[sCPU0]</i> în registrul r2 din GPR COP0 24420001, //0010 01_00 010_0 0010_0000 0000 1100 0001 // addiu, R[rt=2]=R[rs=2]+SignExtImm 20010011, //0010 00_00 000_0 0001_0000 0000 0001 0001 //addi, R[rt=1]=R[rs=0]+SignExtImm (=0011) </pre>
Instrucțiuni redată în formele de undă din Figura 5-8.	<pre> 20010011, //0010 00_00 000_0 0001_0000 0000 0001 0001 //addi, R[rt=1]=R[rs=0]+SignExtImm (=0011) 00000000, //nop 48C10000, //010010_00110_00001_0000000000000000 //instrucțiunea <i>movcr</i> copie registrul r1 GPR COP0 în registrul <i>crTRI[sCPU0]</i> COP0 //realizează schimbarea contextelor dacă sCPU0 nu are evenimente validate și active //---Schimbare context sCPU0->sCPU1--- </pre>

În Figura 5-8 este prezentată situația scrierii valorii $0x00000011$ în registrul pipeline EX/MEM și mai apoi în fișierul de regiștri. Se pot observa formele de undă obținute pentru etajul pipeline WB, ceea ce este important de remarcat este corespondența dintre datele (WB_ALU_Result) scrise în fișierul de regiștri ($RF_Registers$) și câmpul corespunzător locației de scriere specificat de WB_RtRd . Semnalul $WB_MemtoReg$ reprezintă linia de control pe un bit necesară pentru selecția datelor trimise în bancul corespunzător fișierului de regiștri prin intermediul multiplexorului $WBMemtoReg_Mux$.

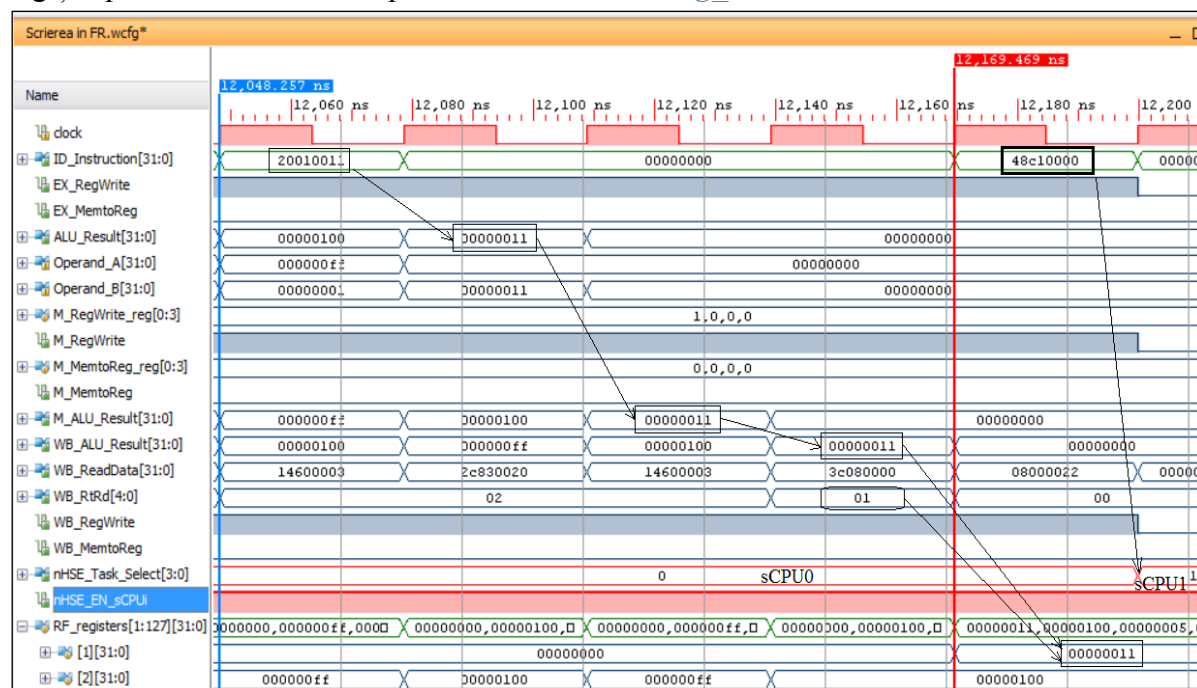


Figura 5-8 Scrierea în fișierul de regiștri și schimbarea contextelor dintre sCPU0 și sCPU1

5.2.2. Planificatorul hardware integrat - nHSE

Această secțiune prezintă și analizează datele finale obținute în urma sintetizării, implementării și testării procesorului nMPRA, utilizând kit-ul de dezvoltare cu FPGA Virtex-7. Prin intermediul acestor teste realizate în hardware s-a dorit testarea în practică a elementelor teoretice prezentate în capitolul 4, inclusiv modelele și algoritmi de planificare pentru STR, incluzând și proiectul SoC. În cele ce urmează va fi prezentată unitatea de planificare integrată nHSE, implementată utilizând limbajul de descriere hardware Verilog. Planificatorul procesorului nMPRA este modulul central al acestei arhitecturi.

După cum a fost explicat în capitolul anterior, schimbarea contextelor task-urilor nu se bazează pe salvarea regiștrilor pe stivă sau pe salvarea într-o memorie internă sau externă, ci folosind principiul remapării resurselor multiplexate. Programul care rulează pe un sCPUi poate fi un task dintr-o aplicație multitasking. Modulul nHSE are rolul de a activa la un moment dat doar un singur sCPUi din cele n . Dacă la un moment dat un sCPUi este oprit de către nHSE și este activat alt sCPUi, toată informația specifică programului care se rula pe sCPUi-ul oprit este conservată datorită multiplicării resurselor precum PC, fișierul de regiștri și regiștrii pipeline. Trecerea de la un sCPUi la altul nu necesită nici o salvare de context și nici o ștergere a conținutului regiștrilor pipeline. Dacă fiecare sCPUi rulează un task, rezultă că și comutarea de la un task la altul se face foarte rapid. Aceasta determină o operație de schimbare de context foarte rapidă, deoarece fiecare sCPUi va rula un task i . În Figura 5-9.a este prezentată schema bloc a planificatorului hardware integrat nHSE.

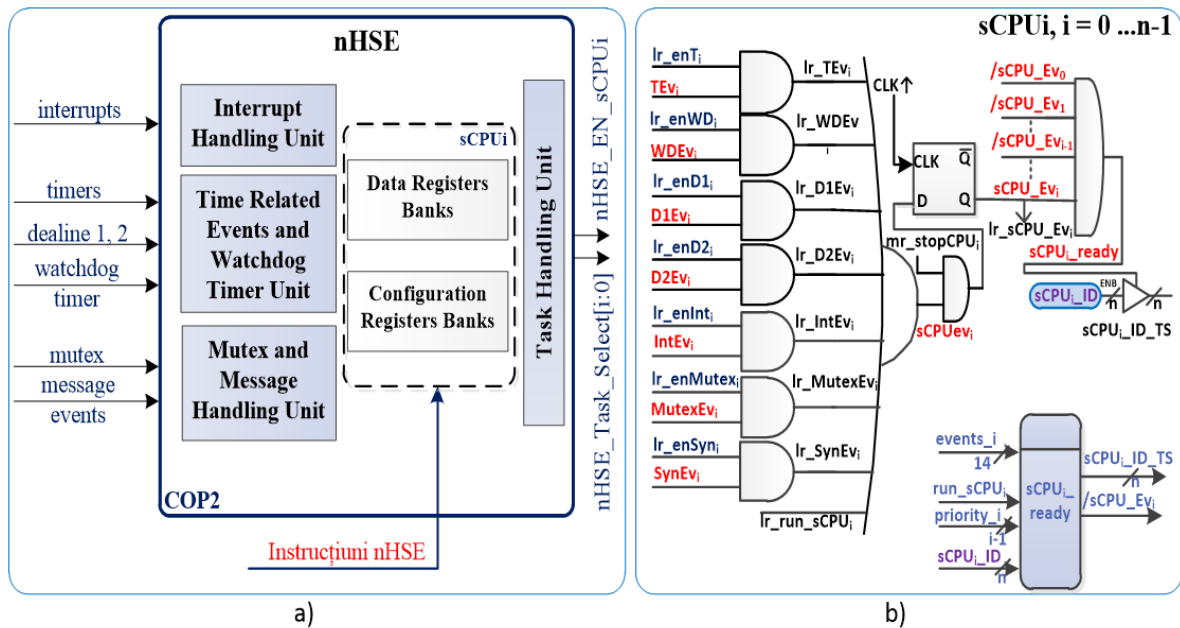


Figura 5-9 Tratarea evenimentelor la nivelul planificatorului nHSE

După cum se poate observa în Figura 5-9.b, evenimentele care reprezintă semnale de intrare pentru modulul nHSE sunt: evenimentul generat de timer (TEv_i), evenimentul generat de ceasul de gardă ($WDEv_i$), evenimentul generat de depășirea primei limite critice ($D1Ev_i$), evenimentul generat de depășirea celei de-a doua limite critice ($D2Ev_i$), întreruperile ($IntEv_i$) și evenimentele generate de mutex-uri ($MutexEv_i$) respectiv de mecanismul de comunicație inter-task ($SynEv_i$). Regiștrii de comandă, control și stare ai nMPRA cu efect direct sau indirect asupra nHSE sunt prezentați și descriși în specificațiile planificatorului de timp real

nHSE. Planificatorul din componența arhitecturii nMPRA monitorizează constant toate evenimentele care sunt adresate semiprocresoarelor. Evenimentele tratate de nHSE sunt reprezentate de următoarele semnale (Figura 5-9.b): întreruperi de timp (lr_TEvi), evenimente generate de ceasul de gardă (lr_WDEvi), două evenimente pentru termene limită (lr_D1Evi , lr_D2Evi), întreruperi ce pot fi atașate unui task i (lr_IntEVi), mutex-uri folosite pentru accesarea resurselor partajate ($lr_MutexEvi$) și evenimente pentru sincronizarea și comunicația între task-uri (lr_SynEVi). În cazul în care se activează simultan multiple evenimente asociate unui task i ce rulează pe un semiprocresor sCPU i , trebuie să existe o cale să selectăm ordinea în care aceste evenimente sunt tratate. Pentru aceasta, fiecare sCPU i are atașat un registru denumit *Event Priority Register* (crEPRI) ce conține nivelul de prioritate pentru fiecare tip de eveniment. Aceste priorități sunt diferite, variind de la 0 la 6 deoarece există 7 tipuri de evenimente.

Planificatorul nHSE folosește un spațiu unificat pentru întreruperi și task-uri, și o regulă de planificare prin care un task cu prioritate ridicată nu poate fi întrerupt de întreruperi asigurate task-urilor cu prioritate scăzută (Figura 5-10). Această regulă suportă nevoia de a asigura respectarea termenelor limită pentru task-urile care trebuie să furnizeze un răspuns de timp real stimulilor externi. nHSE permite activarea sau dezactivarea întreruperilor precum și atașarea acestora unor task-uri cu o prioritate mai scăzută.

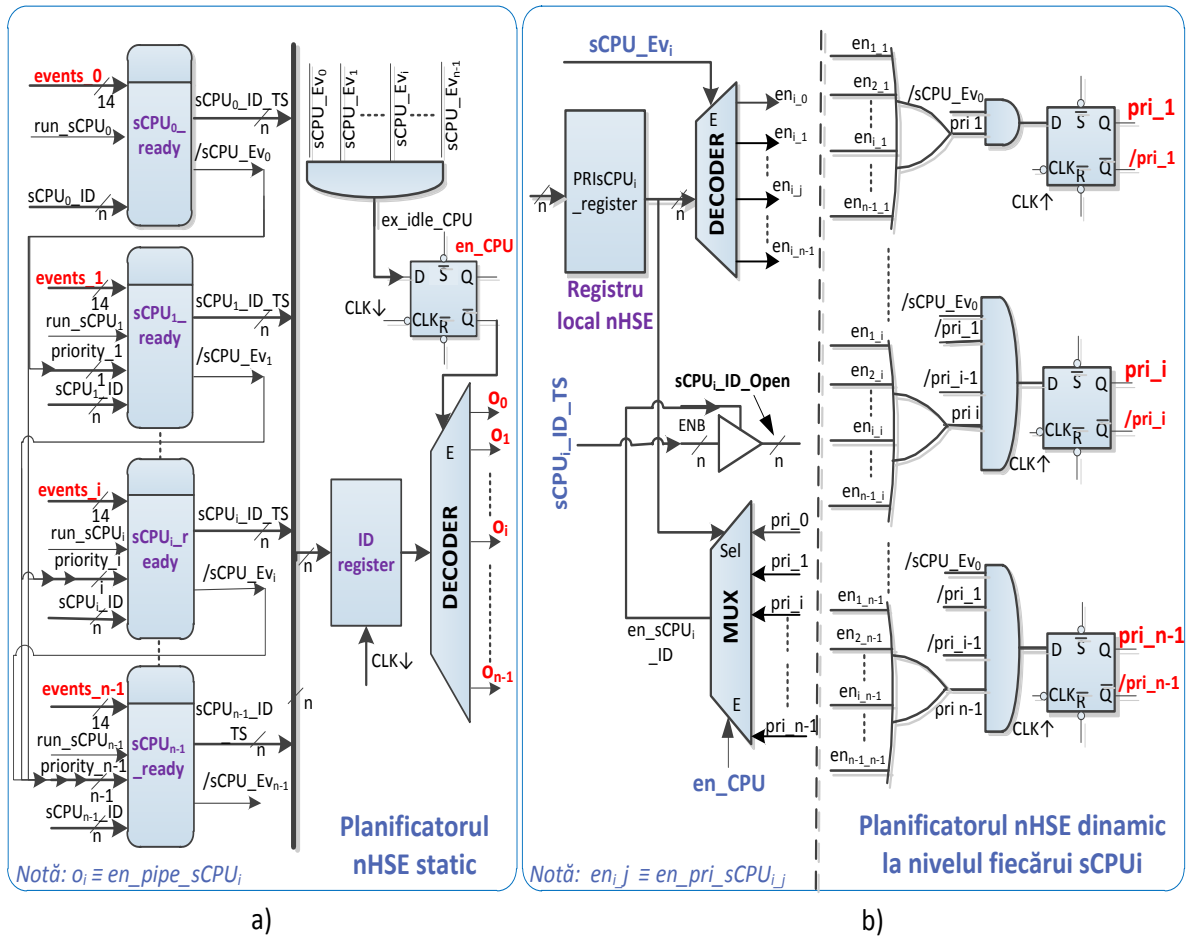


Figura 5-10 Asocierea întreruperilor la sCPUi și suportul hardware pentru planificarea dinamică a planificatorului nHSE [4]

În arhitectura nMPRA, fiecare task are asociat un timer care poate fi configurat să genereze o întrerupere atunci când timpul alocat acelui task se apropie de finalizare. În același timp, task-ul poate răspunde unui eveniment extern dacă acel eveniment este activat de instrucțiunea CTC2 cu mnemonica *wait*. Această instrucțiune este foarte importantă deoarece permite sincronizarea execuției atunci când sunt așteptate evenimente multiple. Sub controlul software-ului, în funcție de prioritatea fiecărui sCPUi, aceste evenimente sunt tratate iar biții relativi acestora sunt resetați. Schema de prioritizare implementată în această arhitectură de planificator selectează categoria de evenimente activă cu cea mai mare prioritate, în scopul de a fi tratată.

Execuția instrucțiunii CTC2 cu mnemonica *wait Rj* de către coprocesorul 2 are ca efect copierea conținutului registrului *Rj* din fișierul de regiștri al COP0 în registrul *crTRi*, pentru a valida evenimentele așteptate de task-ul *i* atașat semiprocessorului sCPUi. Registrul *Rj* poate fi oricare din regiștrii r0-r31 din componența fișierului de regiștri aparținând sCPUi-ului aflat în execuție. Dacă în momentul apelului cel puțin un eveniment este activ task-ul nu se suspendă. Acțiunea instrucțiunii CTC2 - *wait Rj* pentru un semiprocessor *i* este următoarea: *crTRi[sCPUi] <- Rj*, unde *Rj* reprezintă un registru de uz general, iar *crTRi* este un registru care are rolul de a valida sau a inhiba unul din cele șapte evenimente. După cum putem observa în Figura 5-11 la momentul de timp T1, instrucțiunea *0x48c10000* realizează scrierea registrului *crTRi[0][31:0]* cu valoarea *0x00000011*. Această valoare a fost memorată în prealabil în fișierul de regiștri la locația *RF_registers[1][31:0]* (sCPU0:r1) prin intermediul instrucțiunii *0x20010011* executată de sCPU0. După procesarea instrucțiunii *0x48c10000*, la momentul de timp T2 planificatorul nHSE realizează comutarea contextelor dintre sCPU0 și sCPU1.

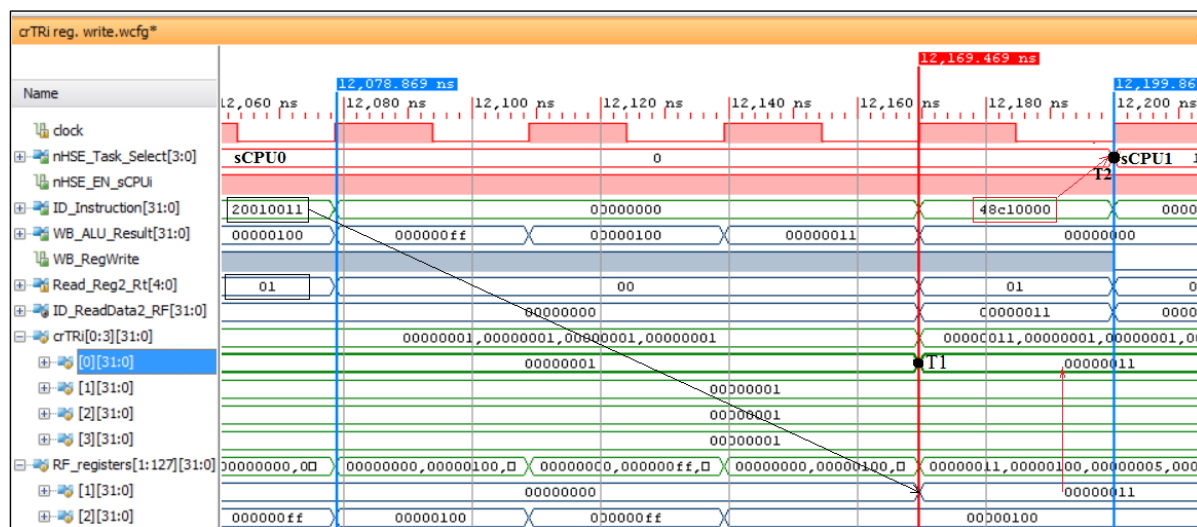


Figura 5-11 Modificarea registrului de control crTRi[0][31:0] și schimbarea contextelor dintre sCPU0 și sCPU1

Tabelul 5-4: Componența registrului crEVi

31..7	6	5	4	3	2	1	0
-	SynEvi	MutexEvi	IntEvi	D2Evi	D1Evi	WDEvi	TEvi

Registrul *crEVi* prezentat în Tabelul 5-4, este un registru care indică apariția unuia din evenimente *lr_TEvi*, *lr_WDEvi*, *lr_D1Evi*, *lr_D2Evi*, *lr_IntEvi*, *lr_MutexEvi* și *lr_SynEvi*. Prioritizarea globală a sistemului de evenimente implică existența în sistem a șapte tipuri de evenimente cu prioritate diversă, împreună cu circuitul de decodificare și selecție a

evenimentului cel mai prioritar. După cum se poate observa în Tabelul 5-5, *crEPRi* reprezintă registrul priorităților evenimentelor majore, alocând câte 3 biți pentru fiecare eveniment. Grupurile de biți *Pri_TEv*, *Pri_WDEv*, *Pri_D1Ev*, *Pri_D2Ev*, *Pri_IntEv*, *Pri_MutexEv* și *Pri_SynEv* reprezintă prioritățile atașate categoriilor de evenimente validate sau inhibitate în registrul de control *crEVi*.

Tabelul 5-5: Componenta registrului *crEPRi*

31..21	20..18	17..15	14..12	11..9	8..6	5..3	2..0
-	Pri_SynEv	Pri_MutexEv	Pri_IntEv	Pri_D2Ev	Pri_D1Ev	Pri_WDEv	Pri_TEv

În exemplul ilustrat în Figura 5-12 pot fi 4 întreruperi (*ExtIntEv[0]* fiind cea mai prioritară întrerupere iar *ExtIntEv[3]* cea mai puțin prioritară), fiecare dintre acestea putând fi atașate unuia din cele 4 semiprocsoare prin intermediul registrului *grINT_Idi[0:3][31:0]*. Prioritatea evenimentului de tip întrerupere are valoarea *crEPRi[0][14:12] = 3'b000*, fiind cel mai prioritar eveniment. La nivelul sCPU0, evenimentul de timp este și el activat prin intermediul bitului corespunzător *crTRi[0][0]*, dar deoarece prioritatea acestuia este setata la valoarea *crEPRi[0][2:0] = 3'b001*, acesta va fi planificat după tratarea întreruperii deoarece *crEPRi[0][14:12] < crEPRi[0][2:0]*.

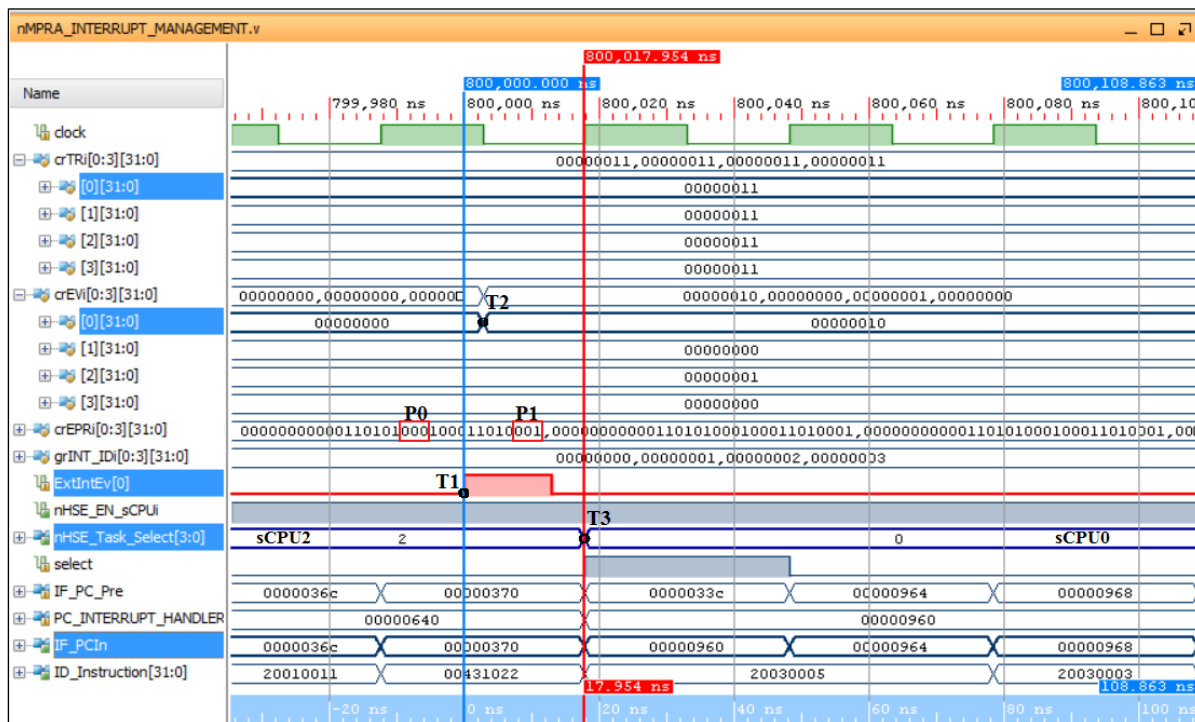


Figura 5-12 Răspunsul planificatorului nHSE la tratarea unei întreruperi externe asincrone

5.2.3. Validarea instrucțiunilor dedicate planificatorului nHSE

Pentru implementarea noilor instrucțiuni prezentate în Tabelul 5-6 s-a utilizat setul de instrucțiuni MIPS32. Această implementare de procesor reprezintă un exemplu de corespondență de tipul instrucțiuni generice nHSE - instrucțiuni coprocesor 2 MIPS. Astfel, alte implementări de procesor pot realiza alte corespondențe în funcție de aplicație. MIPS pune la dispoziția utilizatorului un sistem de coprocsoare pentru extensia de funcționalități ale procesorului de bază [59]. La dispoziția utilizatorului poate fi coprocesorul 2. Instrucțiunea CTC2 (*Copy control word to COP2*) având mnemonica *mover (wait Rj [4])*, face parte din grupul de instrucțiuni dedicate funcționării planificatorului hardware integrat.

Această instrucțiune determină copierea conținutului registrului R_j în registrul $crTRi$, validând astfel evenimentele așteptate de task-ul i . Dacă în momentul apelului instrucțiunii cel puțin un eveniment este activ, task-ul nu se suspendă.

Tabelul 5-6: Instrucțiunile dedicate planificatorului nHSE; *CFC2* - Copy control word from COP2; *CTC2* - Copy control word to COP2; *MFC2* - Move word from COP2; *MTC2* - Move word to COP2; *LWC2* - Load Word to COP2; *SWC2* - Store Word from COP2

Instrucțiune COP2	Ce execută instrucțiunea	OpCode	Rs	Operația	Registrii implicați
CFC2	Transferă un cuvânt de control de la COP2	010010	CF 00010	$GPRi[Rt] \leftarrow COP2[Immediate]$	$crTRi$, $crEVi$, $crEP Ri$, $cr0D1$, $cr0D2$, $cr0MSTOP$, $cr0RESET$, $cr0CPUID$, $crEM Ri j$, $crEER i j$
CTC2	Transferă un cuvânt de control la COP2	010010	CT 00110	$COP2[Immediate] \leftarrow GPRi[Rt]$	$crTRi$, $crEVi$, $crEP Ri$, $cr0D1$, $cr0D2$, $cr0MSTOP$, $cr0RESET$, $crEM Ri j$, $crEER i j$
MFC2	Transferă un cuvânt de la COP2	010010	MF 00000	$GPRi[Rt] \leftarrow COP2[Immediate]$	$mrPRI sCPU i$, $mrTEVi$, $mrWDEV i$, $mrD1EV i$, $mrD2EV i$, $mrCntRun i$, $mrCntSleep i$, $mr0CntSleep$, $mrCommReg i j$
MTC2	Transferă un cuvânt la COP2	010010	MT 00001	$COP2[Immediate] \leftarrow GPRi[Rt]$	$mrTEVi$, $mrWDEV i$, $mrD1EV i$, $mrD2EV i$, $mrCntRun i$, $mrCntSleep i$, $mr0CntSleep$, $mrCommReg i j$
LWC2	Încarcă din memorie un cuvânt în COP2	110010	rs	$COP2[Rt,0] \leftarrow MEM[GPRi[Rs]+Immediate]$	$grINT_Idi$, $grMutex i$, $grERFi$, $grEVG$, $grINT_PR$
SWC2	Salvează în memorie un cuvânt de la COP2	110011	rs	$MEM[GPRi[Rs]+Immediate] \leftarrow COP2[Rt,0]$	$grINT_Idi$, $grMutex i$, $grERFi$, $grEVG$, $grINT_PR$

Figura 5-13 ilustrează efectul instrucțiunilor *LWC2* și *SWC2*, acestea făcând parte din segmentul instrucțiunilor de lucru cu memoria de date. Astfel, liniile întrerupte indică calea de date corespunzătoare instrucțiunii LWC2 (Load Word to COP2) ce încarcă un cuvânt din memoria de date în nHSE (COP2) și SWC2 (Store Word from COP2) ce salvează în memoria de date un cuvânt provenit de la nHSE (COP2).

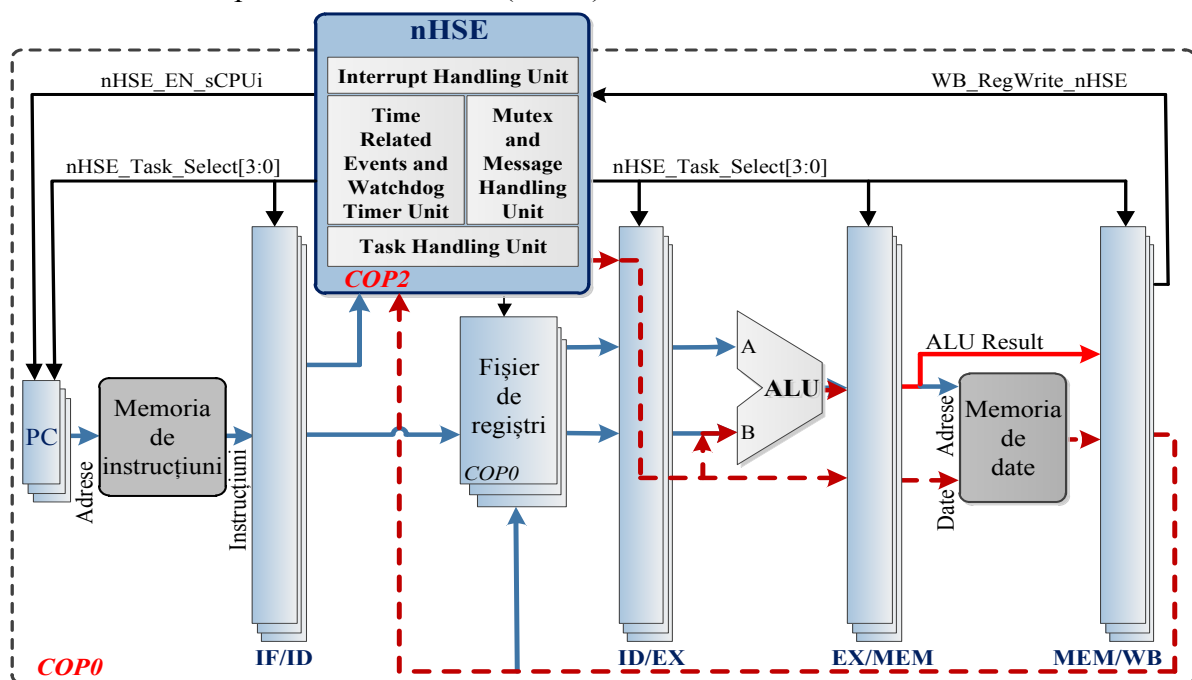


Figura 5-13 Efectul implementării instrucțiunilor LWC2 și SWC2

Prin intermediul formelor de undă din Figura 5-14, se poate măsura jitter-ul procesorului nMPRA la apariția unui eveniment extern asincron cât și timpul necesar pentru schimbarea contextelor. Aceste forme de undă reprezintă semnalele proiectului SoC cât și a procesorului nMPRA. Astfel se testează timpul de răspuns la un eveniment asincron generat de acționarea selectorului DIP1 de pe kit-ul de dezvoltare Virtex-7. Formele de undă ilustrează semnalele interne ale procesorului și semnalul *Switch[0]* ce reprezintă intrarea asignată întreruperii *ExtIntEv[0]*. Se poate observa modificarea semnalului corespunzător pinului de intrare al circuitului FPGA, marcat de momentul de timp T1. Pentru a trata această întrerupere externă atașată la sCPU0, este nevoie de un ciclu de ceas, momentul de timp T2 indicând reacția planificatorului nHSE prin modificarea semnalului intern *nHSE_Task_Select* pentru a introduce în execuție sCPU0. Momentul T3 indică modificarea semnalului *LED[1]*, ieșirea corespunzătoare acestui led fiind mapată în spațiul de adrese al memoriei de date. Astfel, pentru accesarea acestui led procesorul execută o instrucțiune MIPS de tipul *sw* (*0xadcd0000*). Semnalul *M_ALU_Resul[29]* indică o operație cu intrările/ieșirile mapate în spațiul de adrese al memoriei de date, iar semnalele *M_ALU_Resul[28:26] = 3'b100* indică ledurile ca fiind ieșirile destinație pentru data *M_ReadData2*. De la momentul apariției evenimentului extern până la aprinderea LED-ului corespunzător de pe kit-ul de dezvoltare Virtex-7 sunt necesari cinci cicluri de ceas, în funcție de momentul în care este acționat selectorul *DIP[0]*.

Cu o execuție independentă, planificatorul are intrări pentru evenimente cum ar fi întreruperi și timer-e. După cum se poate observa în Figura 5-14, selectorul *nHSE_EN_sCPUi* reprezintă comanda de activare pentru task-urile la care sunt atașate evenimentele. Atunci când planificatorul activează semiprocesoarele sCPUi prin intermediul semnalelor *cr0MSTOP*, diagrama bloc decodifică în același timp regiștrii *mrPRIsCPUi*, *crTRi*, *crEVi* și *grINT_IDi* pentru task-urile active. La un moment dat, în cazul unei execuții normale, numai un sCPUi poate fi în starea RUN. Acest lucru este posibil prin intermediul planificatorului nHSE, stărilor task-urilor, blocului dedicat evenimentelor, cât și semnalelor de sincronizare.

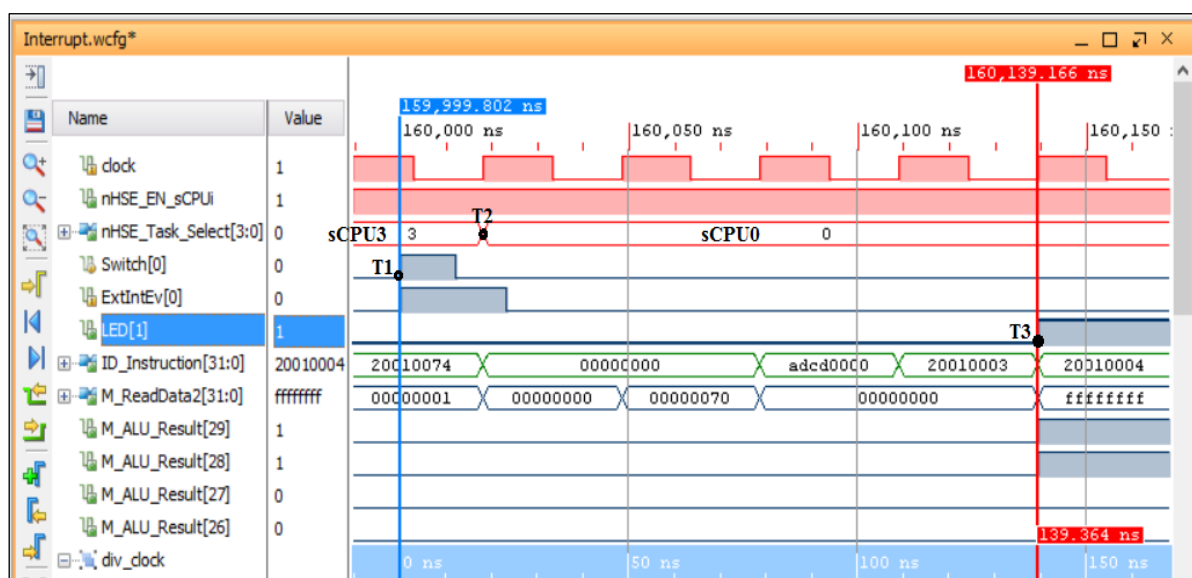


Figura 5-14 Semnalele corespunzătoare tratării unui eveniment extern asincron

5.2.4. Mecanismul de sincronizare prin mutex-uri

Mecanismele de sincronizare și comunicație inter-task reprezintă alte două aspecte importante în implementarea arhitecturii nMPRA. Proiectarea acestor mecanisme în hardware îmbunătățește în mod convenabil coeficientul WCET, furnizând astfel o soluție optimă pentru excluderea mutuală în cazul resurselor partajate și comunicația între task-uri. Acest aspect reprezintă un subiect interesant în partajarea resurselor de către task-urile procesorului nMPRA. Din punctul de vedere al arhitecturii, implementarea unor astfel de mecanisme trebuie să genereze regiuni critice extrem de scurte, corespunzătoare operațiilor efectuate pe resursele partajate. Cu alte cuvinte, implementarea operațiilor pe mutex-uri se bazează pe instrucțiuni atomice, obținând astfel performanțe deosebite și timpi de blocare previzibili.

Suportul hardware pentru implementarea mecanismului de sincronizare este reprezentat de fișierul de regiștri pentru mutex-uri. Acesta conține un număr de regiștri speciali denumiți *grMutexi*, ce conțin starea mutex-urilor și identificatorii task-urilor proprietar. Numărul de mutex-uri respectiv de regiștri *grMutexi* depinde de implementare, iar pentru exemplificare s-a notat acest număr cu *m*. Pe baza acestui număr se constituie fișierul de regiștri pentru mutex-uri. Registrul *grMutexi* selectează ID-ul task-ului la care a fost atașat mutex-ul *i*, după reset toți biții sunt pe 0 logic (Tabel 5-7).

În cele ce urmează se va ilustra situația în care planificatorul așteaptă dezactivarea semnalului *nHSE_inhibit_CC*, întârziind reacția planificatorului nHSE. Astfel, se va putea analiza conținutul regiștrilor coprocesorului 2 atunci când se realizează o schimbare de context concomitent cu apariția unui eveniment de timp periodic.

Tabelul 5-7: Implementarea fișierului de regiștri pentru mutex-uri

grMutex_i	31	30..5	4	3	2	1	0
	Mutex _i		TaskID bit4	TaskID bit3	TaskID bit2	TaskID bit1	TaskIDbit0
grMutex₀	0/1(Mutex ₀)						
grMutex₁	0/1(Mutex ₁)						
...	...	...	...	...	...	...	...
grMutex_{m-1}	0/1(Mutex _{m-1})		TaskID bit4	TaskID bit3	TaskID bit2	TaskID bit1	TaskIDbit0

Tabelul 5-8 prezintă o parte dintr-o aplicație scrisă în cod mașină pentru testarea și validarea mutex-urilor. Pentru acest context s-a ales modificarea registrului *grMutexi[0]* prin intermediul instrucțiunii *0xC8220000*. Efectul execuției instrucțiunilor din Tabelul 5-8 este ilustrat în Figura 5-15. După cum se poate observa, prin execuția instrucțiunii *0xC8220000* se dorește validarea mecanismului de sincronizare folosind mutex-uri. La momentul de timp T1 această instrucțiune este extrasă din memorie, iar la momentul de timp T3 și T5 se poate vedea modificarea registrului global *grMutexi[0]* de către sCPU3 și sCPU0. Pe toată perioada execuției instrucțiunilor de lucru cu mutex-ul *grMutexi[0]* este activat semnalul intern *nHSE_inhibit_CC* pentru a nu permite altor task-uri să întrerupă această instrucțiune. După cum se poate observa în Figura 5-15, cursoarele C1, C2, și C3 indică schimbările de context realizate sub comanda selectorului *nHSE_Task_Select[3:0]* și a liniei de activare *nHSE_EN_sCPUi*. Semnalul *ID_Instruction* furnizează instrucțiunea activă din etajul ID ce corespunde semiprocessorului selectat pentru execuție.

Prioritățile sCPUi-urilor, memorate în regiștrii *mrPRIIsCPUi[3:0]* implementați la nivelul planificatorului nHSE, indică faptul că planificatorul procesorului nMPRA este unul dinamic,

prioritățile celor 4 semiprocsoare (sCPU0, sCPU1, sCPU2, sCPU3) fiind 0, 8, 7 și 6. Regiștrii de control *crTRi[3:0]*, *crEVi[3:0]* și *crEP Ri[3:0]* validează, memorează și prioritizează evenimentele și întreruperile așteptate de fiecare sCPUi.

Tabelul 5-8: Codul unei aplicații pentru validarea mutex-urilor

Descriere	Cod aplicație
Secțiunea de cod testează funcționarea mutex-urilor implementate în hardware la nivelul planificatorului nHSE. Semiprosorul sCPU3 ocupă mutex-ul grMutexi[0]	<pre> //--- sCPU3--- AC2F0000, //1010 11_00 001_0 1111_ 0000 0000 0000 0000 //sw, M[R/rs]+SignExtImm=R[rt], salvează r15 COP0 în mem. C8220000, //1100 10_00 001_0 0010_ 0000 0000 0000 0000 //ldgr, din memoria de date încarcă în grMutexi[0] COP2 20010070, //addi, R[rt=1]=R[rs=0]+SignExtImm (=0070) 20010071, //addi, R[rt=1]=R[rs=0]+SignExtImm (=0071) 20010072, //addi, R[rt=1]=R[rs=0]+SignExtImm (=0072) CC020000, //1100 11_00 000_0 0010_ 0000 0000 0000 0000 //stgr, din grMutex[0] COP2 salvează în memoria de date 8C020000, //1000 11_00 000_0 0010_ 0000 0000 0000 0000 //lw, R[rt=2]=M[R/rs]+SignExtImm, încarcă din mem. în COP0 20010073, //addi, R[rt=1]=R[rs=0]+SignExtImm (=0073) 20010074, //addi, R[rt=1]=R[rs=0]+SignExtImm (=0074) 20010000, //addi, R[rt=1]=R[rs=0]+SignExtImm (=0000) AC210000, //1010 11_00 001_0 0001_ 0000 0000 0000 0000 //sw, M[R/rs]+SignExtImm=R[rt], salvează r1 COP0 în mem. C8220000, //1100 10_00 001_0 0010_ 0000 0000 0000 0000 //ldgr, din memoria de date încarcă în grMutexi[0] COP2 //---Schimbare context sCPU3->sCPU0--- </pre>
sCPU0 trebuie să trateze un eveniment de timp și deoarece este mai prioritar întrerupe sCPU3, încercând să acceseze mutex-ul grMutexi[0] ocupat de sCPU3	<pre> //--- sCPU0--- 20010000, //addi, R[rt=1]=R[rs=0]+SignExtImm (=0000) 00000000, //nop AC2F0000, //1010 11_00 001_0 1111_ 0000 0000 0000 0000 //sw, M[R/rs]+SignExtImm=R[rt], salvează r15 COP0 în mem. C8220000, //1100 10_00 001_0 0010_ 0000 0000 0000 0000 //ldgr, din memoria de date încarcă în grMutexi[0] COP2 //---Schimbare context sCPU0->sCPU3--- </pre>
sCPU0 se suspendă în așteptarea mutex-ului grMutexi[0] iar sCPU3 își reia execuția eliberând mutex-ul	<pre> //--- sCPU3--- C8220000, //1100 10_00 001_0 0010_ 0000 0000 0000 0000 //ldgr, din memoria de date încarcă în grMutexi[0] COP2 00000000, //nop 48C1ffff, //010010_00110_00010_1111111111111111 //instrucțiunea movcr are ca efect schimbarea de context //---Schimbare context sCPU3->sCPU0--- </pre>
sCPU0 intră în execuție pentru a trata un eveniment de tip mutex, banda de asamblare nefiind afectată de schimbarea contextelor efectuată anterior	<pre> //--- sCPU0--- 20010070, //addi, R[rt=1]=R[rs=0]+SignExtImm (=0070) 20010071, //addi, R[rt=1]=R[rs=0]+SignExtImm (=0071) CC030000, //1100 11_00 000_0 0011_ 0000 0000 0000 0000 //stgr, din grMutex[0] COP2 salvează în memoria de date ... </pre>

Pentru a memora ce eveniment este tratat la nivelul fiecărui sCPUi, s-a folosit regiștrii globale *gr_EV_select_sCPU[3:0]*, aceștia luând valori de la 0 la 6 în funcție de evenimentul tratat, valoarea 7 indicând faptul că nu este procesat nici un eveniment.

În această exemplificare, regiștrii *crEM Ri0* = 0x0000000F și *crEERi0* = 0x0000000F indică faptul că sunt validate patru mutex-uri și patru evenimente de comunicație prin mesaje. Trebuie subliniat faptul că acești regiștri au rolul de a valida mecanismele de sincronizare și comunicație și nu pot produce evenimente de tip *MutexEvi* sau *SynEvi*.

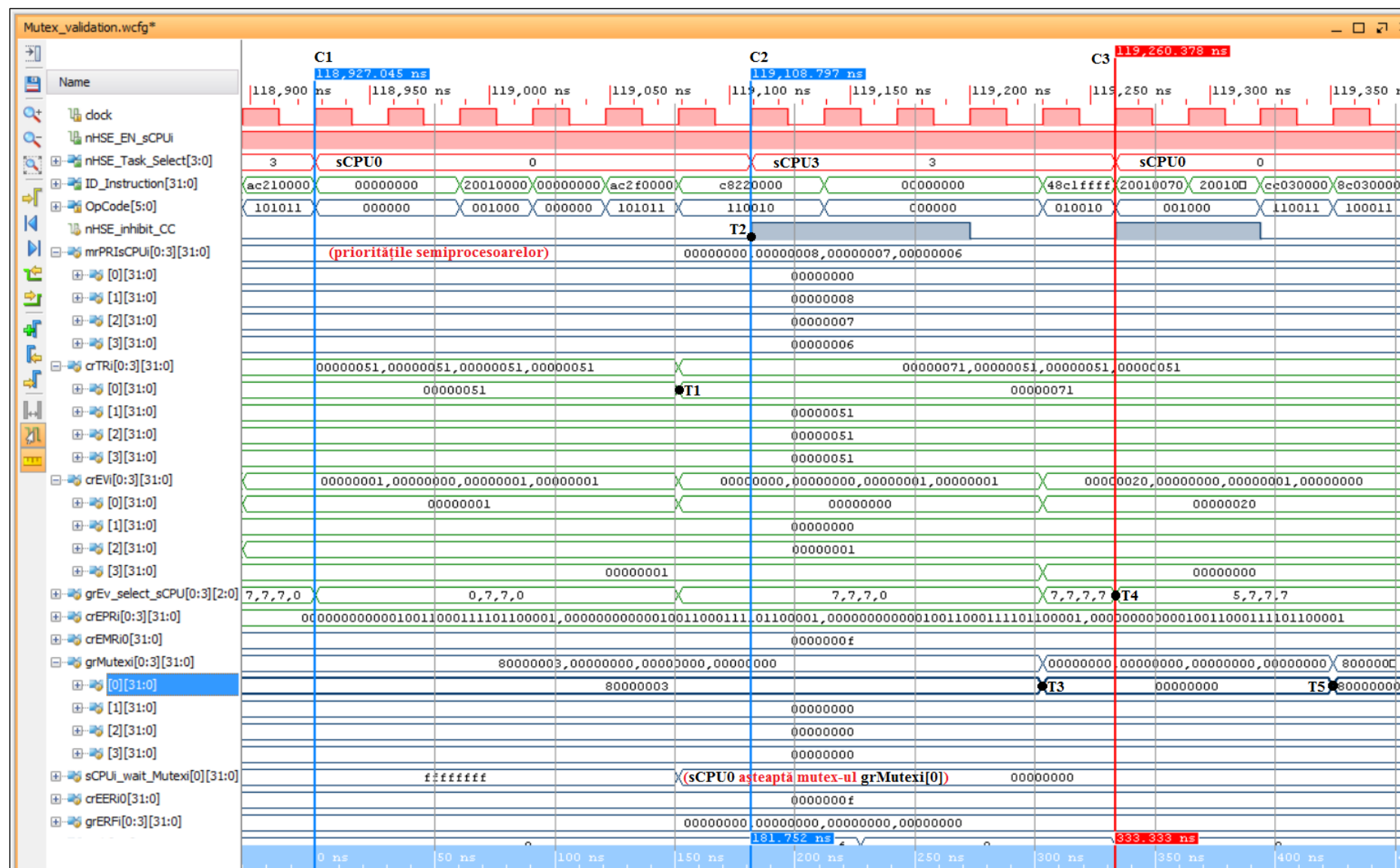


Figura 5-15 Semnalele corespunzătoare validării mutex-urilor

5.2.5. Mecanismul de comunicație inter-task

Pentru implementarea mecanismului de comunicație inter-task arhitectura nMPRA folosește un număr de e regiștri globali $grERFi$, fiecare având $2n+k+1$ biți. Acești regiștri compun fișierul de regiștri pentru evenimente, fiecare registru folosind un bit pentru a memora starea evenimentului, $2*n$ biți pentru a memora ID-ul task-urilor sursă și destinație, și k biți pentru a memora mesajul. După cum se poate vedea în Tabelul 5-9, $grERFi$ este registrul care definește un eveniment și face parte din fișierul de regiștri pentru evenimente, după reset toți biții fiind pe 0 logic.

Tabelul 5-9: Starea evenimentului, ID-ul task-ului sursă și destinație și mesajul corespunzător memorat în fișierul de regiștri pentru evenimente ($grERFi$)

Adresa	Registru	$2n+k+1$	$nj-1$	...	0	$nj-1$	...	0	$k-1$	...	1	0
		Ev_i	$sIDnj-1$	...	$sID0$	$dIDnj-1$	...	$dID0$	$Messk-1$	...	$Mess1$	$Mess0$
$Adresa_0$	$grERF_0$	0/1										
$Adresa_1$	$grERF_1$	0/1										
...	...	...	...	...	...	...	...	...	...	...	...	...
$Adresa_{e-1}$	$grERF_{e-1}$	0/1										

Tabelul 5-10: Implementarea registrului de control $crEERij$

31	30..5	4	3	2	1	0
E31	E_j ($j = 30..5$)	E4	E3	E2	E1	E0

În Tabelul 5-10 este reprezentat registrul $crEERij$ corespunzător validării evenimentelor de sincronizare, valoarea 0 semnifică un eveniment de comunicație prin mesaje inhibat iar 1 un eveniment validat. Instrucțiunile de lucru cu regiștrii $grERFi$ și blocul hardware dedicat implementării acestui mecanism este proiectat la nivelul planificatorului nHSE, generând toate semnalele corespunzătoare modificării regiștrilor din fișierul de regiștri pentru evenimente. Astfel, atunci când unul sau mai multe task-uri așteaptă un mutex sau eveniment, planificatorul nHSE verifică regiștrii $grERFi$ într-un ciclu de ceas, cu ajutorul circuitului de căutare implementat în hardware. Tabelul 5-11 prezintă o secțiune dintr-o aplicație scrisă în cod mașină pentru testarea și validarea evenimentelor de comunicație prin mesaje.

Tabelul 5-11: Codul unei aplicații pentru validarea evenimentelor de sincronizare $grERFi$

Descriere	Cod aplicație
Secțiunea de cod testează funcționarea mecanismului de comunicație implementate în hardware la nivelul planificatorului nHSE. Din motive de reprezentare, doar o parte dintre aceste instrucțiuni au fost capturate în formele de undă redate în Figura 5-16.	<pre> //-----sCPU1----- AC100000, //1010 11_00 000_1 0000_ 0000 0000 0000 0000 //sw, M[R/rs]+SignExtImm]=R[rt], salvează r16 COP0 în mem. C8030000, //1100 10_00 000_0 0011_ 0000 0000 0000 0000 //ldgr, din memoria de date încarcă în grERFi[0] COP2 20020070, //addi, R[rt=2]=R[rs=0]+SignExtImm (=0070) 20020071, //addi, R[rt=2]=R[rs=0]+SignExtImm (=0071) 20020072, //addi, R[rt=2]=R[rs=0]+SignExtImm (=0072) //---Schimbare context sCPU1->sCPU0--- CC030000, //1100 11_00 000_0 0011_ 0000 0000 0000 0000 //stgr, din grERFi[0] COP2 salvează în memoria de date 8C030000, //1000 11_00 000_0 0011_ 0000 0000 0000 0000 //lw, R[rt=3]=M[R/rs]+SignExtImm], încarcă din mem. în COP0 20010070, //addi, R[rt=1]=R[rs=0]+SignExtImm (=0070) AC000000, //1010 11_00 000_0 0000_ 0000 0000 0000 0000 //sw, M[R/rs]+SignExtImm]=R[rt], salvează r0 COP0 în mem. C8030000, //1100 10_00 000_0 0011_ 0000 0000 0000 0000 //ldgr, copie din memoria de date în registrul grERFi[0] COP2 48C2FFFF, //010010_00110_00010_1111111111111111 //instrucțiunea movcr are ca efect schimbarea de context //---Schimbare context sCPU0->sCPU1--- </pre>

Figura 5-16 ilustrează semnalele corespunzătoare mecanismului de comunicație inter-task implementat la nivelul nHSE. Momentul T2 indică efectul execuției instrucțiunii `0xC8030000`, aceasta fiind extrasă din memorie la momentul de timp T1. Dacă un task primește un mesaj, blocul hardware implementat în nHSE trebuie să identifice ID-ul corespunzător sursei mesajului, setând bitul `SynEvi` din componența registrului `crEVi`. Rezultatul execuției instrucțiunii mai sus amintite, este copierea cuvântului `0x8400FFFF` din memoria de date în registrul global `grERFi[0]`. Cuvântul `0x8400FFFF` (`32'b1_00001_00000_00000111111111 111111`) reprezintă operația de trimitere a mesajului `_000001111111111111111111` de la sCPU1 (ID_sursa = 00001) către sCPU0 (ID_destinație = 00000). Acest lucru are ca efect schimbarea contextului dintre sCPU1 și sCPU0, moment marcat de T3, deoarece mesajul a fost trimis de la semiprocessorul sCPU1 la sCPU0, acesta din urmă fiind mai prioritar. Registrul de control `crEVi[0] = 0x00000040` indică planificatorului faptul că a apărut un eveniment de tip mesaj, acesta fiind validat prin intermediul bitului `lr_enSyni` din registrul `crTRi[0] = 0x00000071`. Valoarea 6 memorată în registrul global `grEv_Select_sCPU[0]` indică faptul că sCPU0 tratează un eveniment de tip eveniment de sincronizare prin mesaje. Execuția acestei instrucțiuni este indivizibilă, astfel încât dacă pe toată această perioadă apare un eveniment mai prioritar, tratarea acestuia poate fi întârziată până la trei cicli de ceas, timp necesar finalizării instrucțiunilor atomice dedicate mecanismelor de sincronizare și comunicație inter-task. Cu toate acestea, implementarea planificatorului nHSE și a evenimentelor de sincronizare și comunicație în hardware garantează respectarea termenelor limită și predictibilitatea execuției setului de task-uri ce a trecut testul de fezabilitate.

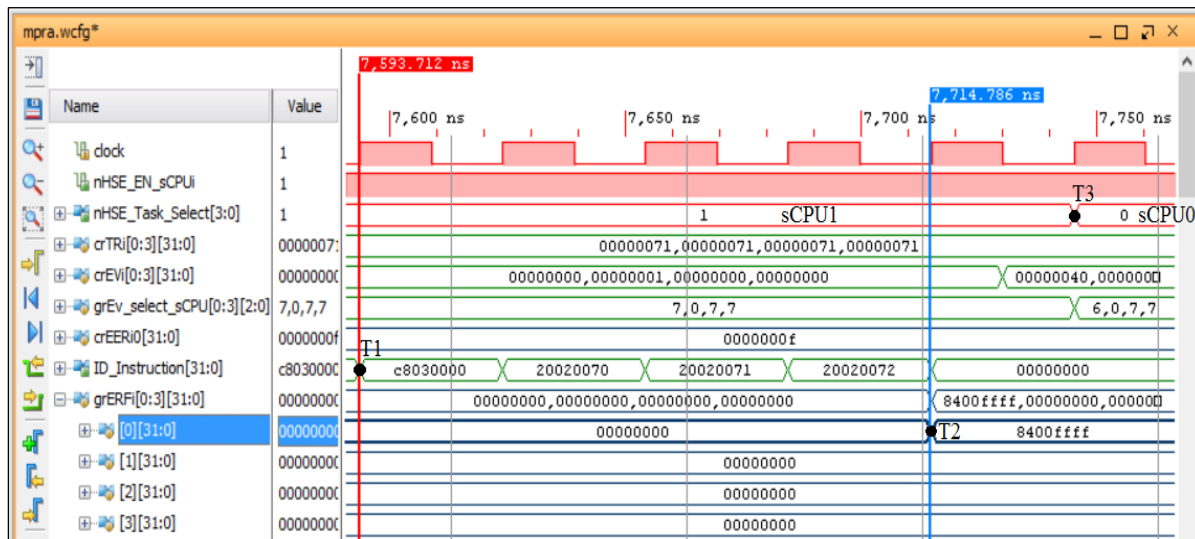


Figura 5-16 Semnalele corespunzătoare evenimentelor de sincronizare prin mesaje

6. Costuri de implementare ale arhitecturii nMPRA și criterii de performanță

Deși nMPRA este o arhitectură cu multiplexare de resurse, costurile de implementare ale acesteia sunt avantajoase față de alte arhitecturi comerciale similare. Trebuie specificat faptul că o astfel de implementare ar avea avantaje semnificative față de implementările comerciale existente pentru un număr de 16 sau chiar 32 de task-uri. Implementarea acestei arhitecturi pentru un număr mare de task-uri ar atrage după sine un necesar de resurse prea mare în raport cu sistemul pentru care este folosit procesorul, timp de propagare ale semnalelor nejustificați de mari, micșorând semnificativ frecvența de lucru a procesorului. În Figura 6-1 este prezentată distribuția celulelor logice utilizate pentru implementarea procesorului nMPRA cu 4 semiprocresoare, planificatorul dinamic nHSE, 4 întreruperi externe, inclusiv mecanismele de sincronizare și comunicație implementate în hardware. Reamintim că procesorul nMPRA se bazează pe o linie de asamblare cu cinci etaje, multiplicarea regiștrilor pipeline, numărătorului program cât și a fișierului de regiștri pentru fiecare sCPUi.

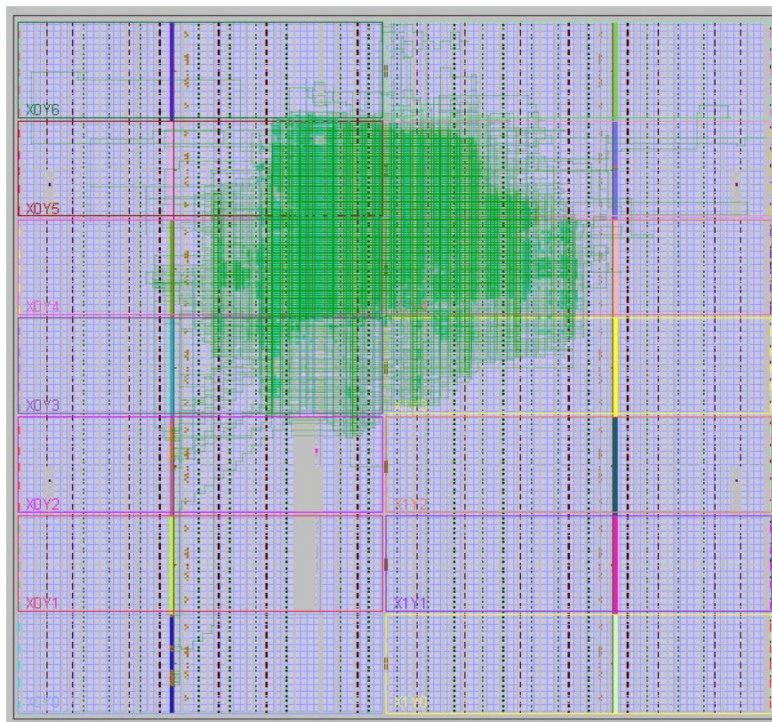


Figura 6-1 Distribuția componentelor logice pe pastila de siliciu

6.1. Analiza puterii consumate

Figura 6-2 prezintă în detaliu puterea consumată de implementarea nMPRA cu 16 semiprocresoare, având la bază suportul hardware pentru tratarea întreruperilor, atașarea dinamică a acestora oricărui sCPUi și implementarea mecanismelor de sincronizare și comunicație inter-task. Valorile prezentate în Figura 6-2 au fost obținute în urma implementării practice a procesorului nMPRA utilizând mediul de dezvoltare Vivado și kit-ul de dezvoltare Virtex-7, valorile putând varia în funcție de versiunea de nHSE și platforma utilizată pentru implementare. Cu toate acestea, totalul puterii consumate este mai mult decât acceptabil, în condițiile în care alte implementări similare depășesc aceste valori empirice.

Făcând o comparație între planificatorul dinamic și planificatorul static prezentat în [32], [60-62], putem afirma că implementarea de față implică un consum suplimentar datorită suportului pentru planificarea dinamică necesară modulului nHSE. Figura 6-2 ilustrează un raport cu privire la puterea consumată de circuitul FPGA în urma implementării proiectului SoC ce include procesorul nMPRA cu 16 sCPUi.

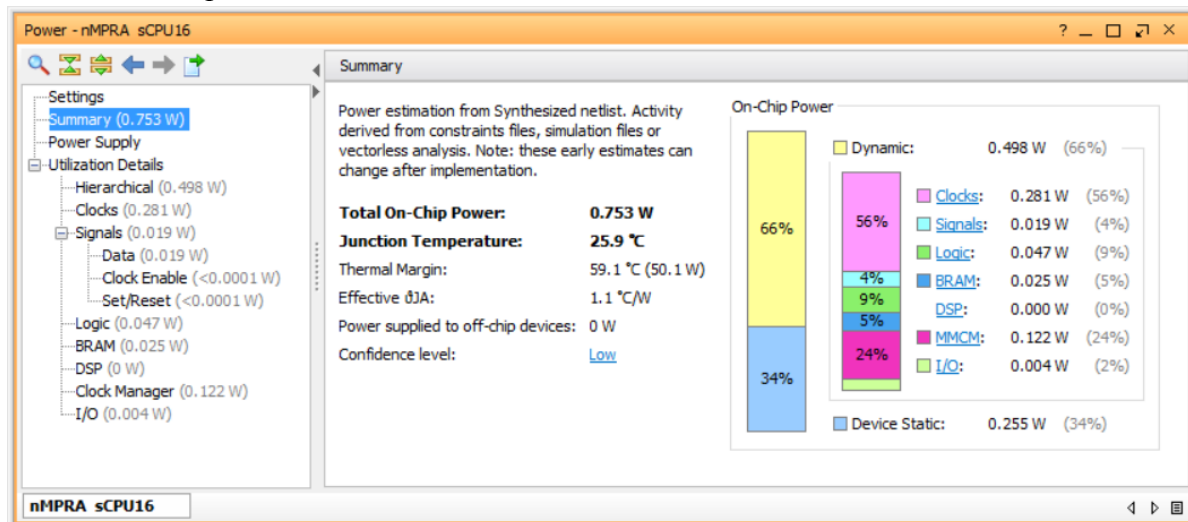


Figura 6-2 Estimarea puterii consumate după implementarea proiectului SoC. *MMCM*-Mixed Mode Clock Manager; *BRAM*-Block Ram; *Device Static*-reprezintă puterea consumată de FPGA la pornire când dispozitivul este configurat cu logica de lucru (Quiescent power)

6.2. Necesarul de resurse pentru implementarea proiectului SoC ce include procesorul nMPRA

Modulele care conțin doar elemente combinaționale, cum ar fi modulul *HazardDetection*, nu au fost multiplicare deoarece acestea furnizează la ieșire semnalele corespunzătoare datelor de intrare, acestea fiind deja memorate în regiștrii pipeline. Așadar, multiplicarea resurselor pentru sCPU16 însumează 4.275 KB de memorie. La această valoare se mai adaugă memoria RAM necesară pentru salvarea contextelor funcțiilor imbricate. Tabelul 6-1 prezintă necesarul de memorie pentru trei posibile implementări cu 4, 8 și 16 sCPUi și suportul pentru tratarea a patru întreruperi externe. Luând în considerare aceste date obținute în urma implementării procesorului nMPRA se poate face o comparație cu alte implementări similare. Tabelul 6-1 prezintă necesarul de blocuri logice pentru implementarea arhitecturii nMPRA cu sCPU4, sCPU8 și sCPU16, tot aici fiind incluse și resursele necesare implementării planificatorului nHSE. Așadar, în proiectarea modulului nHSE se ține cont în primul rând de numărul de semiprocsoare, evenimente de sincronizare și comunicație inter-task și numărul de întreruperi externe [63].

Tabelul 6-1: Necesarul de memorie pentru multiplicarea resurselor arhitecturii nMPRA

Numărul de sCPUi/ Mutex/Evenimente de tip mesaj - Necesar memorie	Memoria necesară pentru nHSE, inclusiv mecanismele de sincronizare și comunicație inter- task	Memoria necesară pentru PC, fișierul de regiștri și regiștrii pipeline	Necesarul de memorie (total)
nMPRA4/ grMutexi4/ grERFi4	0.259kB	0.86kB	1.119kB
nMPRA8/ grMutexi8/ grERFi8	0.451kB	1.72kB	2.171kB
nMPRA16/ grMutexi16/ grERFi16	0.835kB	3.44kB	4.275kB

În tabelele 6-2 și 6-3 sunt prezentate resursele necesare pentru implementarea procesorului nMPRA și a proiectului SoC. Procesorul nMPRA se bazează pe multiplicarea resurselor, astfel încât notațiile *nMPRA4*, *nMPRA8* și *nMPRA16* corespund unor implementări cu 4, 8 și 16 sCPUi. Notațiile *grMutexi4* și *grERFi4* corespund unor variante de procesor cu 4 mutex-uri și 4 regiștri de comunicație prin mesaje.

Tabelul 6-2: Resursele utilizate de procesorul nMPRA utilizând kit-ul cu FPGA Virtex-7

Numărul de sCPUi / Mutex / Evenimente de tip mesaj	Resurse utilizate de procesorul nMPRA și planificatorul nHSE (inclusiv suportul necesar pentru planificarea dinamică)	
	Slice LUTs (LUTs as Logic+LUTs as Memory)	Slice Registers (FFs+Latch)
nMPRA4/ grMutexi4/ grERFi4	10289	4228
nMPRA8/ grMutexi8/ grERFi8	18061	10318
nMPRA16/ grMutexi16/ grERFi16	29495	15988

În Figura 6-3 este reprezentat grafic necesarul de resurse pentru trei variante ale procesorului nMPRA, cu 4, 8 și respectiv 16 sCPUi/mutex-uri/evenimente de comunicație inter-task.

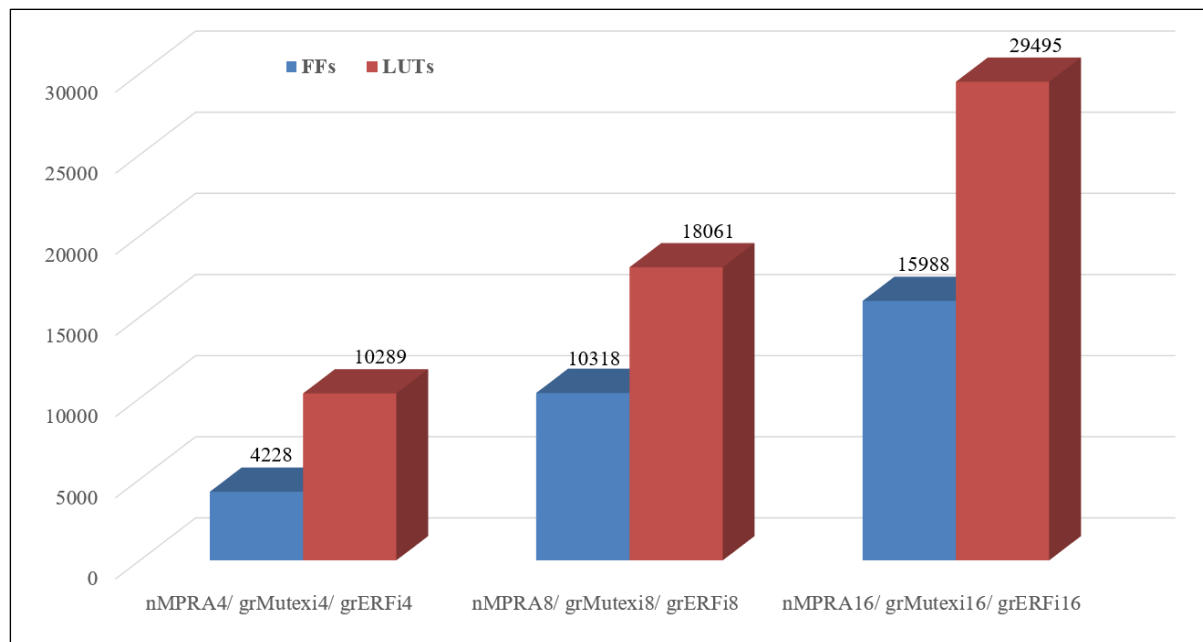


Figura 6-3 Resursele utilizate de procesorul nMPRA. *LUTs*: Look Up Table (blocul de bază într-un FPGA); *FFs*: Flip-Flop (element de bază pentru memorarea unui bit)

Deoarece implementarea acestui procesor fără mecanismele de sincronizare și comunicație inter-task nu era una realistă, s-a inclus aici atât necesarul de resurse pentru implementarea acestora în hardware cât și implementarea planificatorului static și dinamic. În implementările din Figura 6-3, nMPRA16 reprezintă o arhitectură nMPRA cu 16 sCPUi iar grMutexi16 și grERFi16 indică implementarea în hardware a resurselor necesare pentru 16 mutex-uri și 16 evenimente de sincronizare prin mesaje.

După cum se poate observa, necesarul de resurse pentru implementarea părții secvențiale și combinaționale a procesorului nu crește proporțional cu multiplicarea numărului de elemente din componența procesorului implementat.

În Tabelul 6-3 este prezentat necesarul de resurse pentru implementarea întregului proiect SoC ce înglobează procesorul nMPRA. De remarcat faptul că în acest tabel sunt incluse și

resursele utilizate pentru implementarea on-chip a memoriei dual-port pentru instrucțiuni și date, implementarea elementelor necesare interfețelor utilizator, inclusiv proiectarea bufferelor și a mecanismului de supra-eșantionare a datelor recepționate prin UART în cazul modului Bootloader, modul necesar pentru încărcarea instrucțiunilor în memoria program.

Tabelul 6-3: Utilizarea resurselor la nivelul proiectului SoC ce integrează procesorul nMPRA cu sCPU4, sCPU8 și sCPU16

Resurse utilizate de proiectul SoC	LUT	LUTRAM	FF	BRAM	IO	BUFG	MMCM
nMPRA4/ grMutexi4/ grERFi4	11596	908	1506	148	28	16	1
nMPRA8/ grMutexi8/ grERFi8	19371	1084	2203	148	28	16	1
nMPRA16/ grMutexi16/ grERFi16	30809	1436	3596	148	28	16	1
<i>Resurse disponibile</i>	<i>303600</i>	<i>130800</i>	<i>607200</i>	<i>1030</i>	<i>700</i>	<i>32</i>	<i>14</i>

Datele prezentate în Figura 6-4 și Figura 6-5 și corespund unei implementări a proiectului SoC ce conține procesorul nMPRA cu 16 semiprocsoare, 16 mutex-uri, 16 regiștri de comunicație prin mesaje și patru întreruperi externe. Necesarul de resurse pentru implementarea procesorului și consumul energetic redus îl fac ideal și pentru aplicații portabile, oferind astfel soluții scalabile și flexibile la cele mai înalte standarde calitative.

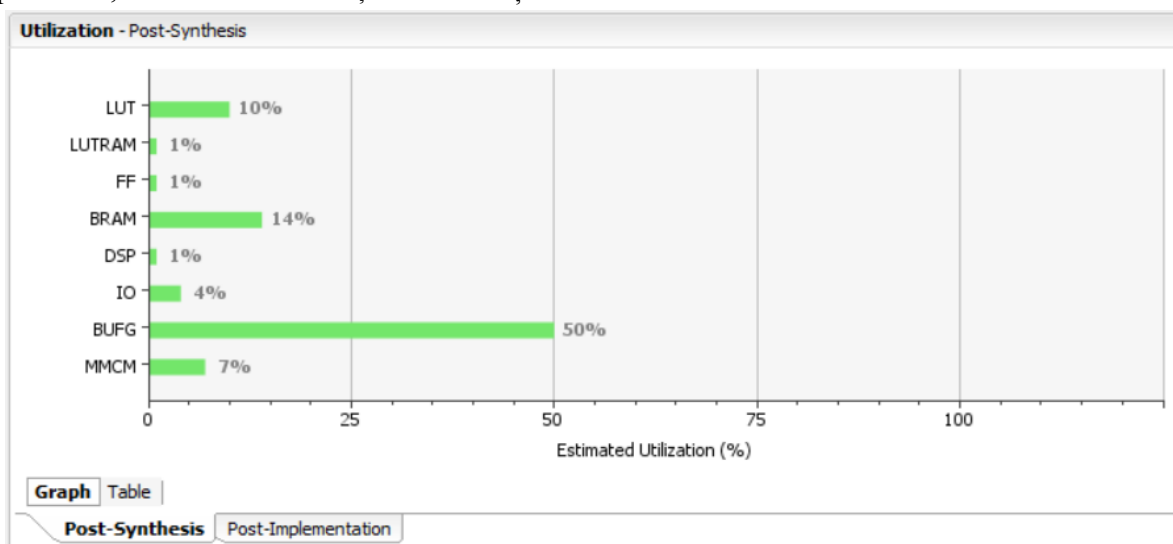


Figura 6-4 Reprezentarea grafică a resurselor utilizate pentru implementarea proiectului SoC ce include procesorul nMPRA cu sCPU16

Utilization - Post-Synthesis			
Resource	Estimation	Available	Utilization %
LUT	30809	303600	10.15
LUTRAM	1436	130800	1.10
FF	3596	607200	0.59
BRAM	148	1030	14.37
DSP	4	2800	0.14
IO	28	700	4.00
BUFG	16	32	50.00
MMCM	1	14	7.14

Figura 6-5 Necesarul de resurse pentru realizarea proiectului SoC ce include procesorul nMPRA cu sCPU16

6.3. Implementarea și analiza comparativă a performanțelor arhitecturii hardware propuse pentru SOTR

Tabelul 6-4 prezintă o comparație între arhitectura nMPRA și alte câteva implementări de procesor propuse și descrise în literatura de specialitate. Deoarece implementarea acestor arhitecturi de procesor a fost realizată utilizând diferite platforme de dezvoltare, s-a încercat realizarea unei comparații realiste între aceste implementări.

Tabelul 6-4: Comparație cu alte implementări de procesor

	Implementare planificator	Frecvență	Platforma utilizată	Pipeline/Implementare	Tipul arhitecturii
nMPRA4 [4]	MIPS32	Static și dinamic în HW	33MHz	Xilinx Virtex-7 XC7VX485T-2FFG1761C	5 etaje pipeline/ Single-core
FlexPret [17]	RISC-V	Static și dinamic (EDF, rate-monotonic)/ SW	80MHz	Xilinx Virtex-5 XC5VLX110T	5 etaje pipeline fine-grained multithreaded/ Single-core
Merasa [20], [21]	MERASA bazat pe SMT cores	Round Robin implementat în HW/SW	25MHz (real-time bus)	Altera Stratix II EP2S180F1020C3	Două benzi de asamblare cu 5 etaje pipeline/Multi-core
MicroBlaze [64]	MicroBlaze Architecture (RISC)	SW	Max 396Mhz (Virtex-7)	Xilinx Virtex-7 XC7VX485T-ffg1761-3	3 etaje pipeline/Single-core
ARPA-MT [65]	Implementare SMT bazată pe MIPS32	HW	42.8MHz (4 contexte)	Spartan-3 FPGA	5 etaje pipeline/Single-core
Amber 23 [66], [67]	ARM-compatible RISC 32 biți	SW	40MHz - Spartan-6 /80MHz - Virtex-6	Xilinx Spartan-6 SP605 xc6slx45t-ffg484-3/ Virtex-6	Amber 23 - 3 etaje pipeline/Single-core, Amber 25 - 5 etaje pipeline/Single-core

Tabelul 6-5 prezintă necesarul de memorie pentru cinci arhitecturi de procesor și diferite configurații ale acestora, astfel încât se poate face o comparație cu necesarul de resurse pentru implementarea procesorului nMPRA. Valorile empirice din Tabelul 6-5 sunt aproximative, în funcție de versiunea implementării și platforma utilizată pentru dezvoltarea procesorului. De exemplu, procesorul *FlexPRET-4T-TI* reprezintă varianta care implementează și instrucțiunile de timp [17], pentru exprimarea semanticii în timp real și îmbunătățirea performanțelor arhitecturii RISC-V ISA.

Tabelul 6-5: Resursele necesare implementării arhitecturilor de procesor propuse în literatura de specialitate

Implementare/Resurse utilizate/Platforma	FFs	LUTs	Platforma utilizată
nMPRA4 (4 contexte – sCPU4, 33MHz)	4228	10289	Xilinx Virtex-7 XC7VX485T-2FFG1761C
MicroBlaze [64] (Typical, 255MHz)	1627	1913	Xilinx Virtex-7 XC7VX485T-ffg1761-3
MicroBlaze [64] (Maximum Performance, 231MHz)	2917	3725	Xilinx Virtex-7 XC7VX485T-ffg1761-3
FlexPret [17] (FlexPRET-4T, 80MHz)	908	3943	Xilinx Virtex-5 XC5VLX110T
FlexPret [17] (FlexPRET-4T-TI, 80MHz)	1622	4323	Xilinx Virtex-5 XC5VLX110T
ARPA-MT [65] (4 contexte, 42.8MHz)	7562	18854	Spartan-3 FPGAXC3S1500
Amber 23 [66], [67] (40MHz)	4912	8732	Xilinx Spartan-6 SP605xc6slx45t-ffg484-3
nMPRA-ARM [68] (4 fire de execuție)	10842	18032	Xilinx Spartan-6 SP605xc6slx45t-ffg484-3

Graficul din Figura 6-6 ilustrează resursele folosite pentru implementarea unui număr de 6 procesoare sau variante ale acestora. Numărul de FFs și LUTs pentru fiecare procesor sunt furnizate de autori în articolele [17], [64-68], acestea putând varia în funcție de implementare, platforma utilizată, funcțiile implementate la nivelul procesorului și multiplicarea contextelor task-urilor. În afară de procesoarele MicroBlaze și Amber 23, toate implementările de procesor luate în considerare au folosit multiplicarea totală sau parțială a resurselor pentru patru sCPUi/task-uri/thread-uri/contexte. Rezultatele obținute de arhitecții acestor procesoare sunt relative la arhitectura de procesor implementat, acest lucru având un impact deosebit asupra datelor prezentate în Figura 6-6.

În concluzie, putem spune că utilizarea arhitecturii nMPRA cu 16 sCPUi și planificator dinamic se justifică pe deplin pentru avantajele pe care ea le aduce, iar raportul performanță/cost implementare este unul satisfăcător [69]. Această implementare de planificator poate fi îmbunătățită prin optimizarea automatului cu stări finite [70] ce stă la baza funcționării modulului nHSE, cât și perfecționarea sistemului de întreruperi și a modulului de tratare a excepțiilor.

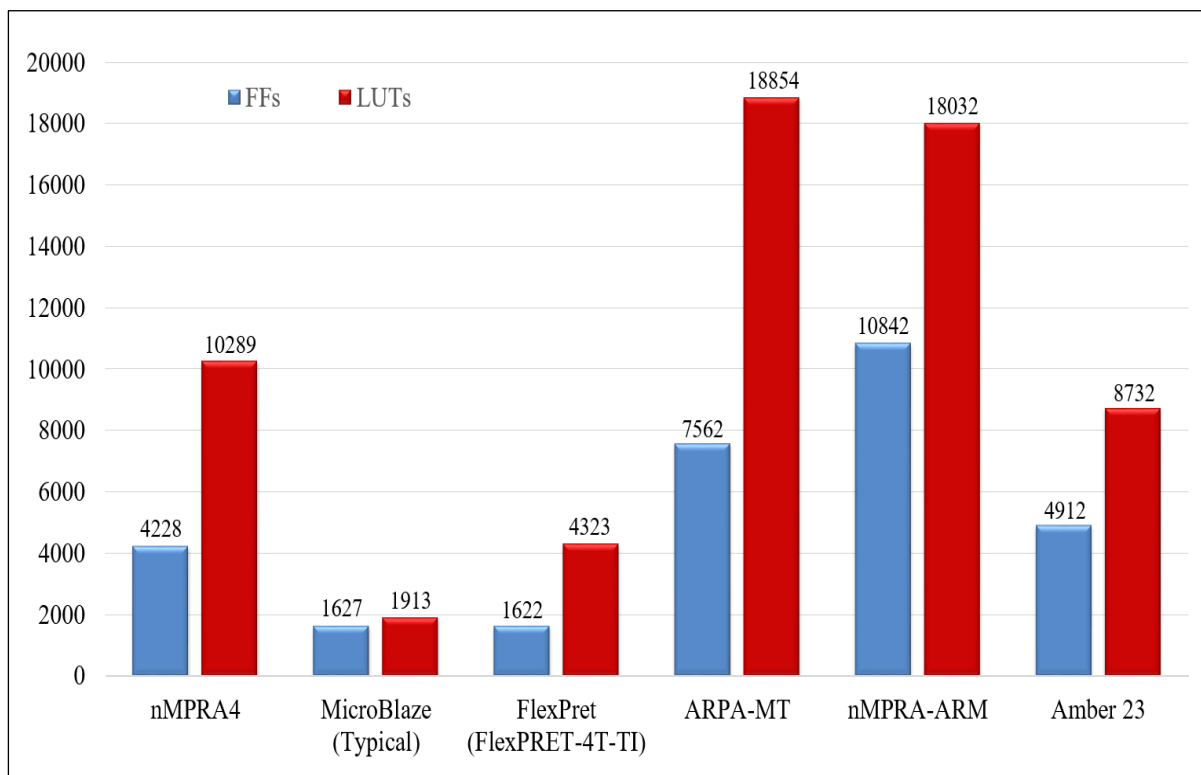


Figura 6-6 Resursele utilizate pentru implementarea procesorului nMPRA4 (sCPU4) cât și a altor proiecte descrise în literatura de specialitate

7. Concluzii, direcții viitoare de cercetare și contribuții

7.1. Concluzii finale

Soluțiile de îmbunătățire rezultate în urma proiectului de doctorat vor conduce la un procesor nMPRA care poate influența semnificativ domeniul sistemelor de operare cu funcții implementate în hardware. **Această nouă implementare este caracterizată de predictibilitate, timpi foarte mici de comutare a task-urilor (1 - 3 cicli mașină), răspuns rapid la evenimentele externe asincrone (maxim 1.5 cicli mașină), un sistem de întreruperi distribuit cu asignarea flexibilă a întreruperilor la task-uri, garantând totodată un timp de răspuns predictibil și fezabilitatea planificării task-urilor critice.** Arhitectura nMPRA va asigura o izolare spațială și temporală a task-urilor, utilizând izolarea bazată pe hardware pentru firele de execuție critice. Aceasta va permite eliminarea supra aprovizionării task-urilor cu resurse, garantând deci o utilizare mai bună a acestora și o minimizare a timpului de proiectare datorită unei prototipări și testări rapide.

Caracterul predictibil al procesorului implementat este definit de capacitatea de schimbare a contextelor task-urilor într-un ciclu de ceas sau în trei cicli, când se dorește asigurarea consistenței memoriei. Printr-o implementare specială a fișierului de regiștri folosind tehnologia logicii programabile, fiecărui sCPUi îi corespunde un banc de regiștri. Multiplicarea regiștrilor pipeline garantează o izolare hardware a task-urilor sub controlul direct al modulului nHSE. Arhitectura nMPRA asigură performanțe superioare în ceea ce privește timpul de răspuns la evenimentele externe și timpul necesar comutării contextelor, fiind pretabilă pentru aplicațiile de timp real de mici dimensiuni datorită consumului de resurse necesar multiplicării regiștrilor.

Elementele de originalitate și inovație pe care implementarea obiectivelor le aduce domeniului, raportat la stadiul actual al cercetării, se referă la o serie de soluții și implementări noi în scopul stabilirii unor performanțe ridicate în domeniu. Astfel, putem enumera următoarele elemente de performanță:

- Obținerea unui timp de comutare a task-urilor de 1 ciclu mașină (dacă se renunță la accesul direct la variabile globale plasate în memoria comună) și execuția instrucțiunilor în același număr de cicli mașină prin eliminarea hazardurilor.
- Implementarea în hardware a unor mecanisme de sincronizare și comunicație inter-task (mutex-uri sau mesaje) cu cea mai redusă energie consumată și testarea siguranței de funcționare a sistemului chiar și în cazul unor situații de hazard.
- Implementarea facilă a planificării sistemelor cu criticitate mixtă prin folosirea planificării dinamice a task-urilor și interoperabilitate dovedită cu o calitate extensivă. La nivelul software nu trebuie implementat nici un sistem de operare în timp real, cu excepția cazului când se folosește planificarea dinamică și planificatorul nu este implementat în hardware.
- Implementarea unui sistem de întreruperi distribuit care permite atașarea întreruperii oricărui task. Prin atașare, întreruperea dobândește prioritatea task-ului la care a fost atașată. Răspunsul la întreruperi este foarte predictibil și scurt, transferul la celula capcană făcându-se automat.

- Implementarea în hardware a unui sistem de evenimente (întreruperi, evenimente de tip timp, evenimente de tip mutex sau mesaj), care moștenesc prioritatea task-ului. Timpul de răspuns la aceste evenimente, dacă task-ul este mai prioritar decât task-ul curent, este predictibil având durată maximă de 1.5 cicli mașină. Sincronizarea task-ului cu aceste evenimente (cu unul, mai multe sau cu toate simultan) se face în mod eficient cu o singură instrucțiune în limbaj de asamblare.

7.2.Direcții viitoare de cercetare

Prin intermediul acestor obiective, se vor continua cercetările referitoare la implementarea în hardware a sistemelor de operare în timp real având în vedere arhitectura nMPRA și componenta sa definitorie nHSE. Menționez că aceste direcții de cercetare reprezintă o continuare și nu o suprapunere a cercetărilor existente. Așadar, se va realiza îmbunătățirea performanțelor arhitecturilor nMPRA și nMPRA-MT, deci a sistemelor de operare în timp real implementate prin hardware.

Cu toate acestea, arhitectura este susceptibilă la îmbunătățiri cum ar fi:

- Definirea explicită a unui model de memorie și periferice pentru a asigura conformitatea cu standardele europene;
- Îmbunătățirea comportamentului de tip PRET și FlexPRET prin eliminarea totală sau parțială a hazardurilor, îmbunătățirea predictibilității (hazardurile de date pot furniza timpi diferiți pentru execuția aceleași instrucțiuni) și creșterea gradului de paralelizare a execuției instrucțiunilor;
- Implementarea în software și mai apoi în hardware a unor algoritmi de planificare dinamică pentru sistemele cu criticitate mixtă ce sunt ușor adaptabili diferitelor cerințe;
- Implementarea nHSE sub formă de coprocesor pentru a profita de facilitățile compilatoarelor profesionale ale arhitecturilor folosite (RISC-V);
- Implementarea unei game diversificate de configurații și de eventuale programe de evaluare a performanțelor procesorului în FPGA, permițând adaptarea facilă la cerințele unei aplicații de timp real;
- Propunerea unor modele de aplicație folosite în domeniul industrial și automotive;
- Studiul posibilității trecerii la producția de masă.

Proiectul lasă loc viitorilor cercetători să-și exprime satisfacția de a îmbunătăți calea de date, să implementeze o unitate de protecție a memoriei sau să proiecteze o versiune de procesor nMPRA quad-core. În cazul aplicațiilor critice, implementarea unei unități de protecție a memoriei (MPU) poate fi o extensie necesară pentru arhitectura nMPRA. Acest modul trebuie să furnizeze protecția memoriei, stabilitate, siguranță și garantarea performanțelor crescute ale procesorului în aplicațiile de timp real. Această componentă, ideală pentru aplicațiile înglobate, trebuie să răspundă standardelor de siguranță riguroase, necesitând certificarea cerută de domenii precum: automotive, electronică medicală sau domeniul industrial. Cu toate acestea, dezvoltarea arhitecturii nMPRA trebuie să fie asigurată într-un timp acceptabil pentru acest domeniu. Pentru dezvoltatorul de dispozitive înglobate, este de obicei o provocare obținerea la timp a unui sistem care să funcționeze conform specificațiilor de marketing. Performanțele cost-eficacitate reprezintă un pas evolutiv și un

avantaj substanțial care se transformă în salvarea resurselor de proiectare a sistemelor din domeniile mai sus menționate.

7.3.Contribuții

Cercetările efectuate pentru elaborarea tezei de doctorat s-au finalizat printr-o serie de contribuții practice iar rezultatele științifice au fost diseminate în literatura de specialitate. Utilizând kit-ul de dezvoltare Virtex-7 s-a validat arhitectura nMPRA pentru 4, 8 și 16 task-uri, care include: integrarea și implementarea în Verilog a elementelor de bază ale procesorului (unitatea de control, memoria de date și instrucțiuni, unitatea de detecție a hazardului, unitatea de redirecționare, ALU, multiplexoare); implementarea în Verilog a structurilor cu resurse multiplicare și multiplexate specifice procesorului nMPRA: PC, regiștrii pipeline, fișierul de regiștri precum și toate elementele de memorare din banda de asamblare. S-a realizat simularea și implementarea arhitecturii nMPRA cu 4, 8 respectiv 16 sCPUi, precum și testarea individuală a structurilor cu multiplexare de resurse. S-a efectuat o analiză a puterii consumate de fiecare în parte și apoi o comparație pentru a demonstra că este îndeplinită cerința sistemelor de timp-real de consum cât mai mic de putere, pentru a evita supradimensionarea platformelor.

Utilizând o platformă de dezvoltare bazată pe FPGA, prin intermediul unui proiect SoC, s-a dezvoltat, simulat și validat în hardware implementarea arhitecturii de procesor dedicat nMPRA ce înglobează planificatorul hardware integrat nHSE. Datorită faptului că arhitectura procesorului nMPRA se bazează pe multiplicarea resurselor precum PC, fișierul de regiștri și regiștrii pipeline, multiplicarea acestor resurse pentru un task i reprezentând un semiprocessor notat sCPUi, putem afirma că rezultatele experimentale validează implementarea în practică a aspectelor teoretice prezentate în capitolul 4, obținând astfel timpi foarte mici pentru operațiile de schimbare a contextelor. Efectul hardware al acestor performanțe deosebite, atât de necesare în STR critice, este reprezentat de consumul de memorie pentru multiplicarea în hardware a resurselor multiplexate. Acest factor crește în funcție de numărul de semiprocesoare implementate, numărul acestora fiind cuprins în intervalul 4-16 deoarece arhitectura validată în această teză de doctorat este propusă pentru aplicațiile înglobate de mici dimensiuni din sectorul industrial, medical sau automotive.

Prin reducerea jitter-ului în cazul tratării întreruperilor asincrone și eliminarea totală a incertitudinilor ce planează asupra limitei de planificare a setului de task-uri s-a obținut o îmbunătățire semnificativă a predictibilității sistemului per ansamblu.

7.3.1. Contribuții teoretice

- Realizarea unei comparații între cele mai reprezentative arhitecturi de procesor prezente în literatura de specialitate și procesorul propus în această teză de doctorat;
- În urma analizei procesorului validat în această teză de doctorat s-a propus o arhitectură de implementare SoC a nMPRA și nHSE care include:
 1. Cercetări pentru integrarea etajelor benzii de asamblare în arhitectura nMPRA;
 2. Cercetări asupra modului în care se realizează multiplicarea tuturor regiștrilor multiplexați (PC, fișierul de regiștri, regiștrii pipeline, COP0, unitatea de control și unitatea de înmulțire);
 3. Studiu pentru integrarea unității nHSE în COP2 din arhitectura MIPS32;

4. Reorganizarea regiștrilor și instrucțiunilor dedicate planificatorului de timp real nHSE;
5. Cercetări cu privire la maparea tuturor regiștrilor nHSE în spațiul fișierului de regiștri corespunzător COP2.

7.3.2. Contribuții practice

- Proiectarea procesorului nMPRA utilizând limbajul de descriere hardware Verilog cu sCPU4, sCPU8 și sCPU16, scalabil, parametrizat și bine documentat:
 1. Proiectarea resurselor multiplexate specifice procesorului nMPRA: regiștrii PC, fișierul de regiștri multiplicat pentru fiecare sCPUi, regiștrii pipeline precum și orice element de memorare din banda de asamblare;
 2. Integrarea modulelor de bază ale procesorului MIPS32 conform specificațiilor arhitecturii nMPRA: unitatea de control, COP0, controlerul de memorie, unitatea de detecție a hazardului, unitatea de avansare a datelor și unitatea ALU;
 3. Implementarea în Verilog a regiștrilor și a instrucțiunilor pentru controlul planificatorului nHSE;
 4. Implementarea modulului nHSE.v (ANEXA III) ce înglobează atât blocul de timer-e folosit pentru generarea evenimentelor de tip deadline precum și a semnalelor de control pentru selecția resurselor multiplexate;
 5. Implementarea mecanismelor de sincronizare și comunicație inter-task în hardware care include și evenimentele nMPRA;
 6. Îmbunătățirea legăturii cu memoria dual-port pentru cod și date;
- Îmbunătățirea predictibilității și a timpului de răspuns la evenimente multiple simultane și întreruperi multiple simultane prin utilizarea codificatoarelor de prioritate și a transferului direct către handler-ele de evenimente utilizând celule capcană;
- Implementarea în Verilog a unui algoritm preemptiv bazat pe priorități pentru planificarea unui set de task-uri;
- Scrierea unor programe de test pentru verificarea modulului nHSE, a procesorului nMPRA în cazul hazardului de date și a excepțiilor, tratarea întreruperilor externe asincrone și a mecanismelor de sincronizare și comunicație inter-task;
- Modelarea și testarea proiectului SoC și a procesorului nMPRA utilizând kit-ul de dezvoltare Virtex-7 care constă din validarea individuală a modulelor arhitecturii nMPRA, inclusiv a resurselor multiplexate care stau la baza procesorului;
- Realizarea necesarului de memorie (Tabelul 6-1) pentru implementarea procesorului nMPRA cu sCPU4, sCPU8 și sCPU16, precum și analiza necesarului de resurse pentru implementarea arhitecturii nMPRA și a întregului proiect SoC utilizând kit-ul de dezvoltare Virtex-7;
- Studiu privind raportul de performanță/cost pentru o gamă diversificată de configurații nMPRA (având în vedere gradul de multiplicare 4, 8, 16, etc.) și implicit a nHSE (având în vedere numărul de mutex-uri și numărul de evenimente de tip mesaj) în FPGA.

7.3.3. Diseminarea rezultatelor

Contribuțiile originale au fost comunicate și publicate la nivel național și internațional în cadrul revistelor de specialitate, conferințelor și rapoartelor de cercetare, după cum urmează:

Articole în reviste cotate ISI:

1. **I. Zagan**, V. G. Găitan, “Improving the Performances of the nMPRA Processor using a Custom Interrupt Management Scheduling Policy”, *Advances in Electrical and Computer Engineering (AECE)*, Volume 16, Issue 4, 30/11/2016, pp. 45-50. Factor Impact 2016: 0.459
2. **I. Zagan**, V. G. Găitan, A. Petrariu, A. Brezulianu, “Healthcare IoT m-GreenCARDIO Remote Cardiac Monitoring System – Concept, Theory of Operation and Implementation”, acceptat pentru publicare: *Advances in Electrical and Computer Engineering (AECE)*, vol., no. , Mai 2017.

Articole în reviste indexate ISI:

3. N. C. Găitan, **I. Zagan**, V. G. Găitan, “Predictable CPU Architecture Designed for Small Real-Time Application - Concept and Theory of Operation”, *International Journal of Advanced Computer Science and Applications (IJACSA)*, 6(4). doi: 10.14569/IJACSA.2015.060406, U.S ISSN: 2156-5570(Online), pp. 47-52, 2015.
4. **I. Zagan**, N. C. Găitan, V. G. Găitan, “An Approach of nMPRA Architecture using Hardware Implemented Support for Event Prioritization and Treating”, *International Journal of Advanced Computer Science and Applications (IJACSA)*, Vol. 8, No. 2, doi: 10.14569/IJACSA.2017.080206, 2017.

Articole în jurnale internaționale indexate BDI:

5. **I. Zagan**, V. G. Găitan, “Improving the performance of CPU architectures by reducing the Operating System overhead (Extended Version)”, *The Scientific Journal of Riga Technical University - Electrical, Control and Communication Engineering*, ISSN: 2255-9140 (print), Iulie 2016, vol. 10, pp. 13-22, doi: 10.1515/ecce-2016-0002.

Articole în conferințe indexate ISI:

6. N. C. Găitan, V. G. Găitan, I. Ungurean, **I. Zagan**, “Methods to improve the performances of the real-time operating systems for small microcontrollers”, 20th International Conference on Control Systems and Computer Science (CSCS), București, România, 27-29 Mai 2015, ISBN: 978-1-4799-1779-2, doi: 10.1109/CSCS.2015.10, pp. 261-266.
7. **I. Zagan**, “Improving the performance of CPU architectures by reducing the Operating System overhead”, *The 3rd IEEE Workshop on Advances in Information, Electronic and Electrical Engineering AIEEE’2015*, Vol., No., pp.1-6, 13 - 14 Noiembrie 2015, Riga, Lituania, doi: 10.1109/AIEEE.2015.7367279.
8. **I. Zagan**, V. G. Găitan, “Schedulability Analysis of nMPRA Processor based on Multithreaded Execution”, 13rt International Conference on Development and Application Systems (DAS), Suceava, România, 19-21 Mai, 2016.

Articole în conferințe internaționale indexate BDI:

9. **I. Zagan**, V. G. Găitan, “Predictable CPU Architecture Designed for Small Real-Time Applications – Implementation Results”, 3rd International Conference on Advances in Computing, Electronics and Communication (ACEC), 10 - 11 Octombrie 2015, Zurich, Elveția, ISBN: 978-1-63248-064-4, doi: 10.15224/ 978-1-63248-064-4-29, pp. 143-150.

Publicat în jurnalul IRED: International Journal of Advances in Computer Science and Its Applications (IJCSIA), 2016, Vol. 6, Issue 1: ISSN: 2250-3765, data publicării: 18/04/2016, pp. 141-148.

10. **I. Zagan**, “Real-time evaluation of nMPRA CPU Architecture based on Multithreaded Execution”, 8th International Conference on Computer Science and Information Technology, ICCSIT 2015, 10 - 11 Decembrie 2015, Amsterdam, Olanda.

Publicat în jurnalul: International Journal of Computer and Electrical Engineering (IJCEE), Vol. 7(6), ISSN: 1793-8163, doi: 10.17706/IJCEE.2015.7.6.424-429, pp. 424-429, 2015.

11. **I. Zagan**, V. G. Găitan, “Improving the Performances of the nMPRA Architecture by Implementing Specific Functions in Hardware”, 19th International Conference on Digital Circuits and Microarchitecture Technologies (ICDCMT 2017), Berlin, Germania, 21-22 Mai, World Academy of Science, Engineering and Technology, International Journal of Electrical, Computer, Energetic, Electronic and Communication Engineering, Vol. 11, No. 5, pp. 417 - 424, 2017.

12. **I. Zagan**, V. G. Găitan, “CPU Architecture Based on Static Hardware Scheduler Engine and Multiple Pipeline Registers”, acceptat pentru prezentare: 19th International Conference on Advanced Computing Systems and Microarchitecture (ICACSM), Zurich, Elveția, 15 - 16 Septembrie, 2017.

Articole în buletine științifice și seminarii:

13. N. C. Găitan, **I. Zagan**, V. G. Găitan, “IMPROVING THE PREDICTABILITY OF NMPRA AND NHSE ARCHITECTURE”, Bulletin of the Polytechnic Institute of Iași, Automatic Control and Computer Science Section, România, fasc. 1/2015. ISSN 1220-2169, pp. 27-38.
14. **I. Zagan**, V. G. Găitan, “CPU architecture description based on fine-grained multithreading and hardware scheduler engine”, Sisteme Distribuite, Vol: XII, Suceava, România, 2014, ISSN/ISBN: 1842-6808.
15. **I. Zagan**, N. C. Găitan, V. G. Găitan, “Scheduling real-time tasks with nMPRA architecture for embedded applications”, Sisteme Distribuite, Vol: XIII, 16 Decembrie 2015, Suceava, România, ISSN/ISBN: 1842-6808.

Bibliografie

1. M. Shahbazi, P. Poure, S. Saadate, and M. R. Zolghadri, "FPGA-Based Reconfigurable Control for Fault-Tolerant Back-to-Back Converter Without Redundancy", *IEEE Transactions on Industrial Electronics*, vol. 60, no. 8, pp. 3360-3371, Aug. 2013.
2. Electronica-Azi, nr. 210, Decembrie 2016, Anul XV, <http://electronica-azi.ro>.
3. G. C. Buttazzo, "Hard Real-Time Computing Systems", *Predictable Scheduling Algorithms and Applications*, Third edition, 2011.
4. V. G. Găitan, N. C. Găitan, and I. Ungurean, "CPU Architecture Based on a Hardware Scheduler and Independent Pipeline Registers", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 23, issue 9, pp. 1661-1674, Sept. 2015, doi: 10.1109/TVLSI.2014.2346542.
5. E. Dodiù and V. G. Gaitan, "Custom designed CPU architecture based on a hardware scheduler and independent pipeline registers – concept and theory of operation", *2012 IEEE EIT International Conference on Electro-Information Technology*, Indianapolis, IN, USA, 6-8 May 2012, ISBN: 978-1-4673-0818-2, ISSN: 2154-0373.
6. M. Shahbazi, P. Poure, S. Saadate, and M. R. Zolghadri, "Fault-tolerant five-leg converter topology with FPGA-based reconfigurable control", *IEEE Trans. Ind. Electron.*, vol. 60, no. 6, pp. 2284–2294, Jun. 2013.
7. T. T. Phuong, K. Ohishi, Y. Yokokura, and C. Mitsantisuk, "FPGA-based high-performance force control system with friction-free and noise-free force observation", *IEEE Trans. Ind. Electron.*, vol. 61, no. 2, pp. 994-1008, Feb. 2014.
8. D. Andrews et al., "hthreads: A hardware/software co-designed multithreaded RTOS kernel", in *Proc. 10th IEEE Conf. Emerg. Technol. Factory Autom.*, pp. 331-338, Catania, Italy, Sep. 2005.
9. J. Agron, D. Andrews, "Hardware Microkernels for Heterogeneous Manycore Systems", *International Conference on Parallel Processing Workshops, ICPPW '09.*, vol., no., pp.19-26, 22-25 Sept. 2009, doi: 10.1109/ICPPW.2009.21.
10. L. Lindh, "Fastchart—A fast time deterministic CPU and hardware based real-time-kernel", in *Proc. Euromicro Workshop Real Time Syst.*, pp. 36–40, Paris-Orsay, France, Jun. 1991.
11. F. Stanischewski, "FASTCHART - Performance, Benefits and Disadvantages of the Architecture", *Proceedings Fifth Euromicro Workshop on Real-Time Systems*, vol., no., pp.246-250, 22-24 June 1993, doi: 10.1109/EMWRT.1993.639104.
12. M. Delvai, W. Huber, B. Rahbaran, and A. Steininger, "SPEAR – Design-Entscheidungen für den Scalable Processor for Embedded Applications in Real-time Environments", *Tagungsband of Austrochip 2001*, pages 25–32, Vienna, Austria, Oct. 2001.
13. E. Dodiù, V. G. Gaitan, and A. Graur, "Custom designed CPU architecture based on a hardware scheduler and independent pipeline registers – architecture description", *IEEE 35'th Jubilee International Convention on Information and Communication Technology, Electronics and Microelectronics*, Croatia, pp. 859-864, 24 May 2012, ISSN: 1847-3946, ISBN 978-953-233-069-4, INSPEC Accession Number: 12865464.
14. N. C. Găitan, **I. Zagan**, and V. G. Găitan, "Predictable CPU Architecture Designed for Small Real-Time Application - Concept and Theory of Operation", *International Journal of Advanced Computer Science and Applications (IJACSA)*, vol. 6, no. 4, 2015, doi: 10.14569/IJACSA.2015.060406.
15. S. A. Edwards, E. A. Lee, "The Case for the Precision Timed (PRET) Machine", *44th ACM/IEEE Design Automation Conference, DAC '07*, vol., no., pp. 264-265, 4-8 June 2007.
16. I. Liu, J. Reineke, and E. A. Lee, "A PRET architecture supporting concurrent programs with composable timing properties", *44th Asilomar Conference on Signals, Systems, and Computers*, pp. 2111-2115, November, 2010.
17. M. Zimmer, D. Broman, Chris Shaver, and E. A. Lee, "FlexPRET: A Processor Platform for Mixed-Criticality Systems", *Proceedings of the 20th IEEE Real-Time and Embedded Technology and Application Symposium (RTAS)*, Berlin, Germany, April 15-17, 2014.
18. M. Schoeberl, "A time predictable Java processor", *Proceedings of the Design, Automation and Test in Europe Conference*, vol. 1, 2006, doi: 10.1109/DATE.2006.244146.

19. M. Nadeem, M. Biglari-Abhari, and Z. Salcic, "JOP-plus - A processor for efficient execution of java programs extended with GALS concurrency", 17th Asia and South Pacific Design Automation Conference, vol., no., pp. 17-22, 30 Jan. - 2 Feb. 2012, doi: 10.1109/ASPDAC.2012.6164940.
20. F. Kluge and J. Wolf, "System-Level Software for a Multi-Core MERASA Processor", Institute of Computer Science, University of Augsburg, report, October 2009, URN: urn:nbn:de:bvb:384-opus4-11134.
21. T. Ungerer et al., "Merasa: Multicore execution of hard real-time applications supporting analyzability", IEEE, Micro, vol. 30, no. 5, pp. 66-75, 2010, doi: 10.1109/MM.2010.78.
22. D. May, "The XMOS XS1 Architecture", XMOS, 2009.
23. D. May, "The XMOS Architecture and XS1 Chips", Micro, IEEE, vol.32, no.6, pp. 28-37, Nov.-Dec. 2012, doi: 10.1109/MM.2012.87.
24. J. Kreuzinger, R. Marston, T. Ungerer, U. Brinkschulte, and C. Krakowski, "The Komodo project: thread-based event handling supported by a multithreaded Java microcontroller", Proceedings 25th EUROMICRO Conference, vol.2, no., pp. 122-128, 1999, doi: 10.1109/EURMIC.1999.794770.
25. X. Hua, C. Guo, H. Wu, D. Lautner and S. Ren, "Schedulability Analysis for Real-Time Task Set on Resource with Performance Degradation and Dual-Level Periodic Rejuvenations", IEEE Transactions on Computers, vol. 66, no. 3, pp. 553-559, March 1 2017, doi: 10.1109/TC.2016.2602833.
26. G. Yao, M. Bertogna, and G. Buttazzo, "Feasibility analysis under fixed priority scheduling with fixed preemption points", IEEE 19th International Conference on Embedded and Real-Time Computing Systems and Applications (2010), pp. 71-80, Macau, China, 23-25 Aug. 2010. doi: 10.1109/RTCSA.2010.40.
27. William Stallings, "Computer Organization and Architecture (10th Edition)", ISBN-13: 978-0134101613, ISBN-10: 0134101618, 2016.
28. K. Jeffay and D. L. Stone, "Accounting for interrupt handling costs in dynamic priority task systems", IEEE Real-Time Systems Symposium, pp. 212-221, 1-3 Dec. 1993, doi: 10.1109/REAL.1993.393497.
29. J. Echague, I. Ripoll, and A. Crespo, "Hard real-time preemptively scheduling with high context switch cost", In Proc. of the 7th Euromicro Workshop on Real-Time Systems, Odense, Denmark, 14-16 June, 1995, doi: 10.1109/EMWRTS.1995.514310.
30. D. A. Patterson, J. L. Hennessy, "Computer Organization and Design, The Hardware/Software Interface", 4th Edition, 2011, ISBN: 9780123747501.
31. MIPS® Architecture For Programmers Volume I-A: Introduction to the MIPS32® Architecture Document Number: MD00082 Revision 3.02 March 21, 2011, Available: <https://courses.engr.illinois.edu/cs426/Resources/MIPS32INT-AFP-03.02.pdf>. (May 2016)
32. E. Dodiu, "Planificator de timp real implementat hardware pentru sisteme embedded bazate pe FPGA", teză de doctorat, Universitatea Ștefan cel Mare Suceava, România, 2013.
33. Grant Ayers, eXtensible Utah Multicore (XUM) project at the University of Utah, 2011-2012, <http://opencores.org/project,mips32r1> (Accesat: Sept. 2015).
34. N. C. Găitan, **I. Zagan**, V. G. Găitan, "IMPROVING THE PREDICTABILITY OF NMPRA AND NHSE ARCHITECTURE", Bulletin of the Polytechnic Institute of Iasi, Automatic Control and Computer Science Section, România, pp. 27-38, fasc. 1/2015, ISSN 1220-2169.
35. D. A. Patterson, J. L. Hennessy, "Computer Organization and Design MIPS Edition: The Hardware/Software Interface", 5th Edition, Sep. 2013, ISBN: 9780124077263.
36. N. C. Găitan, V. G. Găitan, I. Ungurean, **I. Zagan**, "Methods to improve the performances of the real-time operating systems for small microcontrollers", 20th International Conference on Control Systems and Computer Science (CSCS20), Bucharest, Romania, pp. 261-266, May 2015, doi: 10.1109/CSCS.2015.10.
37. V. G. Găitan, "The Highly Functional Distributed System", 4th International Symposium on Automatic Control and Computer Science, Vol. 1, pp. 277-282, Iași 1993, .
38. N. C. Găitan, V. G. Găitan, and E. E Ciobanu Moisuc, "Improving interrupt handling in the nMPRA", 2014 International Conference on Development and Application Systems (DAS), May 2014. pp. 11-15, doi: 10.1109/DAAS.2014.6842419.
39. E. Dodiu, V. G. Gaitan, and A. Graur, "Improving commercial RTOS performance using a custom interrupt management scheduling policy", ACC'10 Proceedings of the 2010 International Conference on Applied Computing Conference, Timișoara, Romania, 2010.

40. N. C. Găitan and L. Andrieș, "Using Dual Priority scheduling to improve the resource utilization in the nMPRA microcontrollers" 2014 International Conference on Development and Application Systems (DAS), May 2014, Suceava, România, pp. 73-78, doi: 10.1109/DAAS.2014.6842431.
41. E. E Ciobanu Moisuc, A. B. Larionescu, and V. G. Găitan, "Hardware Event Treating in nMPRA", 2014 International Conference on Development and Application Systems (DAS), May 2014, Suceava, România, pp. 66-69, doi: 10.1109/DAAS.2014.6842429.
42. **I. Zagan** and V. G. Găitan, "Predictable CPU Architecture Designed for Small Real-Time Applications – Implementation Results", 3rd International Conference on Advances in Computing, Electronics and Communication (ACEC), 10 - 11 Oct. 2015, Zurich, Switzerland, doi: 10.15224/978-1-63248-064-4-29.
43. **I. Zagan**, "Improving the performance of CPU architectures by reducing the Operating System overhead", 3rd IEEE Workshop on Advances in Information, Electronic and Electrical Engineering AIEEE'2015, Riga, Latvia, pp. 1-6, 13-14 Nov. 2015, doi: 10.1109/AIEEE.2015.7367279.
44. S. Kellinman and J. Eykholt, "Interrupts as threads", ACM SIGOPS Operating Systems Review, vol. 29, no. 2, pp. 21-26, Apr. 1995, doi:10.1145/202213.202217.
45. **I. Zagan** and V. G. Găitan, "Improving the performance of CPU architectures by reducing the Operating System overhead (Extended Version)", Electrical, Control and Communication Engineering, vol. 10, no. 1, pp. 13-22, ISSN (Online) 2255-9159, January 2017, doi: <https://doi.org/10.1515/ecce-2016-0002>.
46. B. Meakin, "MULTICORE SYSTEM DESIGN WITH XUM: THE EXTENSIBLE UTAH MULTICORE PROJECT", A thesis submitted to the faculty of The University of Utah in partial fulfillment of the requirements for the degree of Master of Science in Computer Science, The University of Utah, May 2010.
47. Vivado Design Suite User Guide, Programming and Debugging, XILINX, UG908 (v2014.1), May 30, 2014 http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug908-vivado-programming-debugging.pdf, (Accesat: Feb. 2017)
48. Vivado Design Suite User Guide, Getting Started, XILINX, UG910 (v2014.1) 2 April, 2014, http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug910-vivado-getting-started.pdf, (Accesat: Aug. 2016).
49. VC707 Evaluation Board for the Virtex-7 FPGA, User Guide, XILINX, UG885 (v1.7.1) 12 August, 2016 https://www.xilinx.com/support/documentation/boards_and_kits/vc707/ug885_VC707_Eval_Bd.pdf, (Accesat: Feb. 2017).
50. Getting Started with the Virtex-7 FPGA VC707 Evaluation Kit, XILINX, UG848 (v1.4.1) October 14, 2015, https://www.xilinx.com/support/documentation/boards_and_kits/vc707/ug848-VC707-getting-started-guide.pdf, (Accesat: Aug. 2016).
51. K. Arshak, E. Jafer and C. Ibala, "Testing FPGA based digital system using XILINX ChipScope logic analyzer", 2006 29th International Spring Seminar on Electronics Technology, St. Marienthal, 2006, pp. 355-360, doi: 10.1109/ISSE.2006.365129.
52. Samir Palnitkar, "Verilog HDL: A Guide to Digital Design and Synthesis, Second Edition", Feb. 2003, ISBN: 0-13-044911-3.
53. Peter J. Ashenden, "Digital Design An Embedded Systems Approach Using Verilog", 10 Sep. 2007, ISBN: 978-0-12-369527-7, eBook ISBN: 9780080553115.
54. **I. Zagan** and V. G. Găitan, "Improving the Performances of the nMPRA Architecture by Implementing Specific Functions in Hardware", in 19th International Conference on Digital Circuits and Microarchitecture Technologies (ICDCMT 2017), Berlin, Germany, May 21-22. World Academy of Science, Engineering and Technology, International Journal of Electrical, Computer, Energetic, Electronic and Communication Engineering, Vol. 11, No. 5, pp. 417-424, 2017.
55. **I. Zagan** and V. G. Găitan, "CPU Architecture Based on Static Hardware Scheduler Engine and Multiple Pipeline Registers", accepted for presentation: 19th International Conference on Advanced Computing Systems and Microarchitecture (ICACSM), Zurich, Switzerland, September 15 - 16, 2017.
56. **I. Zagan**, N. C. Găitan, and V. G. Găitan, "An Approach of nMPRA Architecture using Hardware Implemented Support for Event Prioritization and Treating", International Journal of Advanced Computer Science and Applications (IJACSA), Vol. 8, No. 2, 2017, doi: 10.14569/IJACSA.2017.080206.

57. **I. Zagan**, “Real-time evaluation of nMPRA CPU Architecture based on Multithreaded Execution”, in 8th International Conference on Computer Science and Information Technology, Amsterdam, Netherlands, 10-11 Dec. 2015, doi:10.17706/ijcee.2015.7.6.424-429.
58. **I. Zagan** and V. G. Găitan, “Schedulability Analysis of nMPRA Processor based on Multithreaded Execution”, 13th International Conference on Development and Application Systems (DAS), Suceava, Romania, pp. 130-134, May 19–21, 2016, doi: 10.1109/DAAS.2016.7492561.
59. MIPS® MT Principles of Operation, MIPS Technologies, Inc. 955 East Arques Avenue Sunnyvale, CA 94085-4521, Document Number: MD00452, Revision 1.02, September 9, 2013.
60. L. Andrieș and V. G. Găitan, “Dual priority scheduling algorithm used in the nMPRA microcontrollers”, 2014 18th International Conference on System Theory, Control and Computing (ICSTCC), Sinaia, 2014, pp. 43-47, doi: 10.1109/ICSTCC.2014.6982388.
61. L. Andrieș, V. G. Găitan and E. E. Moisuc, “Programming paradigm of a microcontroller with hardware scheduler engine and independent pipeline registers - a software approach”, 2015 19th International Conference on System Theory, Control and Computing (ICSTCC), Cheile Gradistei, 2015, pp. 705-710, doi: 10.1109/ICSTCC.2015.7321376.
62. E. Moisuc, “Contribuții asupra proiectării optimizate a arhitecturii cpu bazată pe programare în timp real și regiștri paraleli (pipeline)”, teză de doctorat, Universitatea Ștefan cel Mare Suceava, România, 2016.
63. **I. Zagan** and V. G. Găitan, “Improving the Performances of the nMPRA Processor using a Custom Interrupt Management Scheduling Policy”, Advances in Electrical and Computer Engineering (AECE), vol. 16, no. 4, 30 Nov. 2016, pp. 45-50, ISSN: 1582-7445, doi:10.4316/AECE.2016.04007.
64. MicroBlaze Soft Processor Core, XILINX, Available: <https://www.xilinx.com/products/design-tools/microblaze.html>. (Accesat: May 2016).
65. A. S. R. Oliveira, L. Almeida, and A. d. B. Ferrari, “The ARPA-MT Embedded SMT Processor and Its RTOS Hardware Accelerator”, IEEE Transactions on Industrial Electronics, vol. 58, no. 3, pp. 890-904, March 2011. doi: 10.1109/TIE.2009.2028359
66. Amber Open Source Project, March 2015. Available: <http://opencores.org/websvn,filedetails?repname=amber&path=%2Famber%2Ftrunk%2Fdoc%2Famber-core.pdf> (Accesat: Nov. 2016)
67. Amber Project User Guide, May 2013. Available: <http://opencores.org/websvn,filedetails?repname=amber&path=%2Famber%2Ftrunk%2Fdoc%2Famber-user-guide.pdf> (Accesat: Nov. 2016)
68. C. A. Tănase, “An approach of MPRA technique over ARM cache architecture”, 2016 International Conference on Development and Application Systems (DAS), Suceava, Romania, 2016, pp. 86-90. doi: 10.1109/DAAS.2016.7492553
69. **I. Zagan**, V. G. Găitan, A. Petrariu and A. Brezulianu, “Healthcare IoT m-GreenCARDIO Remote Cardiac Monitoring System – Concept, Theory of Operation and Implementation”, accepted for publication in: Advances in Electrical and Computer Engineering (AECE), vol., no. , ISSN: 1582-7445, May 2017.
70. S. Ramachandran, “Digital VLSI Systems Design, Springer Netherlands”, eBook ISBN: 978-1-4020-5829-5, 2007, doi: 10.1007/978-1-4020-5829-5

ANEXA I. Sintetizarea și implementarea proiectului SoC utilizând Vivado Design Suite și Virtex-7

Mediul *Vivado® Design Suite* înlocuiește cu succes uneltele furnizate de *Xilinx ISE® Design Suite*, reprezentând un instrument esențial în facilitarea activității noastre de cercetare. Toate funcțiile și capabilitățile sunt construite direct în mediul Vivado, constituind un model de prelucrare a datelor scalabil și totodată partajat. Întregul proces de proiectare poate fi executat în memorie fără a avea nevoie să scriem sau să translatăm formate de fișiere intermediare, reducând totodată cerințele de memorie. Astfel, cu acest format nou, *Vivado® Design Suite* este proiectat în scopul de a îmbunătăți productivitatea generală în proiectarea, integrarea și implementarea sistemelor folosind dispozitivele Xilinx® 7 series, Zynq®-7000 All Programmable (AP) SoC și UltraScale™. Dispozitivele Xilinx au devenit mult mai vaste, venind cu o varietate de noi tehnologii inclusiv *stacked silicon interconnect* (SSI), interfața I/O de mare viteză (până la 28 GB), procesarea semnalelor analogice mixte cât și microprocesoare implementate în hardware. Aceste dispozitive complexe implică existența unor provocări de proiectare multi-dimensionale, facilitând accelerarea timpilor de lansare pe piață și creșterea productivității. Mediul Vivado furnizează o analiză completă a proiectului la nivelul fiecărui etaj al implementării, permițând astfel modificări de proiectare și configurare cu un impact pozitiv asupra întregului proiect, reducând iterațiile de proiectare și accelerarea productivității.

Elaborarea proiectului transformă codul programatorului reprezentând nivelul RTL într-o varietate de alte reprezentări vizuale echivalente. Astfel, se poate vizualiza procesorul în diferite stagii de proiectare cu ajutorul opțiunilor *RTL Netlist*, *Schematic* și *Graphical Hierarchy*, cu posibilitatea de depanare și verificare folosind proprietatea *cross-select*. Următorul pas este reprezentat de procesul de sintetizare a proiectului. Această etapă constă în transformarea proiectului de la nivelul RTL în reprezentarea la nivel de porți logice. Așadar, aceasta înseamnă că din codul Verilog împreună cu librăriile standard UNISIM obținem ca rezultat netlista la nivel de porți logice, conținând primitive precum *Flip-Flops* (FFs) sau *Look-Up Tables* (LUTs). Pentru a lansa sinteza proiectului trebuie să selectăm din meniul Flow Navigator opțiunea Synthesis și mai apoi Run Synthesis. Dacă procesul de sintetizare s-a finalizat fără erori, putem deschide proiectul astfel obținut selectând *Open Synthesized Design*. Astfel, se poate observa aria circuitului FPGA cât și resursele care au fost alocate până în acel punct. Prin intermediul opțiunii *Zoom In* putem observa pinii I/O selectați în fișierul de constrângeri și buffer-ele semnalului de ceas, iar selectând resursele evidențiate putem obține mai multe informații. Pentru a vizualiza și analiza utilizarea resurselor (Flip-Flops, Look-Up Tables, I/Os, BUFGs și MMCMs) corespunzătoare implementării proiectului, trebuie să accesăm meniul *Utilization* din tab-ul *Project Summary*. Următorul pas în validarea procesorului este reprezentat de implementarea proiectului. Acest proces constă din operațiile de plasare și trasare, care împreună cu algoritmi corespunzători pune elementele netlistei în circuitul FPGA și le conectează împreună astfel încât toate cerințele să fie îndeplinite. Acest pas poate fi unul destul de lent, mai ales în cazul folosirii instrumentelor de depanare hardware precum analizorul ChipScope.

În cele ce urmează se va descrie procesul de descărcare a fișierului bitstream în circuitul FPGA prin intermediul conexiunii USB-to-JTAG. O dată ce driver-ul pentru interfața USB-to-JTAG este instalat iar setările bitstream sunt selectate corespunzător, putem trece la generarea și programarea fișierului *.bit*. În mod implicit, comanda *Tool Command Language* (Tcl) *write_bitstream* va genera numai fișierul *.bit* (*binary bitstream*). Pentru a realiza acest pas, din meniul Program and Debug accesăm opțiunea *Generate Bitstream*. După terminarea procesului de generare a fișierului *.bit*, trebuie să obținem fișierul denumit *nMPRA.bit*, acesta fiind generat în directorul “\\MPRA_v47\project_1\project_1.runs\impl_2”.

Pentru programarea circuitului FPGA trebuie să ne asigurăm că poziția celor cinci linii ale selectorului DIP SW11 este cea corectă (*00101*), precum este ilustrat în Figura I-1, după care putem trece la pasul următor ce constă din alimentarea kit-ului de dezvoltare cu tensiunea de alimentare.

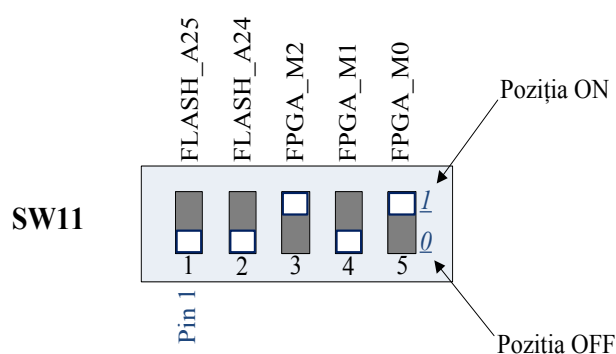


Figura I-1 Configurația selectorului DIP SW11 pentru modificarea setărilor JTAG

Instrumentul Vivado IDE include funcționalități care permit programatorului să se conecteze la dispozitive hardware care conțin unul sau mai multe circuite FPGA, în scopul de a programa și interacționa cu toate acestea. Acest lucru se poate realiza utilizând comenzi Tcl sau prin interfața grafică parcurgând următorii pași:

- Deschiderea sesiunii hardware reprezintă primul pas în programarea și depanarea procesorului în hardware. Pentru aceasta trebuie să selectăm din meniul *Program and Debug* opțiunea *Open Hardware Session*.
- Deschiderea hardware-ului țintă ce conține un canal JTAG a unuia sau a mai multor dispozitive FPGA, administrat de către serverul *vcse_server*. Pentru aceasta se poate selecta redeschiderea unei conexiuni deja folosite în precedență selectând *Open Recent Hardware Target* din fereastra Hardware, sau adăugarea unei conexiuni noi utilizând opțiunea *Open New Hardware Target wizard*. Pentru aceasta trebuie să localizăm kit-ul de dezvoltare VC707 în localhost.
- Asocierea fișierului de date pentru programare cu dispozitivul FPGA corespunzător, cât și fișierul de debug în câmpul *Probes file* dacă este cazul. După conectarea dispozitivului hardware, se va selecta și verifica fișierul *.bit* necesar pentru programare, în cazul în care acest fișier nu este selectat automat.
- Programarea sau descărcarea fișierului *.bit* în dispozitivul hardware. Odată ce fișierul pentru programare a fost asociat cu dispozitivul hardware, se poate realiza descărcarea acestuia efectuând click dreapta pe dispozitivul xc7vx485t_0 în fereastra *Hardware Manager* și selectând opțiunea de meniu *Program Device* (sau utilizând comanda Tcl

program_hw_device). Pentru a verifica dacă dispozitivul hardware a fost programat cu succes se poate examina starea biților din fereastra *Hardware Device Properties*, după cum se poate observa în Figura I-2.

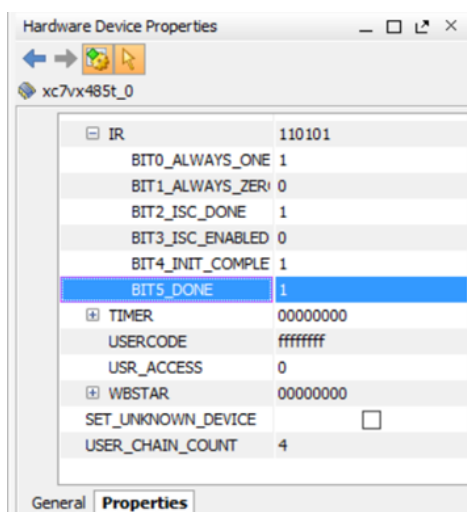


Figura I-2 Starea bitului BIT5_DONE în cazul în care circuitul FPGA a fost programat cu succes

Placa de dezvoltare VC707 proiectată pentru circuitul FPGA Virtex-7 asigură o platformă hardware robustă pentru implementarea și evaluarea proiectelor care vizează circuitul FPGA Virtex-7 XC7VX485T-2FFG1761C.

Pentru a testa următoarele exemple este necesar de un PC pe care rulează un sistem de operare compatibil. PC-ul trebuie să dispună de un port USB, interfață Ethernet, având instalat Vivado Design Suite și driver-ul *Silicon Laboratories CP210x VCP*. Proiectul BIST folosește un program terminal pentru a comunica între PC-ul gazdă și kit-ul VC707. Pentru aceasta, trebuie să conectăm mai întâi placa VC707 la PC-ul gazdă după cum se poate observa în Figura I-3.



Figura I-3 Conectarea PC-ului gazdă la kit-ul de evaluare prin intermediul portului COM

Programul terminal afișează meniul BIST după cum se poate observa în Figura I-4. În cele ce urmează trebuie ales un număr corespunzător testului de efectuat. Pentru a realiza un alt test, în condițiile în care circuitul FPGA a răspuns cu succes, trebuie să acționăm o tastă pentru a afișa din nou meniul inițial.

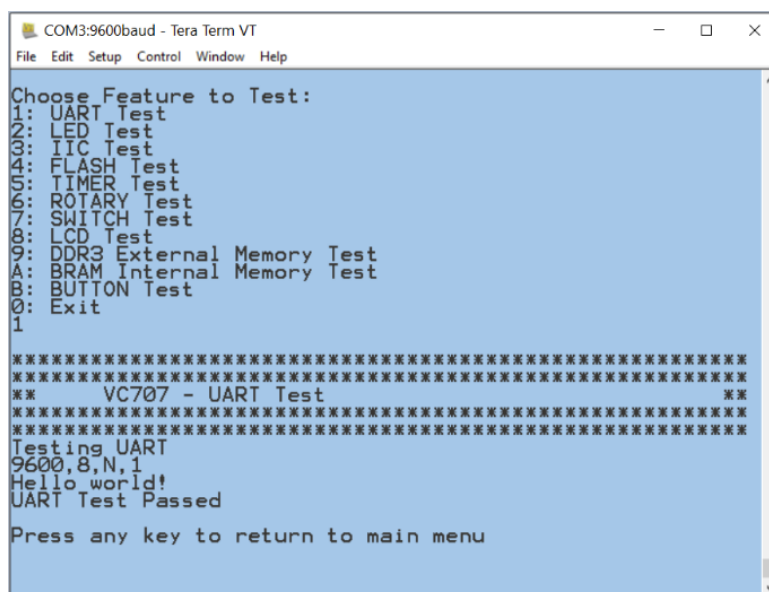


Figura I-4 Meniul proiectului BIST și testarea interfeței UART

După implementarea cu succes a procesorului nMPRA, următorii pași constau din rularea proiectului în hardware programând circuitul FPGA și depanarea procesorului folosind analizorul ChipScope. Toate aceste comenzi sunt disponibile în secțiunea *Program and Debug* din fereastra *Flow Navigator*. Programarea hardware a procesorului constă din generarea fișierului de date pentru programat corespunzător proiectul implementat, conectarea la hardware-ul VC707 și trimiterea fișierului .bit către dispozitivul FPGA țintă. Depanarea proiectului FPGA este un proces interactiv format din mai mulți pași. Acest proces poate fi împărțit în mai multe secțiuni, în scopul de a testa separat mai multe părți simple și nu de a accesa direct întregul proiect. Metodologia de proiectare și testare poate fi orice combinație a următoarelor stagii din cadrul fluxului de proiectare:

- Simularea proiectului la nivelul RTL;
- Simularea proiectului Post-implemented;
- Depanarea In-system.

Implementarea proiectului SoC și validarea procesorului nMPRA cu ajutorul nucleelor de depanare se face accesând opțiunea *Run Implementation* din mediul Vivado IDE. O dată ce avem introduse în proiect nucleele de debug, putem folosi caracteristicile *run-time logic analyzer* pentru a depana proiectul în hardware.

Astfel, se poate lansa analizorul ChipScope direct din Vivado IDE în orice proiect implementat pe care a fost rulată opțiunea *Generate Bitstream*. Dacă aplicația software ChipScope Pro Analyzer este prezentă în PC, putem folosi unelte specifice să interacționăm cu nucleele ILA v1.x existente în proiectul implementat. De asemenea, o dată ce avem setat proiectul final împreună cu serverele *vcse_server*, urmând următorii pași putem folosi nucleul de debug hardware ILA v2.0 pentru a depana proiectul:

- Lansăm Vivado IDE în modul GUI;
- Conectăm kit-ul de dezvoltare folosind un cablu corespunzător pentru JTAG Chain (Figura I-5);

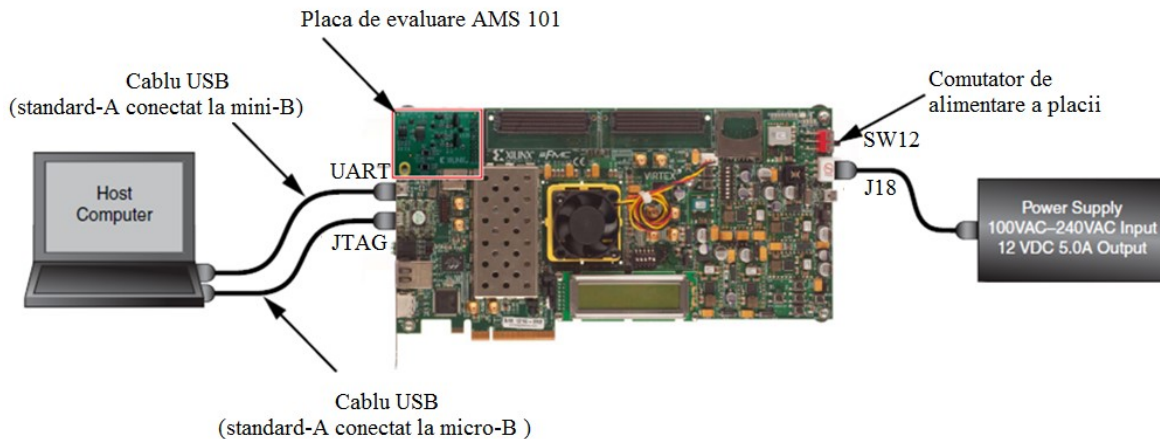


Figura I-5 Conexiunile necesare pentru depanarea proiectului SoC

- Accesăm Open Hardware Session din Flow Navigator;
- Ne conectăm la serverul *vcse_server* care rulează în localhost:50001 și programăm dispozitivul FPGA;
- Lansăm hw_ila_1 (u_ila_0) din meniul Hardware Manager;
- Stabilim semnalele și momentele în care dorim să obținem informații;
- Analizăm formele de undă ale procesorului nMPRA precum și comportamentul întregului proiect SoC (Figura I-6).

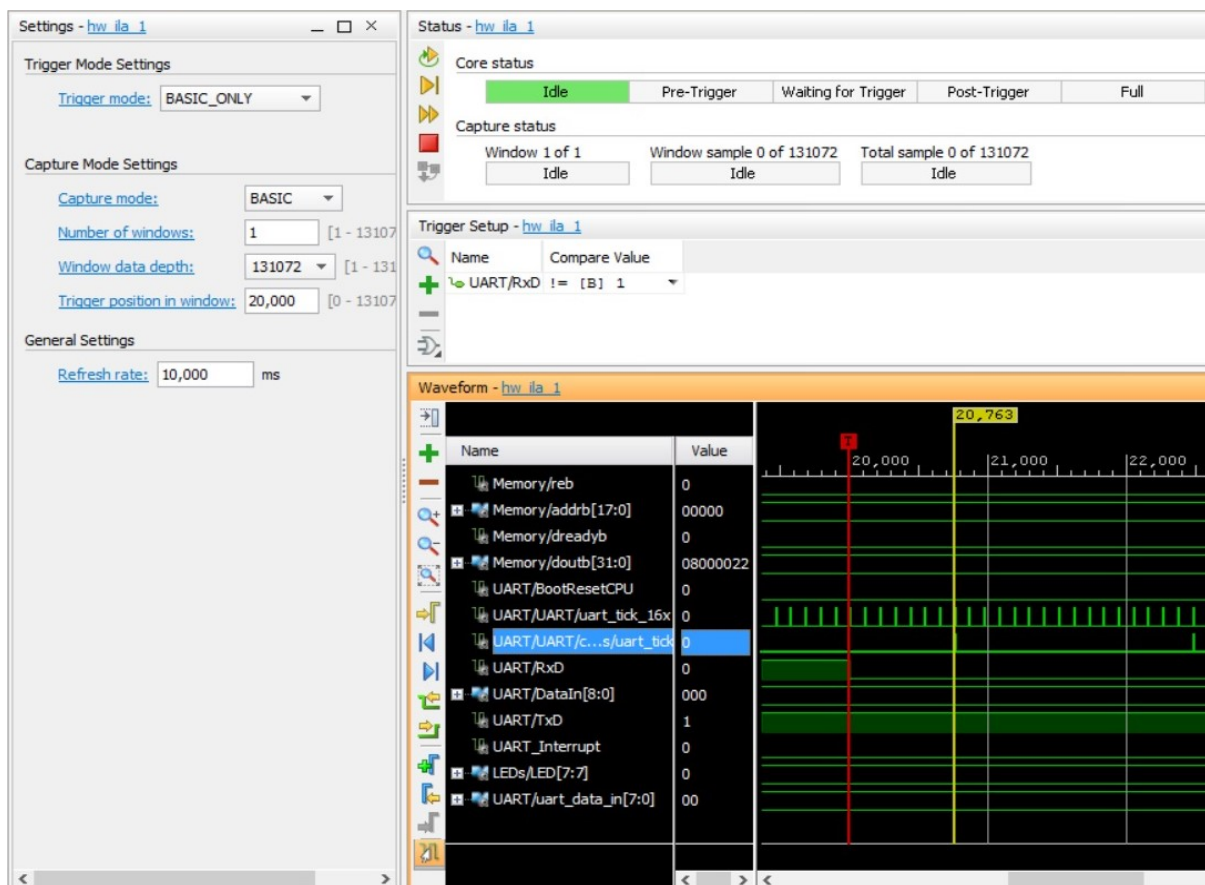
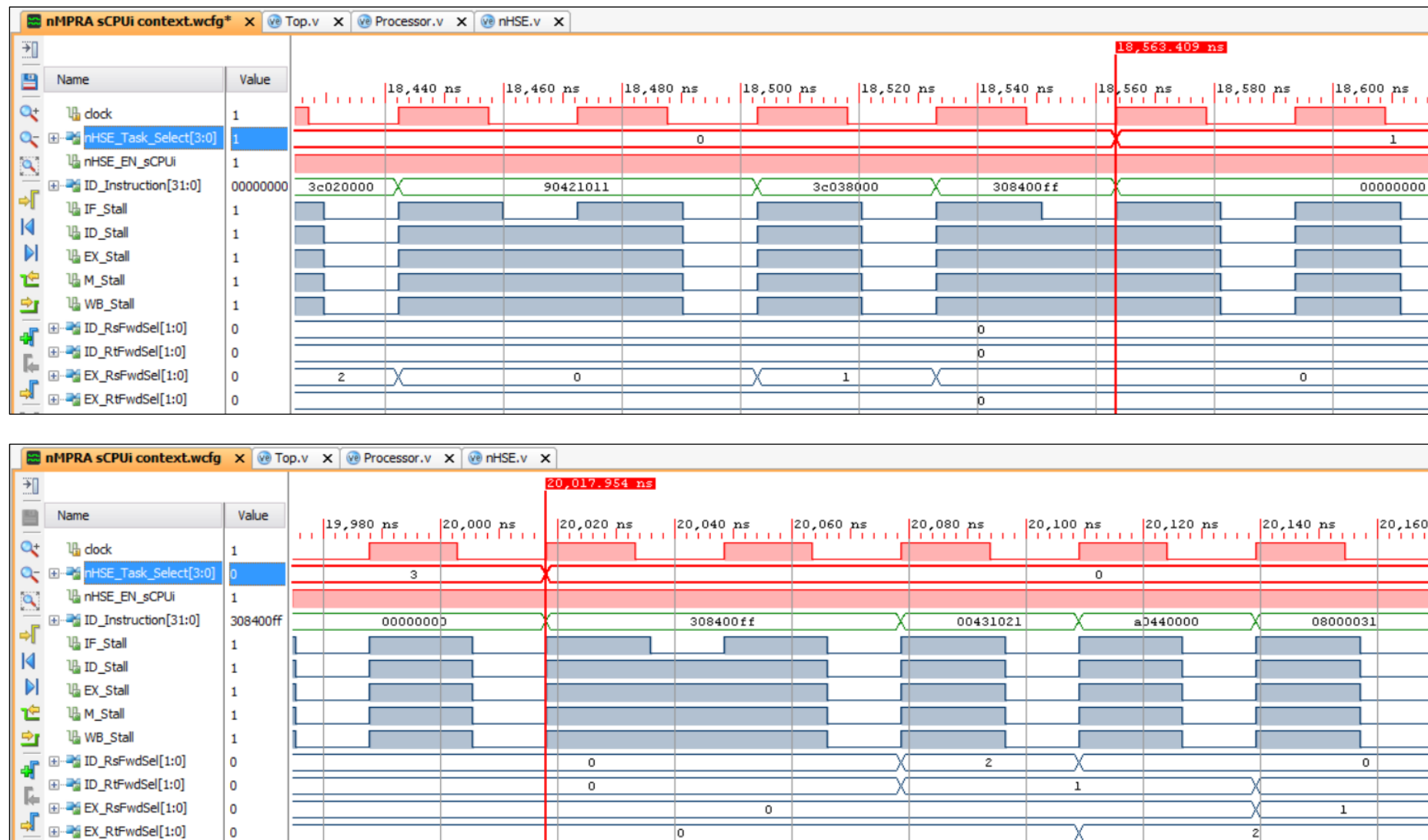
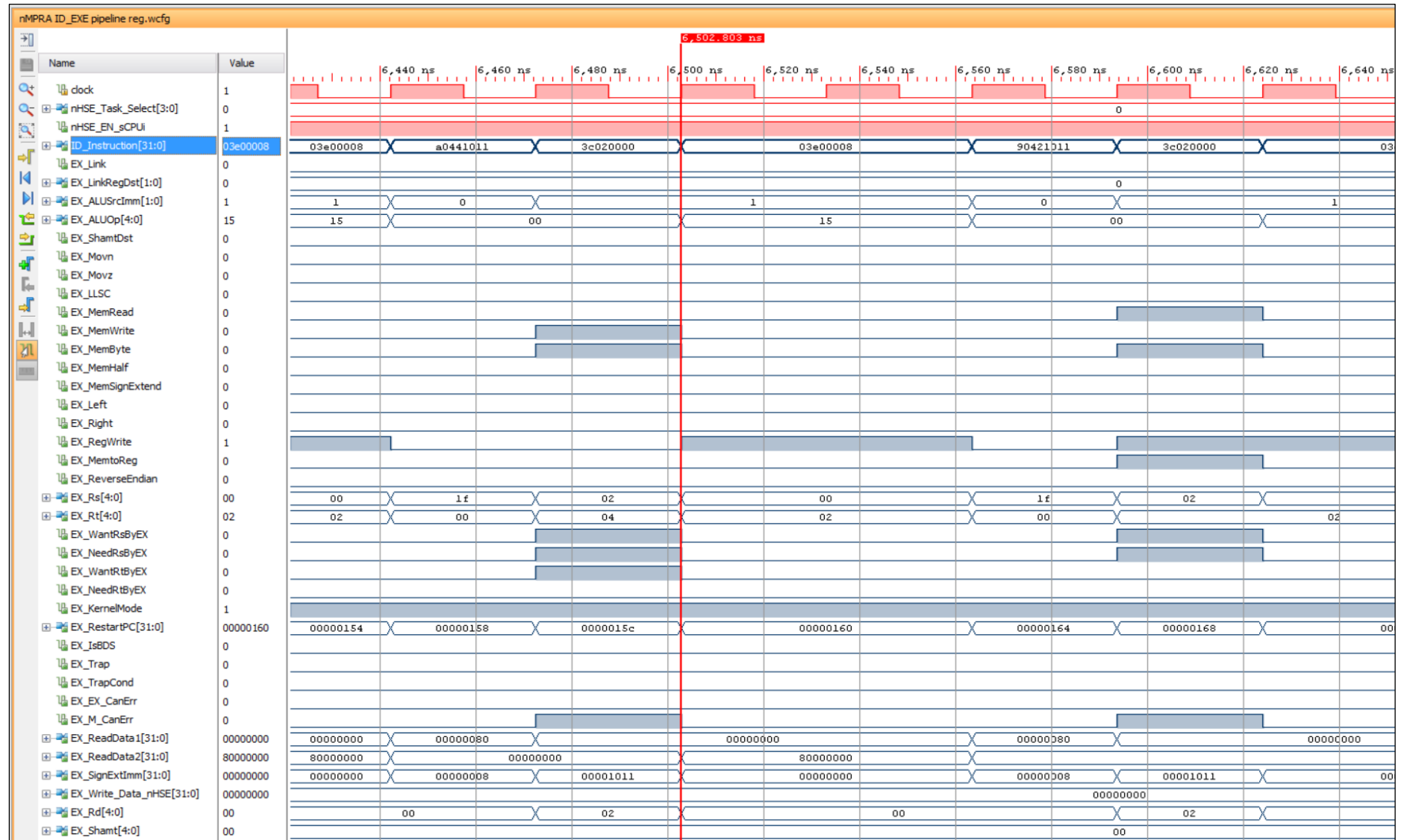


Figura I-6 Vizualizarea semnalelor corespunzătoare comunicației UART prin intermediul nucleelor de depanare hardware ILA v2.0

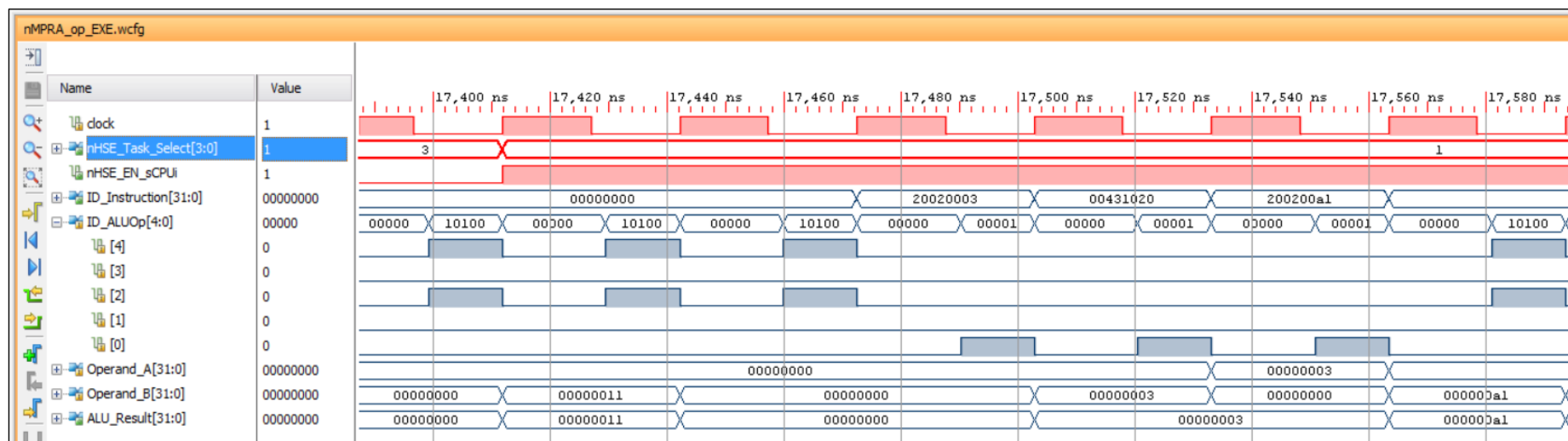
ANEXA II. Figura 1. Schimbările de context dintre sCPU0-sCPU1 și sCPU3-sCPU0



ANEXA II. Figura 2. Semnalele de ieșire ale etajului pipeline ID/EX



ANEXA II. Figura 3. Operanzii furnizați unității aritmetice și logice



ANEXA III. Secvență din codul Verilog pentru implementarea planificatorului nHSE

//-----Logica FSM pentru stările nHSE-----

```

case (nHSE_FSMstate)
  FSM_IDLE: begin
    nHSE_FSMstate <= (nHSE_inhibit_CC)? FSM_IDLE : FSM_WAIT; // nHSE_inhibit - activ pe 1 logic
  end
  FSM_WAIT: begin
    //nHSE_FSMstate <= (sched_en)? FSM_sCPU0 : FSM_WAIT; //nHSE dezactivat de instructiunea nHSE_en
    if(sched_en) begin
      //in functie de planificator verificam daca a aparut un eveniment la oricare din semiprosesoare
      if(StaticDynamic_Scheduler)begin // = 1 > planificator dinamic
        if(sCPUi_Lower_PRI!=32'hFFFFFFFF)begin
          for(i=0;i<NR_TASKS;i=i+1)begin
            if(sCPUi_Lower_PRI == mrPRI[sCPUi[i]])begin
              nHSE_FSMstate = FSM_sCPUi[i]; // FSM_sCPU0;
            end
          end
        end
      end
      else nHSE_FSMstate = FSM_WAIT;

    end
  else begin // = 0 > planificator static //la sCPU0 nu verificam bitul cr0MSTOP
    if ((crTRi[sCPU0_ID] & crEVi[sCPU0_ID])!= 32'h00000000) begin
      //daca avem cel putin un ev. validat SI a aparut ev. respectiv (eliminat:((crTRi[sCPU0_ID] & 32'h00000080) ==
      32'h00000080))

      nHSE_FSMstate = FSM_sCPU0; // sCPU0 ruleaza deja sau trebuie sa tratam un ev.
    end //altfel verificam celelalte sCPUi daca indeplinesc conditiile de executie
    else if (((cr0MSTOP & Mask1_bit1)!=32'h00000000) && ((crTRi[sCPU1_ID] & crEVi[sCPU1_ID])!= 32'h00000000))

  begin
    //daca sCPU1 nu este oprit din executie si daca avem cel putin un ev. validat SI a aparut ev. respectiv

```

```

        nHSE_FSMstate = FSM_sCPU1;
    end
    else if(((cr0MSTOP & Mask1_bit2)!=32'h00000000) && ((crTRi[sCPU2_ID] & crEVi[sCPU2_ID])!= 32'h00000000))
begin
        //daca sCPU2 nu este oprit din executie si daca avem cel putin un ev. validat SI a aparut ev. respectiv
        nHSE_FSMstate = FSM_sCPU2;
    end
    else if(((cr0MSTOP & Mask1_bit3)!=32'h00000000) && ((crTRi[sCPU3_ID] & crEVi[sCPU3_ID])!= 32'h00000000))
begin
        //daca sCPU3 nu este oprit din executie si daca avem cel putin un ev. validat SI a aparut ev. respectiv
        nHSE_FSMstate = FSM_sCPU3;
    end
    else nHSE_FSMstate = FSM_WAIT; //altfel nu este selectat nici un sCPUi

    end //end Planificator static
end //end ~ nHSE activat
else begin
    nHSE_FSMstate = FSM_WAIT;
end
end
FSM_sCPU0: begin
    if(sched_en) begin
        if(nHSE_inhibit_CC) begin //daca avem semnalul nHSE_inhibit activ nu facem schimbarea de context
            nHSE_FSMstate = FSM_sCPU0;
        end
        else begin //realizam schimbarea de context - daca este cazul
            if(StaticDynamic_Scheduler)begin // = 1 > planificator dinamic
                if(sCPUi_Lower_PRI!=32'hFFFFFFFF)begin
                    for(i=0;i<NR_TASKS;i=i+1)begin
                        if(sCPUi_Lower_PRI == mrPRI_sCPUi[i])begin
                            nHSE_FSMstate = FSM_sCPUi[i]; // FSM_sCPU0;
                        end
                    end
                end
            end
        end
    end
end

```

```

        end
    end
    else nHSE_FSMstate = FSM_WAIT;

    end
    else begin // = 0 > planificator static
        if ((crTRi[sCPU0_ID] & crEVi[sCPU0_ID])!= 32'h00000000) begin
            //daca sCPU0 nu este oprit din executie si daca avem cel putin un ev. validat SI a aparut ev. respectiv
            (eliminat:((crTRi[sCPU0_ID] & 32'h00000080) == 32'h00000080))
            nHSE_FSMstate = FSM_sCPU0; // sCPU0 ruleaza deja sau trebuie sa tratam un ev.
        end //altfel verificam celelalte sCPUi daca indeplinesc conditiile de executie
        else if (((cr0MSTOP & Mask1_bit1)!=32'h00000000) && ((crTRi[sCPU1_ID] & crEVi[sCPU1_ID])!=
32'h00000000)) begin
            //daca sCPU1 nu este oprit din executie si daca avem cel putin un ev. validat SI a aparut ev. respectiv
            nHSE_FSMstate = FSM_sCPU1;
        end
        else if (((cr0MSTOP & Mask1_bit2)!=32'h00000000) && ((crTRi[sCPU2_ID] & crEVi[sCPU2_ID])!= 32'h00000000))
begin
            //daca sCPU2 nu este oprit din executie si daca avem cel putin un ev. validat SI a aparut ev. respectiv
            nHSE_FSMstate = FSM_sCPU2;
        end
        else if (((cr0MSTOP & Mask1_bit3)!=32'h00000000) && ((crTRi[sCPU3_ID] & crEVi[sCPU3_ID])!= 32'h00000000))
begin
            //daca sCPU3 nu este oprit din executie si daca avem cel putin un ev. validat SI a aparut ev. respectiv
            nHSE_FSMstate = FSM_sCPU3;
        end
        else nHSE_FSMstate = FSM_WAIT; //altfel nu este selectat nici un sCPUi

        end //end Planificator static
    end //end ~inhibit
end //end ~ nHSE activat
else begin

```

```

        nHSE_FSMstate = FSM_WAIT;
    end
end
FSM_sCPU1: begin
    if(sched_en) begin
        if(nHSE_inhibit_CC) begin //daca avem semnalul nHSE_inhibit activ nu facem schimbarea de context
            nHSE_FSMstate = FSM_sCPU1;
        end
        else begin //realizam schimbarea de context - daca este cazul
            if(StaticDynamic_Scheduler)begin //planificator dinamic
                if(sCPUi_Lower_PRI!=32'hFFFFFFFF)begin
                    for(i=0;i<NR_TASKS;i=i+1)begin
                        if(sCPUi_Lower_PRI == mrPRI_sCPUi[i])begin
                            nHSE_FSMstate = FSM_sCPUi[i]; // FSM_sCPU0;
                        end
                    end
                end
            end
            else nHSE_FSMstate = FSM_WAIT;
        end
        else begin //planificator static
            if ((crTRi[sCPU0_ID] & crEVi[sCPU0_ID])!= 32'h00000000) begin
                //daca sCPU0 nu este oprit din executie si daca avem cel putin un ev. validat SI a aparut ev. respectiv
                (eliminat:((crTRi[sCPU0_ID] & 32'h00000080) == 32'h00000080))
                nHSE_FSMstate = FSM_sCPU0; // sCPU0 ruleaza deja sau trebuie sa tratam un ev.
            end //altfel verificam celelalte sCPUi daca indeplinesc conditiile de executie
            else if (((cr0MSTOP & Mask1_bit1)!=32'h00000000) && ((crTRi[sCPU1_ID] & crEVi[sCPU1_ID])!=
32'h00000000)) begin
                //daca sCPU1 nu este oprit din executie si daca avem cel putin un ev. validat SI a aparut ev. respectiv
                nHSE_FSMstate = FSM_sCPU1;
            end
            else if(((cr0MSTOP & Mask1_bit2)!=32'h00000000) && ((crTRi[sCPU2_ID] & crEVi[sCPU2_ID])!= 32'h00000000))
begin

```